Implementing WiFi ax in SDR

Elijah Barker

A Senior Thesis submitted in partial fulfillment
of the requirements for graduation
in the Honors Program
Liberty University
Spring 2019

Acceptance of Senior Honors Thesis

This Senior Honors Thesis is accepted in partial fulfillment of the requirements for graduation from the Honors Program of Liberty University.

____
Kyung Kyoon Bae, Ph.D.
Thesis Chair

____
Feng Wang, Ph.D.
Committee Member

____
Allen A. Harper, Ph.D.
Committee Member

____
David Schweitzer, Ph.D.
Assistant Honors Director

____
Date

Abstract

Both Carrier Sensing Multiple Access (CSMA) and Orthogonal Frequency Division Multiple Access (OFDMA) are vital techniques for WiFi radio operations. CSMA deals with decentralized sharing of the medium, and OFDMA deals with dividing up the channel into multiple smaller allocations of the channel to transmit data from multiple users simultaneously. OFDMA is a new multiple access scheme introduced in the upcoming IEEE 802.11ax standard. This paper details research with Dr. Bae in implementing portions of the upcoming 802.11ax standard using software defined radio. First, this paper provides in-depth setup information for the Wime Project and explains the Wime Project implementation in great detail. Second, this paper provides pseudo code for how to implement CSMA and an attempt at implementing OFDMA.

Keywords: SDR, CSMA, OFDMA, 802.11ax, Wime Project

Implementing WiFi ax in SDR

Introduction

Both Carrier Sensing Multiple Access (CSMA) and Orthogonal Frequency Division Multiple Access (OFDMA) are vital techniques for the IEEE 802.11ax standard. CSMA deals with decentralized sharing of the transmission medium, and OFDMA deals with splitting up the channel into multiple smaller sections of the channel to transmit data from multiple users at the same time. OFDMA, a multiple access scheme, is introduced in the upcoming IEEE 802.11ax standard. This Honors Thesis details research with Dr. Bae in implementing some parts of the upcoming IEEE 802.11ax standard using software defined radio (SDR). First, this paper provides a detailed setup process for the Wime Project and explains the Wime Project implementation of current WiFi in great depth. Second, this paper provides pseudocode for a CSMA algorithm and an attempt at implementing OFDMA that builds on the Wime Project.

The Wime Project is an open-source repository of code blocks for GNU Radio. These blocks have been linked together in GNU Radio with an example transmitter and receiver functionality. The blocks edited in attempt to implement OFDMA are from the Wime Project. In order to get a baseline to build from, this paper covers a working setup of the current software defined radio hardware in the engineering department. Ubuntu Linux served as the operating system to ease development.

Existing Code

This section covers the programming environment and getting the existing code in the repository working. The skills required to get everything in the software to work properly include knowledge of specific parts of communications and networking. Knowledge of Linux kernel parameters, basic networking and routing, ARP, and MAC addresses was required in order to troubleshoot most of the issues to get the existing code working. Knowledge about radio modulation schemes and a basic understanding of WiFi channels is necessary to implement CSMA and OFDMA once the existing software is fully working. Increased knowledge of the network stack is a byproduct of an attempt to implement these technical functionalities.

Setup and Troubleshooting

This paper details most of the setup and troubleshooting issues encountered to get the existing code working and to modify certain parts of the code. The setup process was greatly aided by in-depth experience with Linux, programming, and use of the command line.

Installation

The first installation of the Wime Project repositories was pretty rough. It was fraught with compilation errors. Switching to a more stable branch in the repository fixed the issue. Later when installing from a newer version of the same repository, the compilation process encountered a new dependency (Swig 3.0) that was added by an update to the repository. Every time the setup was installed on a new computer, the same errors would appear, wasting time to figure out what was wrong. The following script automatically installs everything in the right places to provide an easier setup.

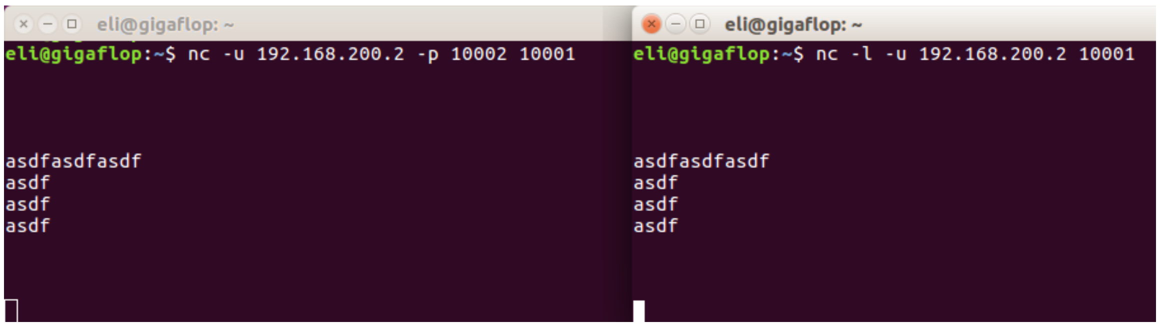
```
#!/bin/bash

echo "Are you sure you want to remove the current setup?"
echo "Enter to continue, Ctrl+C to cancel..."
read x
```

```
sudo ls > /dev/null
sudo apt-get install cmake build-essential swig git gnuradio wireshark net-
tools -y
rm -rf /home/$USER/git/gr-*
chmod +x `dirname $0`/*.sh
`dirname $0`/kill.sh

mkdir -p /home/$USER/git
pushd /home/$USER/git
git clone https://github.com/bastibl/gr-foo.git
pushd gr-foo
git checkout master
# master should work, but if not, uncomment specific commit that worked
#git checkout cb74ba2be81096213c4b981d43131e42e77ed815
mkdir build
pushd build
cmake ..
make
sudo make install
sudo ldconfig

pushd /home/$USER/git
git clone git://github.com/bastibl/gr-ieee802-11.git
pushd gr-ieee802-11
git checkout master
mkdir build
pushd build
cmake ..
make
sudo make install
sudo ldconfig
```

Previous Researcher's Progress

Another student worked on this project for Dr. Bae. There are examples in the Wime Project repo that use a UDP sink in GNU Radio that the other student tested out.

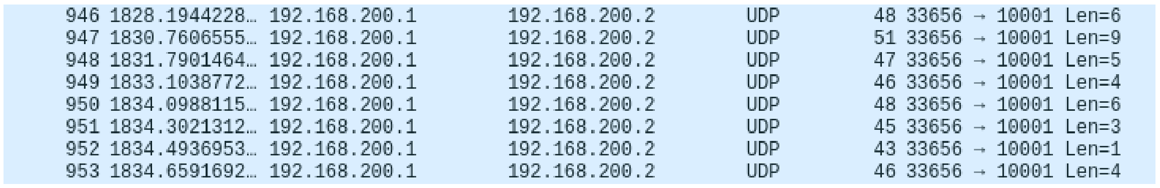
Working for short text over UDP. From the beginning, the other student had the UDP example working for small packet sizes. The picture of that setup working is shown in Figure 1. Running Wireshark (Wireshark, Version 2), the other student saw the packets going across the connection as shown in Figure 2.



```
eli@gigaflop:~$ nc -u 192.168.200.2 -p 10002 10001



asdfasdfasdf
asdf
asdf
asdf
```

```
eli@gigaflop:~$ nc -l -u 192.168.200.2 10001



asdfasdfasdf
asdf
asdf
asdf
```

Figure 1. UDP connection working for small packets



```
946 1828.1944228… 192.168.200.1        192.168.200.2        UDP         48 33656 → 10001 Len=6
947 1830.7606555… 192.168.200.1        192.168.200.2        UDP         51 33656 → 10001 Len=9
948 1831.7901464… 192.168.200.1        192.168.200.2        UDP         47 33656 → 10001 Len=5
949 1833.1038772… 192.168.200.1        192.168.200.2        UDP         46 33656 → 10001 Len=4
950 1834.0988115… 192.168.200.1        192.168.200.2        UDP         48 33656 → 10001 Len=6
951 1834.3021312… 192.168.200.1        192.168.200.2        UDP         45 33656 → 10001 Len=3
952 1834.4936953… 192.168.200.1        192.168.200.2        UDP         43 33656 → 10001 Len=1
953 1834.6591692… 192.168.200.1        192.168.200.2        UDP         46 33656 → 10001 Len=4
```

Figure 2. UDP packets seen in Wireshark

MTU and recursive algorithm. However, the other student noticed that when sending packets more than about 1500 bytes through the UDP connection, not all of the bytes were sent. The other student had troubleshooted the issue, and, upon realizing that the MTU was the problem, edited the blocks to recursively make smaller packets until the packets were small enough. The other student's recursive algorithm can be roughly described by the following pseudo code.

```
process_bytes(n_bytes, *bytes)
{
        n=n_bytes;
        if(n_bytes>1500)
        {
                n=1500;
        }
        do_work_of_function();
        process_bytes(n_bytes-1500, bytes+1500);
}
```

Tap Interfaces Giving Trouble

Using the TAP transceiver interface example instead of the UDP example made testing easier. The TAP interface generalized to any traffic instead of just UDP traffic, and made it easier to test different packet types, different services, and the speed of the setup in general.

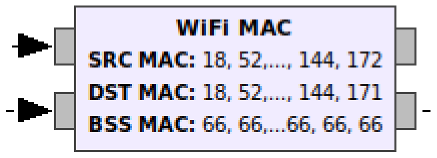
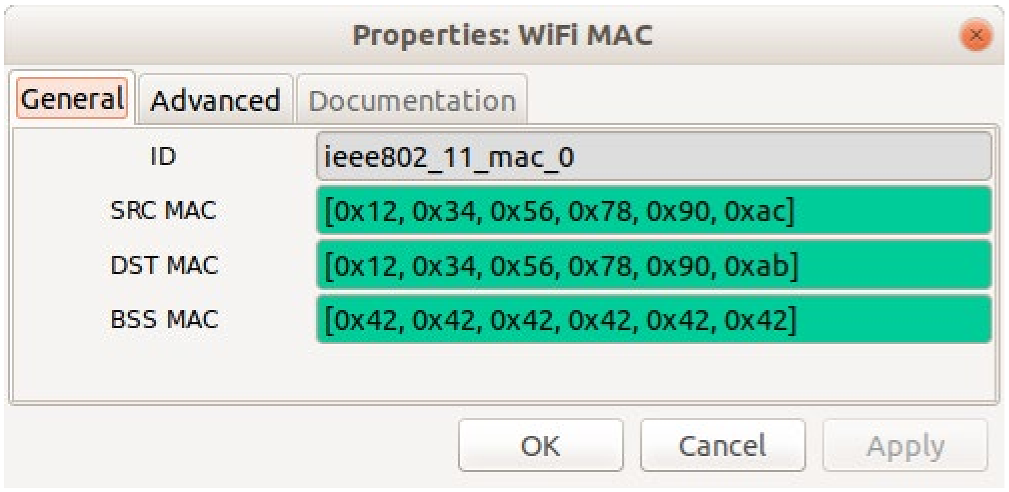

WiFi MAC
SRC MAC: 18, 52,..., 144, 172
DST MAC: 18, 52,..., 144, 171
BSS MAC: 66, 66,...66, 66, 66

Figure 3. WiFi MAC block



ID	ieee802_11_mac_0
SRC MAC	[0x12, 0x34, 0x56, 0x78, 0x90, 0xac]
DST MAC	[0x12, 0x34, 0x56, 0x78, 0x90, 0xab]
BSS MAC	[0x42, 0x42, 0x42, 0x42, 0x42, 0x42]

Figure 4. Editing the MAC address in block properties

The default setup of the TAP interface example code didn't work out of the box. The GNU Radio TAP interface, when it started up, said to set an IP, which was not hard to follow. However, the code specific to the TAP transceiver has a filter on the packets in the WiFi MAC block shown in Figure 3. Because of this filter, the MAC address of the TAP interface needed to be set to that of the WiFi MAC block. The MAC address values are changed according to python syntax in the block properties as shown in Figure 4.

The MAC address also needed to be set in the ARP table in the Linux networking stack. Because the packet is filtered by MAC address in the WiFi MAC block, source and destination MAC addresses had to be correct. Linux settings and kernel parameters related to MAC addresses and ARP—although applied—did not impact the outcome during troubleshooting. Return packet filtering is a kernel parameter in Linux that, if enabled, filters out any inbound packet that came in a different path way than a related packet was sent. Therefore, packets that get routed through different network hops are dropped for security reasons. As it turned out, there were no routing issues with the TAP setup, so the kernel parameter did not need to be changed from default.

Another parameter related to MAC addresses is promiscuous mode. A normal network interface that is not in promiscuous mode will automatically drop packets that are destined for another machine but arrived at the network interface anyway. A network interface not in promiscuous mode will compare the destination MAC address of the packet with the MAC address of the interface on which it received that packet. If the MAC addresses match, the received packet continues up the network stack. If the MAC addresses do not match, the packet is dropped. Enabling the promiscuous mode on the network interface disables the MAC filter and allows all packets through.

Massive Packet Drop

Some of the most frustrating issues are intermittent ones with no logs or errors. In this case, the issue was an intermittent transmission of packets, and an intermittent response from the Linux computer to some of those packets. For example, from the first machine, the first machine pinged the second machine across the TAP interfaces. Most of the packets crossed the WiFi channel and reached the other side (second machine) to be easily seen in Wireshark which was capturing on the TAP interface. However, the second machine was not responding to those packets. Those ICMP packets seemed to disappear somewhere in the network stack between being received and being responded to. Drop Watch (Pavel-Odintsov, 2012), a kernel module to monitor dropped packets in the kernel, logs where the packets are getting to in the kernel and where they are encountering an issue. However, compiling a kernel module is an error prone task, and the kernel module was designed for Ubuntu 14.04 instead of Ubuntu 16.04. Sure enough, a setting called lo_offset which was described in the Wime Project troubleshooting guide was the key to the packet loss. The USRP needed the frequency offset to be reasonably close in order to be able to even detect the carrier frequency offset. What still makes no sense is that the packet showed up in Wireshark when being received on the TAP interface, but Linux never responded to the packet. The lo_offset setting can be selected from a total of three options in a dropdown menu when the program is run. Changing the lo_offset to a value of 11 million fixed the packet drop issue.

TAP Interface Working

The TAP interface behaves to the Linux operating system the same as an Ethernet connection, a VPN connection, or a WiFi connection that has already been authenticated to. Because the WiFi radio connection used the TAP interface instead of the previous UDP connection, the first machine could easily send HTTP traffic across the channel to a web server hosted on the other side. By this time, the connection was working well enough to be able to SSH into one machine from the other. There was enough speed to run a lightweight graphical program over the SSH connection called xeyes. Xeyes loaded pretty quickly and was at least responsive although it lagged a bit. The xeyes program was getting approximately 2 frames per second, so the connection was at least as functional as a Dial-Up modem.

Automated run. To avoid future confusion, now that everything for the initial setup was working, the process to run the GNU Radio flowgraph is documented in a bash script shown below. The script automates the process of closing Wireshark and the previous running flowgraph, starting the GNU Radio flowgraph, adding the IP and MAC addresses to the TAP interface, and starting Wireshark again. This makes starting the flowgraph take only a few seconds as everything automatically initializes with minimal input on the user's part. Eventually, a second bash script was written for the second computer in the setup to eliminate confusion between which code was running on each computer. (The computers must have different IP addresses, and different MAC addresses from each other.) The scripts are below.

```
#!/bin/bash
# teraflop.sh
`dirname $0`/kill.sh
cp `dirname $0`/../wifi_transceiver_teraflop.py ~/git/gr-ieee802-11/examples
cd ~/git/gr-ieee802-11/examples
wireshark &
sleep 5

sudo ufw disable
sudo python -u wifi_transceiver_teraflop.py &
sleep 5
sudo ifconfig tap0 hw ether 12:34:56:78:90:ac
sudo ifconfig tap0 192.168.200.2
sudo arp -s 192.168.200.1 12:34:56:78:90:ab -i tap0
sudo arp -s 192.168.200.2 12:34:56:78:90:ac -i tap0
#sudo sysctl -w net.ipv4.conf.tap0.rp_filter=2
```

```
#sudo sysctl -w net.ipv4.conf.all.rp_filter=2
sudo ip link set dev tap0 mtu 1050
#sudo ip link set tap0 promisc on
ip a
arp -an
```

```
#!/bin/bash
# kill.sh
for i in `ps -ef | grep wifi_transceiver | grep -v grep | sed -E -e
's/[[:blank:]]+/ /g' | cut -d" " -f2`
do
      sudo kill $i
done
killall wireshark
sudo ufw enable
```

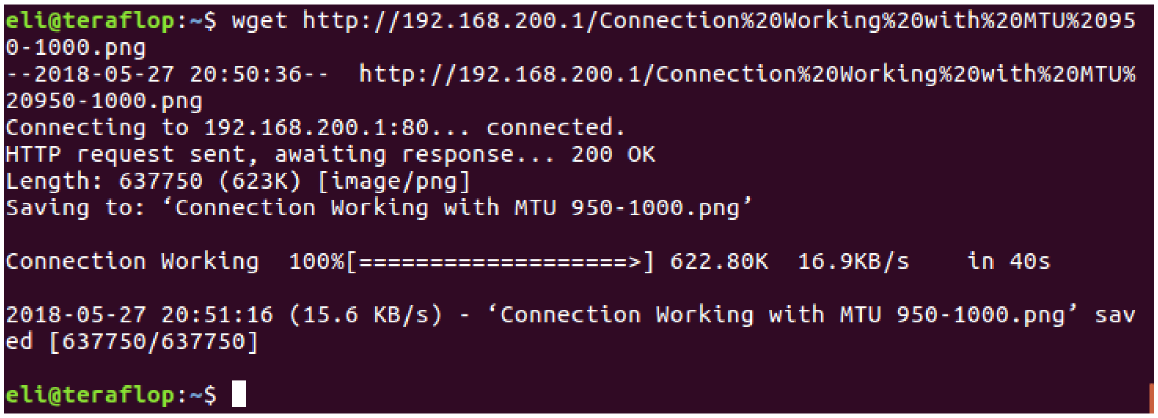
```
#!/bin/bash
# gigaflop.sh
`dirname $0`/kill.sh
cp `dirname $0`/../wifi_transceiver_gigaflop.py ~/git/gr-ieee802-11/examples
cd ~/git/gr-ieee802-11/examples
wireshark &
sleep 5

sudo ufw disable
sudo python -u wifi_transceiver_gigaflop.py &
sleep 5
sudo ifconfig tap0 hw ether 12:34:56:78:90:ab
sudo ifconfig tap0 192.168.200.1
sudo arp -s 192.168.200.1 12:34:56:78:90:ab -i tap0
sudo arp -s 192.168.200.2 12:34:56:78:90:ac -i tap0
#sudo sysctl -w net.ipv4.conf.tap0.rp_filter=2
#sudo sysctl -w net.ipv4.conf.all.rp_filter=2
sudo ip link set dev tap0 mtu 1050
#sudo ip link set tap0 promisc on
ip a
arp -an
```

Testing TAP Setups and Changing Values

The connection was stable enough to use a web browser to make the connection. Ping worked. Now that everything was working, it was time to try different settings within the working configuration. How fast can the radios communicate? Experiments with the MTU, the distance between the SDRs, the types of encoding, the sample rate, and external WiFi interference showed how robust the communication channel was.

Performance Testing Setup

When testing whether packets could even be transmitted, ping worked the best because it had a small packet size, had a 1 second polling time, and would continue to run in a loop. For example, the ping command was extremely useful when moving the USRPs around while the code was running to test the range of the USRP radio signal. When quickly testing if the channel was reliable to send data, wget would test and find out a quick estimate of the speed for a small file. For example, wget retrieved several small picture files from a web server hosted on the other end of the channel as shown in Figure 5. When a more robust and accurate method of measurement of speed is needed, iperf should be chosen over wget. Iperf and wget do not perform any type of error correction on the incoming bytes. Since both wget and iperf use TCP connections, they can however request dropped frames again, or in iperf's case, count the dropped frames. Error correction is not needed for the transport layer since the error correction happens in the physical layer using the convolutional error correcting code, the Viterbi decoder, and the checksum.



```
eli@teraflop:~$ wget http://192.168.200.1/Connection%20Working%20with%20MTU%2095
0-1000.png
--2018-05-27 20:50:36--  http://192.168.200.1/Connection%20Working%20with%20MTU%
20950-1000.png
Connecting to 192.168.200.1:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 637750 (623K) [image/png]
Saving to: ‘Connection Working with MTU 950-1000.png’

Connection Working  100%[===================>] 622.80K  16.9KB/s    in 40s

2018-05-27 20:51:16 (15.6 KB/s) - ‘Connection Working with MTU 950-1000.png’ sav
ed [637750/637750]

eli@teraflop:~$ 
```

Figure 5. Speed test using wget command

Performance results using wget. Normalized gain is the ratio of transmit samples' power to the possible power the USRP can put out. In the GUI of the running GNU Radio program, there is a slider that determines the transmit power of the radio samples. This value worked best at the default of 75%. The normalized gain worked at the maximum of 100%, but with some performance hit. When the gain was 45% the connection still worked, but barely. If the gain was any lower than that, the signal could not be detected no matter the encoding. The MTU of this test was 1050/1100.

MTU stands for Maximum Transmission Unit. The MTU was default at 390/440 for debugging purposes. In the scope of this project, an MTU of 390/440 means that the MTU of the TAP adapter in Linux was set to 390 using the ip command, and the MTU of the GNU Radio TAP block is set to 440. When the two MTUs are the same, the Linux TAP interface breaks the data into the given length but also adds the encapsulation (up to 48 bytes). The packet is then bigger than the given length in the TAP block and causes the interface to drop the packets in software. (TUN adapters don't have this MTU discrepancy since they interface with the software using full IP packet headers.)

The MTU set to 390/440 gets about 6-8 KB/s. The test was done using wget on an 11KB text file. With an MTU of 950/1000, the speed was greatly increased. The connection managed to get 309KB/s once, but the average was about 30KB/s, so the large value is assumed to be an outlier. The minimum speed for the MTU of 950/1000 is approximately 15KB/s. Most of the time there were no errors for the entire file since the file size was so small. The file size and therefore lack of errors caused an overestimation in average speed of a TCP transmission.

The connection performed nearly the same at an MTU of 1050/1100. Larger files, namely some pictures, gave a better estimate of continuous transmission speed. Pictures ranged in size up to about 600KB. However, the larger number of packet errors and retransmits from transferring a large file caused the 30KB/s to slow down to about 15KB/s. This reduction in speed confirms the fact that errors, and therefore frames dropped in the physical layer, caused the average speed of the channel for a TCP connection to decrease. The MTU of 1050/1100 was the maximum attainable MTU within 100 bytes. When the MTU was 1150/1200, neither a TCP connection nor ping worked. No packets even transmitted, no matter what encoding. These highest values either triggered some hard-coded maximum in the code or caused too many transmission errors for the Viterbi decoder to fix.

As more bits are allocated to a carrier, it becomes more difficult to distinguish between those values when they are received. That means that encodings like 16 QAM and 64 QAM are more susceptible to errors than are BPSK and QPSK. Higher encodings that are punctured with the 3/4 error correction are also more sensitive to errors and are less likely to transmit the packet without errors. With MTU 950/1000, the connection functioned with the encoding set to QPSK 1/2. If the encoding was any higher, such as QPSK 3/4, the larger packets ended up getting dropped due to there being more errors.

With the radios closer together—within about 5 feet—with MTU set to 1050/1100, QPSK 3/4 worked as well. With the SDRs at 1ft (see Table 1), QPSK 3/4 did not work again. Therefore, 5ft is the most reliable, least error-prone distance between the SDRs and is even better than having the SDRs 1ft apart.

The transmitter of one computer can send a different encoding than it is receiving (for example send BPSK and receive QPSK), and the connection will still work. This means that the receiver implementation was able to tell what encoding scheme was being used independent of the encoding it was transmitting on.

Changing the sample rate from 10MHz to 5MHz or 20MHz caused all packets to drop. Ideally, increasing the sample rate would increase the speed of packet transmission. However, the detection algorithm on the receiving end is hard-coded to work with the 10MHz sample rate. The auto-correlation was sensitive to this change in sample rate, so one would need more than just a simple tweaking of a single value to make the connection work at a different sample rate. (An MTU of 1050/1100 was used for this test.)

Performance results using iperf. Using BPSK 1/2, with the SDRs spaced at about 6 ft, the connection was able to get a maximum of 80.4 kbps. 6-7 ft was the reliable maximum separation for the SDRs. The minimum distance between the SDRs was about 1ft, with the antennas just 3 inches apart. However, QPSK 3/4 did not work at this distance. Stacking the SDRs on top of each other and putting the antennas right next to each other did not work due to the electromagnetic properties of such proximity. See Table 1 for the results of 1ft separation tests. Tests were performed with an ongoing ping in the background.

External WiFi interference from routers and computers made packets sent and received by the USRP to be not able to be detected at all. All of the 2.4 GHz WiFi access points on the Liberty-Secure campus network are either on channel 1 or channel 11 in the location the research was performed. To verify which channels were free, on a Kali Linux machine, the WiFi card was put in monitor mode, collecting SSID's, access points, and what channel they are on. Except for random people's printers and ad-hoc networks, channel 6 was free.

Table 1

SDRs at 1 ft with 3 in Between Antennas

Encoding	Speed (kbps)
BPSK 1/2	149
BPSK 3/4	157
QPSK 1/2	159
QPSK 3/4	(didn't complete) 0

In conclusion, for the most reliable transmission in further tests, one should separate the USRP radios by 5ft with line of sight, pointing the antennas in the same direction or toward each other. Use the BPSK 1/2 encoding, and the lo_offset of 11M. The channel was set to an unused WiFi channel, so interference didn't ruin any further results.

Attempt at Separating Channels (OFDMA)

Now with a reliable test environment set up, the code could be changed towards the end goal of implementing OFDMA. Before an entire parallel setup, the OFDM Carrier Allocator block was changed to simply not allocate half of the data and pilot carriers. Since this change was made on a different day than the previous benchmarks, the speed was tested again as a control group. This time, using the full channel on BPSK 1/2, with the SDRs spaced at about 6 ft, the connection was able to get a maximum of 80.4 kbps. The allocated carriers were allocated to only the upper half of the channel.

Surprisingly, network traffic was functional, and the band was automatically detected, but it made sense if the encoding was also detected automatically. As covered later, this functioning setup ended up being a false hope.

More Performance Results Using Iperf

Speeds and limits of how little of the band could be used was tested. Using half of the channel, with the SDRs spaced at about 6 ft, the connection was able to get the speed results in Table 2.

Table 2

Maximum Speed of 5 Iperf Tests per Encoding Using 'Half' of the Band (6 ft)

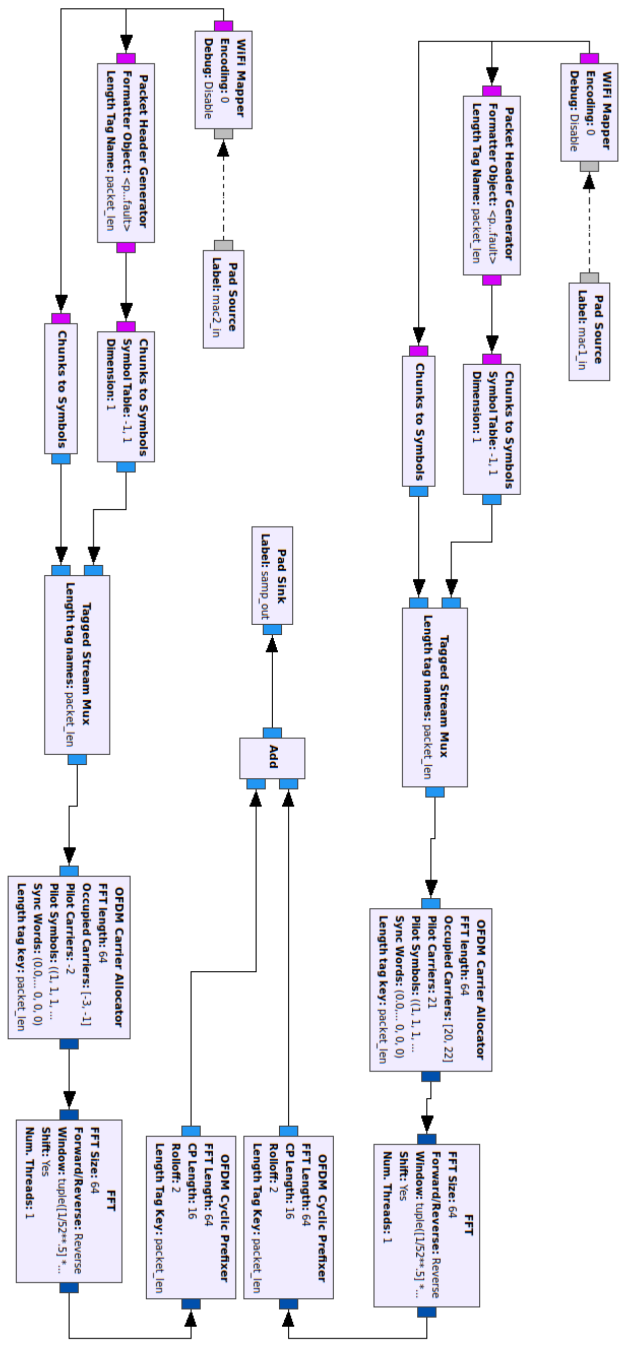
Encoding	Speed (kbps)
BPSK 1/2	67.3
BPSK 3/4	70.2
QPSK 1/2	63.9
QPSK 3/4	36.7
16QAM 1/2	(didn't complete) 0

Surprisingly, the speed was only decreased from the full channel speeds by a fraction of the original speed. It turns out that as long as the pilot carriers were surrounded on either side by the allocated carriers, the code compiled, and data was transmitted. At this point, all that was needed was to devise a method to combine and separate the signals at the source and destination respectively. However, the speed results were disconcerting because there was still no degradation of performance even when reducing the number of carriers to 2, with 1 pilot carrier in the middle.

Not Editing Block Properly

Lack of performance degradation lead to suspicsion that either the bottleneck is in the software's ability to handle incoming radio samples, or the carriers were not actually allocated as presumed. Nowhere else seemed to determine which carriers are assigned. GNU Radio documentation for the OFDM Carrier Allocator block which stated how carriers were allocated. From this information, it appeared that it had been done correctly and that the issue was a bottleneck in the software. It actually turned out that the carriers were not allocated as presumed.

This unfortunate fact was discovered when splitting and combining signals in the WiFi PHY Hierarchy block. The hierarchical block was edited to contain the logic shown in Figure 6. The inputs and outputs for the block were added and renamed, so when saved, the block should have more inputs where it is used in the broader transceiver flowgraph. However, when saved, the block in the transceiver flowgraph remained unchanged, and there were no errors when running the script. This information showed that the hierarchical block was not edited properly. The block contents were dis-abstracted and then pasted in the location of where they should go in the transceiver flowgraph, matching up inputs and outputs, and modifying variable names so the flowgraph shown in Figure 7 would execute properly. The connection finally worked. It is clear why Bloessl (2017b) used a hierarchical block, however, in this case, necessity has rendered it impractical. At this point, a full understanding of the source code of each Wime Project block is required in order to implement OFDMA. There were no checks in place that would automatically detect the OFDMA frames. This paper now provides an in-depth explanation of the source code for Bloessl's Wime Project blocks, block by block. The source code for these blocks can be found in the git repositories linked by the Wime Project website and the references in this document.


WIFI Mapper
Encoding: 0
Debug: Disable

Pad Source
Label: mac2_in

Packet Header Generator
Formatter Object: <p...fault>
Length Tag Name: packet_len

Chunks to Symbols

Chunks to Symbols
Symbol Table: -1, 1
Dimension: 1

Tagged Stream Mux
Length tag names: packet_len

OFDM Carrier Allocator
FFT length: 64
Occupied Carriers: [-3, -1]
Pilot Carriers: -2
Pilot Symbols: ((1, 1, 1, ...
Sync Words: (0.0,... 0, 0, 0)
Length tag key: packet_len

FFT
FFT Size: 64
Forward/Reverse: Reverse
Window: tuple([1/52**.5] *...
Shift: Yes
Num. Threads: 1

OFDM Cyclic Prefixer
FFT Length: 64
CP Length: 16
Rolloff: 2
Length Tag Key: packet_len

Pad Sink
Label: samp_out

Add

OFDM Cyclic Prefixer
FFT Length: 64
CP Length: 16
Rolloff: 2
Length Tag Key: packet_len

WIFI Mapper
Encoding: 0
Debug: Disable

Pad Source
Label: mac1_in

Packet Header Generator
Formatter Object: <p...fault>
Length Tag Name: packet_len

Chunks to Symbols

Chunks to Symbols
Symbol Table: -1, 1
Dimension: 1

Tagged Stream Mux
Length tag names: packet_len

OFDM Carrier Allocator
FFT length: 64
Occupied Carriers: [20, 22]
Pilot Carriers: 21
Pilot Symbols: ((1, 1, ...
Sync Words: (0.0,... 0, 0, 0)
Length tag key: packet_len

FFT
FFT Size: 64
Forward/Reverse: Reverse
Window: tuple([1/52**.5] *...
Shift: Yes
Num. Threads: 1

Figure 6. Combining logic in the hierarchical block for a transmitter

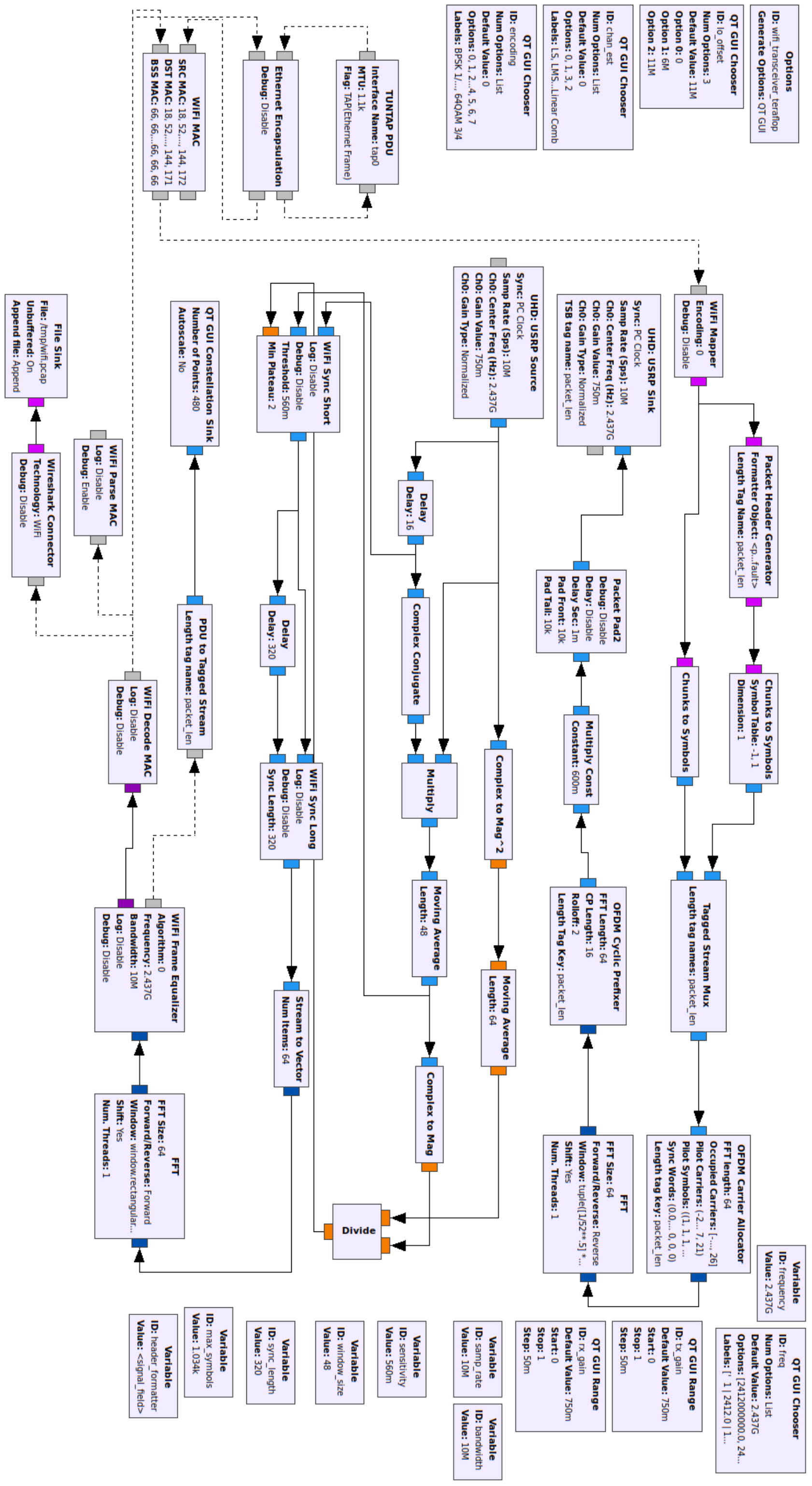


Figure 7. Hierarchy block replaced by the logic contained in the block. The flowgraph is somewhat illegible due to the number of blocks, but the general flow is still apparent.

Transmitter Blocks Source Overview

The source code for the transmitter is less complex than the source code for the receiver because the receiver must detect the signal which requires lots of fine-tuned math and error correction, etc. The transmitter can instead deterministically produce a signal on the wire without regard to noise and incoming signal.

WiFi Mapper

The source code for the WiFi Mapper block is defined in the mapper.cc file (Bloessl, 2017b). The WiFi Mapper takes in the packets from the WiFi MAC and splits them up into bite size chunks so that they are ready to be assigned to data carriers. This block is where the specific type of encoding is selected by the GUI. The radio buttons choose encodings among BPSK, QPSK, and up to 64 QAM. BPSK holds one bit per data carrier while 64 QAM holds 6 bits per data carrier. Based on the encoding, for a smaller encoding like BPSK, we will get chunks of 48 data carriers * 1 bit = 48-bit chunks. For 64 QAM, we end up with 48 data carriers * 6 bits/carrier = 288-bit chunks. This number of bits is the number of bits that are sent on a single OFDM symbol.

WiFi Mapper rearranges the bits. The WiFi Mapper first scrambles the bits. Scrambling means that each individual bit is flipped according to a somewhat chaotic, yet entirely deterministic pattern. This helps reduce the number of consecutive bits that are the same. After the scrambling, the bits are then wrapped in a convolutional error correcting code and then interleaved. Interleaving changes the positions of the bits to improve consecutive bit (burst) error correction of the convolutional code since convolutional codes are not particularly good at correcting burst errors. After the bits have been scrambled, encoded, and interleaved, WiFi Mapper splits the bits up for allocation to their intended data carriers based on the chosen OFDM symbol encoding as mentioned above. These chunks are then passed into the Chunks to Symbols blocks where the bits are converted to complex carrier symbols. The relevant source code for the WiFi Mapper block found in mapper.cc (Bloessl, 2017b) is shown below.

```
//alloc memory for modulation steps
char *data_bits        = (char*)calloc(frame.n_data_bits, sizeof(char));
char *scrambled_data   = (char*)calloc(frame.n_data_bits, sizeof(char));
char *encoded_data     = (char*)calloc(frame.n_data_bits * 2, sizeof(char));
char *punctured_data   = (char*)calloc(frame.n_encoded_bits, sizeof(char));
char *interleaved_data = (char*)calloc(frame.n_encoded_bits, sizeof(char));
char *symbols          = (char*)calloc((frame.n_encoded_bits / d_ofdm.n_bpsc),
sizeof(char));

//generate the WIFI data field, adding service field and pad bits
generate_bits(psdu, data_bits, frame);

// scrambling
scramble(data_bits, scrambled_data, frame, d_scrambler++);
if(d_scrambler > 127) {
      d_scrambler = 1;
}

// reset tail bits
reset_tail_bits(scrambled_data, frame);
// encoding
convolutional_encoding(scrambled_data, encoded_data, frame);
// puncturing
puncturing(encoded_data, punctured_data, frame, d_ofdm);
//std::cout << "punctured" << std::endl;
// interleaving
interleave(punctured_data, interleaved_data, frame, d_ofdm);
//std::cout << "interleaved" << std::endl;

// one byte per symbol
split_symbols(interleaved_data, symbols, frame, d_ofdm);

d_symbols_len = frame.n_sym * 48;

d_symbols = (char*)calloc(d_symbols_len, 1);
std::memcpy(d_symbols, symbols, d_symbols_len);
```
(Bloessl, 2017b, lines 90-124)

OFDM Carrier Allocator

The OFDM Carrier Allocator block is probably one of the easiest to understand yet useful blocks in the transmitter. Since it is a GNU Radio block, it is documented in the block properties. The Occupied Carriers are the 48 carriers dedicated to sending the data bits. The pilot carriers are 4 carriers used for synchronization at the receiver. The pilot symbols are a predetermined set of bits that will be sent across the channel on the pilot carriers.

FFT

The data going into the FFT block is in the frequency domain. Because the IFFT algorithm can only be performed on a data size of a power of 2, it is expedient to use all the slots and not have to pad the input. The FFT block performs an IFFT on the incoming symbol data. That time domain data is then passed on to the OFDM Cyclic Prefixer.

OFDM Cyclic Prefixer

The OFDM Cyclic Prefixer block prepends a wave pattern to the beginning of each piece of the modulated data for synchronization. The cyclic prefix consists of 16 samples which, when divided by the sample rate, give us $16*10^{-7} = 1.6\mu s$, which conforms to IEEE standard.

Packet Pad

The Packet Pad2 block, which is implemented in the gr-foo repository (Bloessl, 2017a) and linked by the Wime Project website, gives the receiver space to be able to recognize the packet. Without this block, the WiFi Sync Long block probably would not work properly as there is such a long packet delay that it is synchronizing with. The number of samples in front of and behind the OFDM frame is 10^3 samples which, given our sample rate, is 1ms worth of samples. Given that each OFDM frame is 8μs and the packet padding is 2ms, we can see that there is a lot of overhead in the implementation of OFDM. This is probably why the connection is so slow. Given our xeyes program (shown with the constellation in Figure 8) with graphics piped across the SSH connection, the Wireshark capture shows packet sizes for one frame as 66+110+334+102 = 612 bytes. This is 612B * 8b/B = 4896 bits. Divide that by 48 carriers and multiply by 2ms gives us $(66+110+334+102)\times 8 \div 48 \times 2 = 204$ms of air time on the SDR, not even taking into account the software delays encountered with the signal processing. This suggests that the delay of 2 frames per second or approximately 500ms per frame is a reasonable outcome for this xeyes program. The relevant source code adding padding to the bytes of the packet for the Packet Pad2 block found in packet_pad2.cc (Bloessl, 2017a) is shown below.

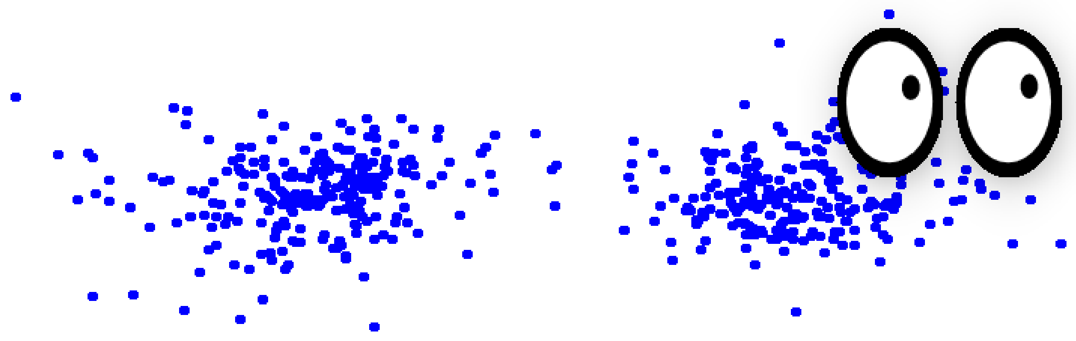


Figure 8. xeyes program on top of BPSK constellation. The groupings of dots represent a detected 1 or 0. The xeyes program is the set of eyeballs that 'looks' at the user's mouse by moving the black dots around.

```
const gr_complex *in = (const gr_complex*)input_items[0];
gr_complex *out = (gr_complex*)output_items[0];

std::memset(out, 0x00, sizeof(gr_complex) * (ninput_items[0] + d_pad_front +
d_pad_tail));

std::memcpy(out + d_pad_front, in, sizeof(gr_complex) * ninput_items[0]);


int produced = ninput_items[0] + d_pad_front + d_pad_tail;
const pmt::pmt_t src = pmt::string_to_symbol(alias());
```

(Bloessl, 2017a, lines 54-63)

Receiver Blocks Source Overview

The receiver is different from the transmitter in that it must perform synchronization steps. The samples are auto-correlated and then passed into the WiFi Sync Short block for a course frequency estimation. The signal is then passed through the WiFi Sync Long block for a fine frequency estimation. It is only after this that the signal can be demodulated and decoded.

Auto Correlation

The incoming raw digital samples from the SDR first get passed through lots of simple gnu-radio math blocks. These math blocks primarily generate the auto-correlation of the data in order to detect OFDM traffic.

There are 3 inputs to the WiFi Sync Short block, and the code described is in sync_short.cc (Bloessl, 2017b). The first is the raw input samples from the SDR. The second is called abs, but it basically serves to estimate the frequency offset. The third is the auto-correlation which is used to detect when the frame is being transmitted.

WiFi Sync Short

The code in the sync_short block first searches the auto-correlation values for a value above the threshold. When it finds the auto-correlation above the threshold, it copies over the raw input values to the output. If there happens to be another OFDM signal on top of the one it is copying, then it immediately stops so that when it is run again, it will detect the next OFDM frame right away. If there is no OFDM frame being received, no data is being forwarded to the stream. This way, the next block, the WiFi Sync Long block, gets only OFDM frames. The relevant source code for the sync_short block found in sync_short.cc (Bloessl, 2017b) is shown below.

```
switch(d_state) {

case SEARCH: {
	int i;

	for(i = 0; i < ninput; i++) {
		if(in_cor[i] > d_threshold) {
			if(d_plateau < MIN_PLATEAU) {
				d_plateau++;

			} else {
				d_state = COPY;
				d_copied = 0;
				d_freq_offset = arg(in_abs[i]) / 16;
				d_plateau = 0;
				insert_tag(nitems_written(0), d_freq_offset,
nitems_read(0) + i);
				dout << "SHORT Frame!" << std::endl;
				break;
			}
		} else {
			d_plateau = 0;
		}
	}

	consume_each(i);
	return 0;
}

case COPY: {

	int o = 0;
	while( o < ninput && o < noutput && d_copied < MAX_SAMPLES) {
		if(in_cor[o] > d_threshold) {
			if(d_plateau < MIN_PLATEAU) {
				d_plateau++;

			// there's another frame
			} else if(d_copied > MIN_GAP) {
				d_copied = 0;
				d_plateau = 0;
```

```
                          d_freq_offset = arg(in_abs[o]) / 16;
                          insert_tag(nitems_written(0) + o, d_freq_offset,
nitems_read(0) + o);
                          dout << "SHORT Frame!" << std::endl;
                          break;
                    }

             } else {
                    d_plateau = 0;
             }

             out[o] = in[o] * exp(gr_complex(0, -d_freq_offset * d_copied));
             o++;
             d_copied++;
      }

      if(d_copied == MAX_SAMPLES) {
             d_state = SEARCH;
      }

      dout << "SHORT copied " << o << std::endl;

      consume_each(o);
      return o;
}
}
```

(Bloessl, 2017b, lines 61-125)

WiFi Sync Long

The WiFi Sync Long block described in sync_long.cc (Bloessl, 2017b) is also searching through the samples, but this time it is looking for a specific sequence that is consistent with the transmitted packets. It needs to determine the beginning of the valid frame so that further blocks can perform the FFTs properly. Relevant parts of sync_long.cc are shown below for context.

```
case SYNC:
      d_fir.filterN(d_correlation, in, std::min(SYNC_LENGTH, std::max(ninput -
63, 0)));

      while(i + 63 < ninput) {

             d_cor.push_back(pair<gr_complex, int>(d_correlation[i],
d_offset));

             i++;
             d_offset++;
```

```
			if(d_offset == SYNC_LENGTH) {
				search_frame_start();
				mylog(boost::format("LONG: frame start at %1%") %
d_frame_start);
				d_offset = 0;
				d_count = 0;
				d_state = COPY;

				break;
			}
		}

		break;

case COPY:
		while(i < ninput && o < noutput) {

			int rel = d_offset - d_frame_start;

			if(!rel)  {
				add_item_tag(0, nitems_written(0),
						pmt::string_to_symbol("wifi_start"),
						pmt::from_double(d_freq_offset_short -
d_freq_offset),
						pmt::string_to_symbol(name()));
			}

			if(rel >= 0 && (rel < 128 || ((rel - 128) % 80) > 15)) {
				out[o] = in_delayed[i] * exp(gr_complex(0, d_offset *
d_freq_offset));
				o++;
			}

			i++;
			d_offset++;
		}

		break;

case RESET: {
		while(o < noutput) {
			if(((d_count + o) % 64) == 0) {
				d_offset = 0;
				d_state = SYNC;
				break;
			} else {
				out[o] = 0;
				o++;
			}
		}

		break;
}
}
```
(Bloessl, 2017b, lines 95-155)

The block performs this synchronization by first taking the correlation with a fixed set of 64 complex values. Based on the sections of samples it finds within each chunk given to it by the WiFi Sync Short block, it then determines the highest correlated data to synchronize with. Once it finds a good synchronization, it provides another frequency offset adjustment to the relevant data from the input. The frequency adjusted and synchronized data is passed to the output which then goes to the FFT block. This synchronization is shown in the following source code from sync_long.cc (Bloessl, 2017b).

```
void search_frame_start() {

      // sort list (highest correlation first)
      assert(d_cor.size() == SYNC_LENGTH);
      d_cor.sort(compare_abs);

      // copy list in vector for nicer access
      vector<pair<gr_complex, int> > vec(d_cor.begin(), d_cor.end());
      d_cor.clear();

      // in case we don't find anything use SYNC_LENGTH
      d_frame_start = SYNC_LENGTH;

      for(int i = 0; i < 3; i++) {
             for(int k = i + 1; k < 4; k++) {
                    gr_complex first;
                    gr_complex second;
                    if(get<1>(vec[i]) > get<1>(vec[k])) {
                          first = get<0>(vec[k]);
                          second = get<0>(vec[i]);
                    } else {
                          first = get<0>(vec[i]);
                          second = get<0>(vec[k]);
                    }
                    int diff  = abs(get<1>(vec[i]) - get<1>(vec[k]));
                    if(diff == 64) {
                          d_frame_start = min(get<1>(vec[i]), get<1>(vec[k]));
                          d_freq_offset = arg(first * conj(second)) / 64;
                          // nice match found, return immediately
                          return;

                    } else if(diff == 63) {
                          d_frame_start = min(get<1>(vec[i]), get<1>(vec[k]));
                          d_freq_offset = arg(first * conj(second)) / 63;
                    } else if(diff == 65) {
                          d_frame_start = min(get<1>(vec[i]), get<1>(vec[k]));
```

```
                        d_freq_offset = arg(first * conj(second)) / 65;
                    }
                }
            }
}
```
(Bloessl, 2017b, lines 179-219)

FFT

The FFT block is a built-in GNU Radio block. The FFT block converts the samples from the time domain into frequency domain because OFDM uses the samples from the frequency domain to form the constellations. Because of the synchronization performed in the WiFi Sync Long block, we end up with properly aligned and accurate OFDM symbols as the output from the FFT.

WiFi Frame Equalizer

The WiFi Frame Equalizer is partly implemented in the base.cc (Bloessl, 2017b) and ls.cc (Bloessl, 2017b) files but is mainly implemented in the frame_equalizer_impl.cc file (Bloessl, 2017b). The WiFi Frame Equalizer takes in the 64 complex values from the FFT and discards values that aren’t actual network data bits. It first discards values in the range 0-5 (the guard band), 32 (the DC offset from the carrier frequency), and 59-63 (the guard band). This is a total of 11 carriers devoted to guard band, and 1 carrier is the carrier frequency. In the frame_equalizer_impl.cc file, the WiFi Frame Equalizer also checks values on the pilot carriers against the pilot symbols it knows were transmitted to better estimate the constellations of the carrier symbols by subtracting some residual frequency offset. Next, the 4 pilot carriers are discarded. (In this case, they are 11, 25, 39, and 53). This means that a total of 11 + 1 + 4 = 16 carriers has been discarded, leaving the 64 – 16 = 48 carriers carrying the actual data bits in the physical layer of the network traffic. The relevant source code of frame_equalizer_impl.cc below shows calculating frequency offset from pilot carriers.

```
int i = 0;
int o = 0;
gr_complex symbols[48];
gr_complex current_symbol[64];

dout << "FRAME EQUALIZER: input " << ninput_items[0] << "  output " <<
noutput_items << std::endl;

while((i < ninput_items[0]) && (o < noutput_items)) {

	get_tags_in_window(tags, 0, i, i + 1,
pmt::string_to_symbol("wifi_start"));

	// new frame
	if(tags.size()) {
		d_current_symbol = 0;
		d_frame_symbols = 0;
		d_frame_mod = d_bpsk;

		d_freq_offset_from_synclong = pmt::to_double(tags.front().value)
* d_bw / (2 * M_PI);
		d_epsilon0 = pmt::to_double(tags.front().value) * d_bw / (2 *
M_PI * d_freq);
		d_er = 0;

		dout << "epsilon: " << d_epsilon0 << std::endl;
	}

	// not interesting -> skip
	if(d_current_symbol > (d_frame_symbols + 2)) {
		i++;
		continue;
	}

	std::memcpy(current_symbol, in + i*64, 64*sizeof(gr_complex));

	// compensate sampling offset
	for(int i = 0; i < 64; i++) {
		current_symbol[i] *= exp(gr_complex(0,
2*M_PI*d_current_symbol*80*(d_epsilon0 + d_er)*(i-32)/64));
	}

	gr_complex p = equalizer::base::POLARITY[(d_current_symbol - 2) % 127];
	gr_complex sum =
		(current_symbol[11] *  p) +
		(current_symbol[25] *  p) +
		(current_symbol[39] *  p) +
		(current_symbol[53] * -p);

	double beta;
	if(d_current_symbol < 2) {
```

```
                beta = arg(
                                current_symbol[11] -
                                current_symbol[25] +
                                current_symbol[39] +
                                current_symbol[53]);

        } else {
                beta = arg(
                                (current_symbol[11] *  p) +
                                (current_symbol[39] *  p) +
                                (current_symbol[25] *  p) +
                                (current_symbol[53] * -p));
        }

        double er = arg(
                        (conj(d_prev_pilots[0]) * current_symbol[11] *  p) +
                        (conj(d_prev_pilots[1]) * current_symbol[25] *  p) +
                        (conj(d_prev_pilots[2]) * current_symbol[39] *  p) +
                        (conj(d_prev_pilots[3]) * current_symbol[53] * -p));

        er *= d_bw / (2 * M_PI * d_freq * 80);

        d_prev_pilots[0] = current_symbol[11] *  p;
        d_prev_pilots[1] = current_symbol[25] *  p;
        d_prev_pilots[2] = current_symbol[39] *  p;
        d_prev_pilots[3] = current_symbol[53] * -p;

        // compensate residual frequency offset
        for(int i = 0; i < 64; i++) {
                current_symbol[i] *= exp(gr_complex(0, -beta));
        }

        // update estimate of residual frequency offset
        if(d_current_symbol >= 2) {

                double alpha = 0.1;
                d_er = (1-alpha) * d_er + alpha * er;
        }

        // do equalization
        d_equalizer->equalize(current_symbol, d_current_symbol,
                        symbols, out + o * 48, d_frame_mod);

        // signal field
        if(d_current_symbol == 2) {

                if(decode_signal_field(out + o * 48)) {

                        pmt::pmt_t dict = pmt::make_dict();
                        dict = pmt::dict_add(dict, pmt::mp("frame_bytes"),
pmt::from_uint64(d_frame_bytes));
                        dict = pmt::dict_add(dict, pmt::mp("encoding"),
pmt::from_uint64(d_frame_encoding));
                        dict = pmt::dict_add(dict, pmt::mp("snr"),
pmt::from_double(d_equalizer->get_snr()));
```

```
                        dict = pmt::dict_add(dict, pmt::mp("freq"),
pmt::from_double(d_freq));
                        dict = pmt::dict_add(dict, pmt::mp("freq_offset"),
pmt::from_double(d_freq_offset_from_synclong));
                        add_item_tag(0, nitems_written(0) + o,
                                        pmt::string_to_symbol("wifi_start"),
                                        dict,
                                        pmt::string_to_symbol(alias()));
                }
        }

        if(d_current_symbol > 2) {
                o++;
                pmt::pmt_t pdu = pmt::make_dict();
                message_port_pub(pmt::mp("symbols"), pmt::cons(pmt::make_dict(),
pmt::init_c32vector(48, symbols)));
        }

        i++;
        d_current_symbol++;
}
```

(Bloessl, 2017b, lines 118-233)

The relevant source code of frame_equalizer_impl.cc (Bloessl, 2017b) below shows the interleave pattern.

```
const int
frame_equalizer_impl::interleaver_pattern[48] = {
  0, 3, 6, 9,12,15,18,21,
 24,27,30,33,36,39,42,45,
  1, 4, 7,10,13,16,19,22,
 25,28,31,34,37,40,43,46,
  2, 5, 8,11,14,17,20,23,
 26,29,32,35,38,41,44,47};
```

(Bloessl, 2017b, lines 340-347)

The relevant source code of ls.cc (Bloessl, 2017b) below shows the pilot carriers and guard band being discarded.

```
if(n == 0) {
        std::memcpy(d_H, in, 64 * sizeof(gr_complex));

} else if(n == 1) {
        double signal = 0;
        double noise = 0;
        for(int i = 0; i < 64; i++) {
                if((i == 32) || (i < 6) || ( i > 58)) {
                        continue;
                }
                noise += std::pow(std::abs(d_H[i] - in[i]), 2);
```

```
            signal += std::pow(std::abs(d_H[i] + in[i]), 2);
            d_H[i] += in[i];
            d_H[i] /= LONG[i] * gr_complex(2, 0);
        }

        d_snr = 10 * std::log10(signal / noise / 2);

} else {

        int c = 0;
        for(int i = 0; i < 64; i++) {
            if( (i == 11) || (i == 25) || (i == 32) || (i == 39) || (i == 53)
|| (i < 6) || ( i > 58)) {
                continue;
            } else {
                symbols[c] = in[i] / d_H[i];
                bits[c] = mod->decision_maker(&symbols[c]);
                c++;
            }
        }
}
```

(Bloessl, 2017b, lines 26-56)

The relevant source code of base.cc (Bloessl, 2017b) below shows the scrambling patterns for decoding.

```
const gr_complex base::LONG[] = {
        0,  0,  0,  0,  0,  0,  1,  1, -1, -1,
        1,  1, -1,  1, -1,  1,  1,  1,  1,  1,
        1, -1, -1,  1,  1, -1,  1, -1,  1,  1,
        1,  1,  0,  1, -1, -1,  1,  1, -1,  1,
       -1,  1, -1, -1, -1, -1, -1,  1,  1, -1,
       -1,  1, -1,  1, -1,  1,  1,  1,  1,  0,
        0,  0,  0,  0
};

const gr_complex base::POLARITY[127] = {
        1, 1, 1, 1,-1,-1,-1, 1,-1,-1,-1,-1, 1, 1,-1, 1,
       -1,-1, 1, 1,-1, 1, 1,-1, 1, 1, 1, 1, 1, 1,-1, 1,
        1, 1,-1, 1, 1,-1,-1, 1, 1, 1,-1, 1,-1,-1,-1, 1,
       -1, 1,-1,-1, 1,-1,-1, 1, 1, 1, 1, 1,-1,-1, 1, 1,
       -1,-1, 1,-1, 1,-1, 1, 1,-1,-1,-1, 1, 1,-1,-1,-1,
       -1, 1,-1,-1, 1,-1, 1, 1, 1, 1,-1, 1,-1, 1,-1, 1,
       -1,-1,-1,-1,-1, 1,-1, 1, 1,-1, 1,-1, 1, 1, 1,-1,
       -1, 1,-1,-1,-1, 1, 1, 1,-1,-1,-1,-1,-1,-1,-1 };
```

(Bloessl, 2017b, lines 24-42)

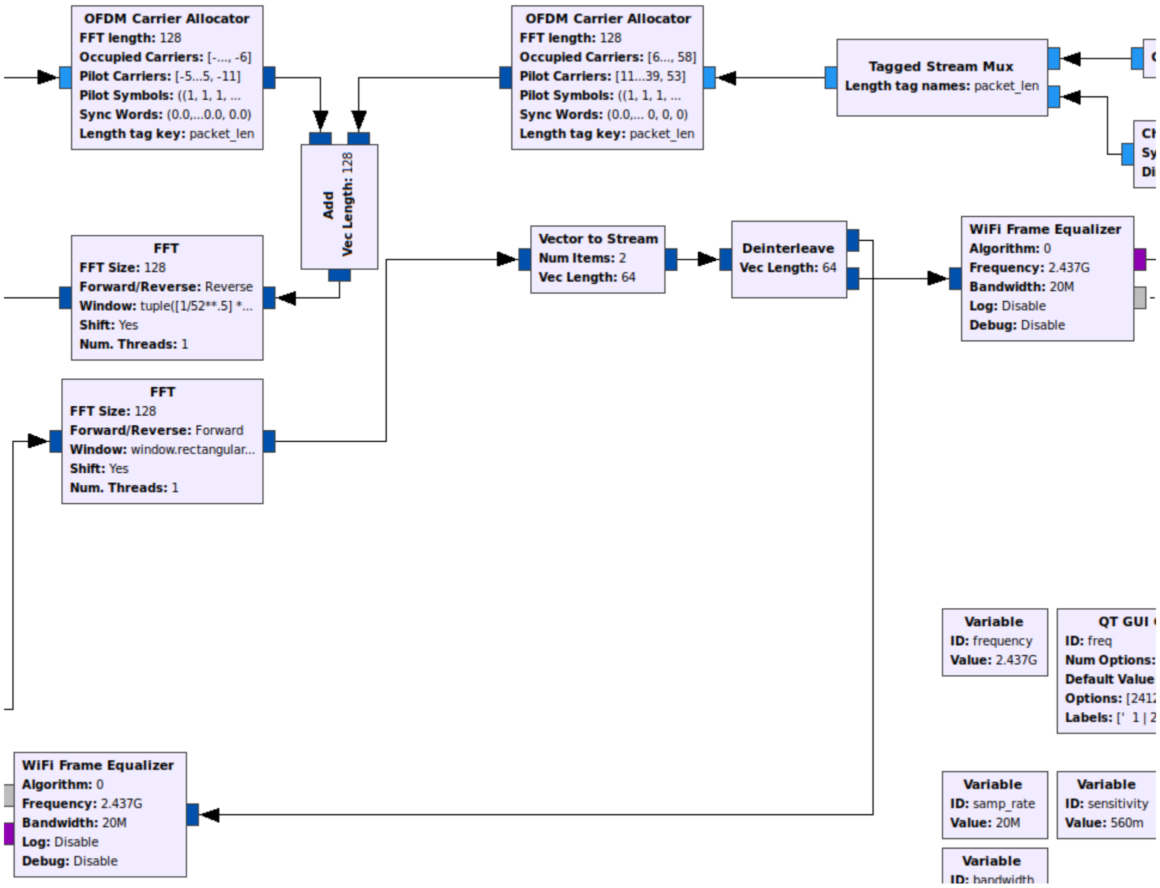
Once the WiFi Frame Equalizer has the 48 carrier symbols, it then applies in reverse the interleave pattern which is a structured way of unscrambling the 48 carrier symbols. Interleaving the symbols helps space out contiguous errors so that error checking and correcting codes can more easily spot and fix errors. This interleaving is performed in the frame_equalizer_impl.cc file (Bloessl, 2017b). Once the WiFi Frame Equalizer has passed on the carrier symbols, the WiFi Decode MAC translates the constellations from the carrier symbols into bits. (The interleaving and scrambling apply also to bits, not just symbols.) WiFi Decode MAC then de-interleaves and unscrambles the bits and drops any invalid WiFi frames and any frames that fail the parity or checksum.

Above the Physical Layer

This concludes the physical layer of the receiver. The resultant bits are sent through the Ethernet encapsulation so that the TAP interface in Linux receives a valid Ethernet packet. At this point the data behaves like normal networking traffic from a VPN or Ethernet adapter, and Wireshark can capture the traffic.

2nd Attempt at Splitting the Channels (OFDMA)

On the latest attempt at implementing OFDMA, the following basic block diagram should implement a successful transmitter. There is not yet a way of testing whether the transmitter works, as the receiver has not been completely re-implemented. However, it should provide a reasonable template, and with modification to the C++ code for certain blocks and modification to the auto-correlation detection blocks, it should be possible to implement the OFDMA receiver. The most interesting part of the modified flowgraph is shown in Figure 9. Figure 9 shows the logic that combines and separates the OFDMA signals. As this new flowgraph was built off copied parts of the nonhierarchical flowgraph, the craziness has doubled to the point where it would be impractical to show the entire flowgraph legibly, but this paper shows it anyway for completeness in Figure 10.



OFDM Carrier Allocator
FFT length: 128
Occupied Carriers: [-..., -6]
Pilot Carriers: [-5...5, -11]
Pilot Symbols: ((1, 1, 1, ...
Sync Words: (0.0,...0.0, 0.0)
Length tag key: packet_len

OFDM Carrier Allocator
FFT length: 128
Occupied Carriers: [6..., 58]
Pilot Carriers: [11...39, 53]
Pilot Symbols: ((1, 1, 1, ...
Sync Words: (0.0,... 0, 0, 0)
Length tag key: packet_len

Tagged Stream Mux
Length tag names: packet_len

Add
Vec Length: 128

FFT
FFT Size: 128
Forward/Reverse: Reverse
Window: tuple([1/52**.5] *...
Shift: Yes
Num. Threads: 1

FFT
FFT Size: 128
Forward/Reverse: Forward
Window: window.rectangular...
Shift: Yes
Num. Threads: 1

Vector to Stream
Num Items: 2
Vec Length: 64

Deinterleave
Vec Length: 64

WiFi Frame Equalizer
Algorithm: 0
Frequency: 2.437G
Bandwidth: 20M
Log: Disable
Debug: Disable

Variable
ID: frequency
Value: 2.437G

QT GUI
ID: freq
Num Options:
Default Value
Options: [241
Labels: [' 1 | 2

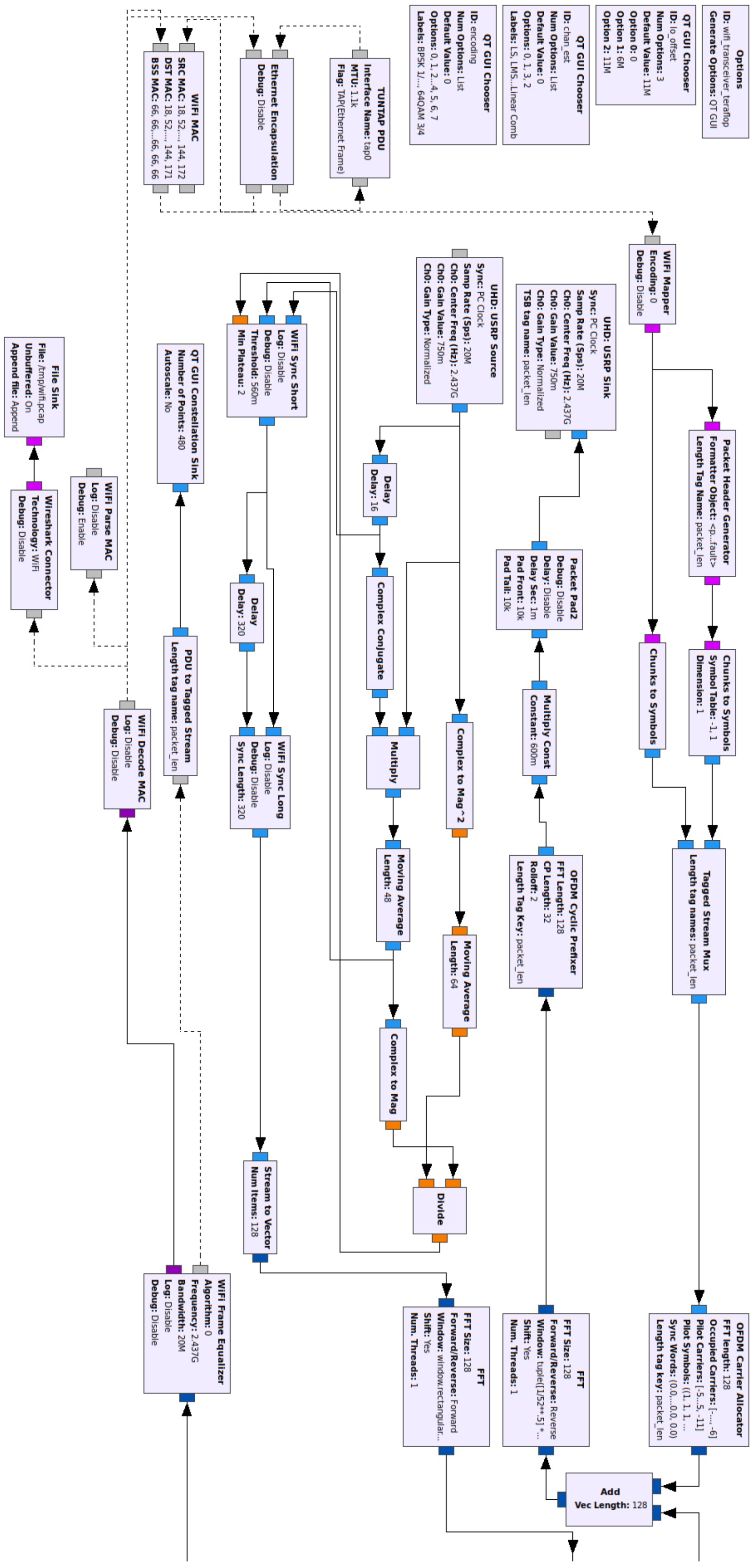
Variable
ID: samp_rate
Value: 20M

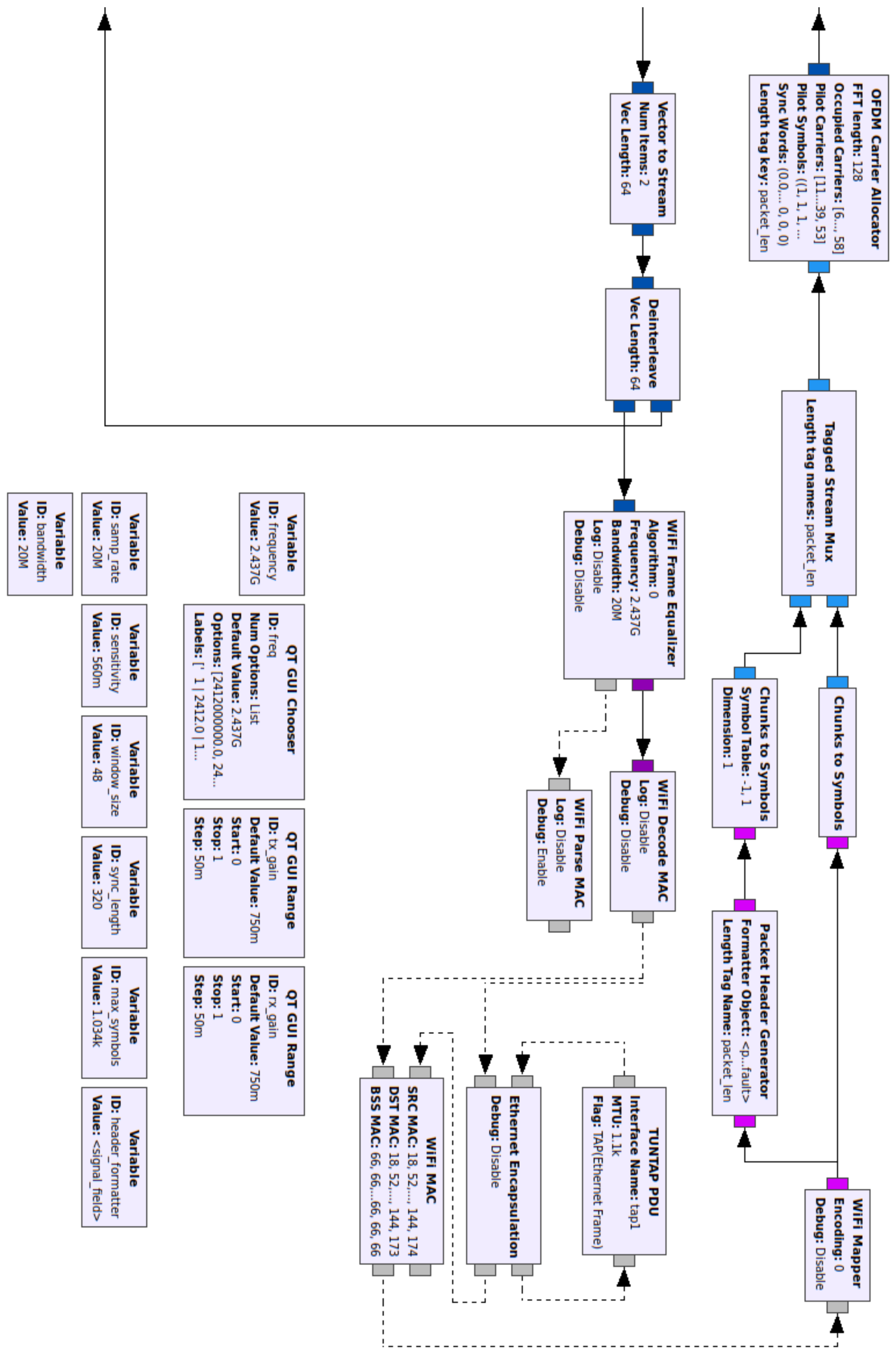
Variable
ID: sensitivity
Value: 560m

Variable
ID: bandwidth

WiFi Frame Equalizer
Algorithm: 0
Frequency: 2.437G
Bandwidth: 20M
Log: Disable
Debug: Disable

Figure 9. Logic for the transceiver to combine and separate the halves of the channel. Transmitter data is added together, and receiver data is de-interleaved.


Options
ID: wifi_transceiver_teraflop
Generate Options: QT GUI
QT GUI Chooser
ID: lo_offset
Num Options: 3
Default Value: 11M
Option 0: 0
Option 1: 6M
Option 2: 11M
QT GUI Chooser
ID: chan_est
Num Options: List
Default Value: 0
Options: 0, 1, 3, 2
Labels: LS, LMS,...Linear Comb
QT GUI Chooser
ID: encoding
Num Options: List
Default Value: 0
Options: 0, 1, 2,...4, 5, 6, 7
Labels: BPSK 1/..., 64QAM 3/4
TUNTAP PDU
Interface Name: tap0
MTU: 1.1k
Flag: TAP(Ethernet Frame)
Ethernet Encapsulation
Debug: Disable
WiFi MAC
SRC MAC: 18, 52,..., 144, 172
DST MAC: 18, 52,..., 144, 171
BSS MAC: 66, 66,...66, 66, 66
WiFi Mapper
Encoding: 0
Debug: Disable
UHD: USRP Sink
Sync: PC Clock
Samp Rate (Sps): 20M
Ch0: Center Freq (Hz): 2.437G
Ch0: Gain Value: 750m
Ch0: Gain Type: Normalized
TSB tag name: packet_len
UHD: USRP Source
Sync: PC Clock
Samp Rate (Sps): 20M
Ch0: Center Freq (Hz): 2.437G
Ch0: Gain Value: 750m
Ch0: Gain Type: Normalized
WiFi Sync Short
Log: Disable
Debug: Disable
Threshold: 560m
Min Plateau: 2
QT GUI Constellation Sink
Number of Points: 480
Autoscale: No
File Sink
File: /tmp/wifi.pcap
Unbuffered: On
Append file: Append
WiFi Parse MAC
Log: Disable
Debug: Enable
Wireshark Connector
Technology: WiFi
Debug: Disable
Packet Header Generator
Formatter Object: <p...fault>
Length Tag Name: packet_len
Delay
Delay: 16
Packet Pad2
Debug: Disable
Delay: Disable
Delay Sec: 1m
Pad Front: 10k
Pad Tail: 10k
Delay
Delay: 320
Complex Conjugate
Chunks to Symbols
Symbol Table: -1, 1
Dimension: 1
Chunks to Symbols
Multiply Const
Constant: 600m
Complex to Mag ^2
Multiply
WiFi Sync Long
Log: Disable
Debug: Disable
Sync Length: 320
PDU to Tagged Stream
Length tag name: packet_len
WiFi Decode MAC
Log: Disable
Debug: Disable
Tagged Stream Mux
Length tag names: packet_len
OFDM Cyclic Prefixer
FFT Length: 128
CP Length: 32
Rolloff: 2
Length Tag Key: packet_len
Moving Average
Length: 48
Moving Average
Length: 64
Complex to Mag
Divide
Stream to Vector
Num Items: 128
OFDM Carrier Allocator
FFT length: 128
Occupied Carriers: [-...,-6]]
Pilot Carriers: [-5,-5,-11]]
Pilot Symbols: ((1,1,1,...
Sync Words: ((0,0,...0,0,0))
Length tag key: packet_len
FFT
FFT Size: 128
Forward/Reverse: Reverse
Window: tuple([1/52**.5] *...
Shift: Yes
Num. Threads: 1
FFT
FFT Size: 128
Forward/Reverse: Forward
Window: window.rectangular...
Shift: Yes
Num. Threads: 1
WiFi Frame Equalizer
Algorithm: 0
Frequency: 2.437G
Bandwidth: 20M
Log: Disable
Debug: Disable
Add
Vec Length: 128


OFDM Carrier Allocator
FFT length: 128
Occupied Carriers: [6..., 58]
Pilot Carriers: [11...39, 53]
Pilot Symbols: ((1, 1, 1, ...
Sync Words: ((0.0,... 0, 0, 0)
Length tag key: packet_len

Vector to Stream
Num Items: 2
Vec Length: 64

Deinterleave
Vec Length: 64

Tagged Stream Mux
Length tag names: packet_len

WiFi Frame Equalizer
Algorithm: 0
Frequency: 2.437G
Bandwidth: 20M
Log: Disable
Debug: Disable

Chunks to Symbols
Symbol Table: -1, 1
Dimension: 1

Chunks to Symbols

WiFi Decode MAC
Log: Disable
Debug: Disable

WiFi Parse MAC
Log: Disable
Debug: Enable

Packet Header Generator
Formatter Object: <p...fault>
Length Tag Name: packet_len

TUNTAP PDU
Interface Name: tap1
MTU: 1.1k
Flag: TAP(Ethernet Frame)

Ethernet Encapsulation
Debug: Disable

WiFi MAC
SRC MAC: 18, 52,..., 144, 174
DST MAC: 18, 52,..., 144, 173
BSS MAC: 66, 66,...66, 66, 66

WiFi Mapper
Encoding: 0
Debug: Disable

Variable
ID: frequency
Value: 2.437G

QT GUI Chooser
ID: freq
Num Options: List
Default Value: 2.437G
Options: [2412000000.0, 24...
Labels: [' 1 | 2412.0 | 1...

QT GUI Range
ID: tx_gain
Default Value: 750m
Start: 0
Stop: 1
Step: 50m

QT GUI Range
ID: rx_gain
Default Value: 750m
Start: 0
Stop: 1
Step: 50m

Variable
ID: samp_rate
Value: 20M

Variable
ID: sensitivity
Value: 560m

Variable
ID: window_size
Value: 48

Variable
ID: sync_length
Value: 320

Variable
ID: max_symbols
Value: 1.034k

Variable
ID: header_formatter
Value: <signal_field>

Variable
ID: bandwidth
Value: 20M

Figure 10. Full flowgraph of transceiver implementation (on 2 pages)

Most of the blocks from the TAP down to the OFDM Carrier Allocator have been duplicated. This duplication occurs after and before the signals are combined. The changes made there include changing the MAC addresses for the new TAP interfaces, changing the sample rate, and changing the size of the FFT. Because the size of the FFT was changed, the sync words had to be changed to fit 128 values. Each side of the flowgraph was padded with zeros so that when they were added together, the sync words would not overlap each other. The allocated carriers were separated to each side of the band. To perform this shift of carriers, 32 was simply added or subtracted from each carrier number. The reason the sample rate could simply be increased, and the band could be split into more carriers is because the new specifications specify that carrier spacing can be closer together in the IEEE 802.11ax standard.

Further Coding Necessary (OFDMA)

This paper shows an attempt at implementing Orthogonal Frequency Division Multiple Access (OFDMA). Implementing OFDMA will require modifying how the underlying radio wave is interpreted. In order to fully implement OFDMA with dynamic carrier allocation, the CSMA portion would need to be implemented first, so carriers (specific frequencies that the radio will transmit on) could be properly assigned to different radios. However, this paper will not cover that because it would require too much debug time, and this paper looks to get a simple proof of concept of OFDMA working.

There is some further coding that will be necessary in order to tune the transmitter and receiver for the OFDMA. Most of the coding will need to be done for the receiver in

C++, but there may need to be some slight change in the transmitter code or simply the values in the transmitter blocks to match the receiver if necessary.

Receiver

Tweaking the receiver code will be much harder than changing the transmitter code was. Since the sample rate was changed to 20MHz from 10MHz, the auto-correlation blocks may need different values to be able to detect the OFDM cyclic prefix. Because of these changes, a future researcher may need to make the following changes in the WiFi Sync Short block. If the future researcher changes the delay from 16 samples in the auto-correlation blocks, he might need to change d_freq_offset divisor from 16 to something else. The future researcher may need to double MAX_SAMPLES since there are twice as many samples for the same amount of time. The future researcher may also need to make the following changes in the WiFi Sync Long block. The future researcher may need to modify the length values for vectors and symbols similar to how MAX_SAMPLES may need to be doubled. This could be very tricky because there are numbers that are *n*-1 hard-coded into the code, not just *n*, so the future researcher will have to understand all the algorithms in the WiFi Sync Long block, and what they do, and if they are related to the task at hand. The future researcher may need to modify the sync words that the receiver is looking for since the previous sync words are only taking up an equivalent of half of the band but are then added next to each other within the band. The future researcher may need to figure out what the FIR filter stored in LONG at the end of WiFi Sync Long is doing and generate a new FIR filter for the 128 length FFT case. The future researcher might even be able to copy the existing FIR filter, and simply make it twice as long. My best guess is that the FIR filter improves the flatness of the frequency response curve for the given input samples.

If issues with routing arise, the future researcher might try duplicating the WiFi MAC, but not the Ethernet encapsulation and TAP. That way the future researcher will only have one TAP interface, and MAC filtering will be handled by the WiFi MAC block. That way packets intended for one machine will get sent to the correct half of the band and to the correct client. Once the future researcher does that, the future researcher should be able to implement a single user use case where the packet is being sent across both halves of the band. This might require some modification of how the PDU's come in and go out of the TAP interface.

Transmitter

The only value that might need tweaking on the transmitter is the length of the cyclic prefix for each OFDM frame. The current value should remain within standard since the number of samples has been doubled to 32 and the sample rate has been doubled so the cyclic prefix lasts the same amount of time: 1.6µs.

Further Coding Necessary (CSMA)

This paper provides pseudocode for how to implement Carrier Sensing Multiple Access (CSMA) in GNU Radio. The software defined radio hardware used has an FPGA (Field Programmable Gate Array) in it. Ideally, this paper would implement all the CSMA functionality in FPGA but cannot due to the hardware limitations of the Ettus N210. The FPGA is composed of a finite number of logic gates that can be programmed to certain configurations. However, the FPGA in the software defined radios available through the engineering department (Ettus N210) do not have enough programmable gates in order to implement such a complex configuration as is required by the CSMA functionality. Bloessl, Dressler, Puschmann, and Sommer (2014) write that the number of logic gates in the Ettus USRP N210 is not enough:

> This would, however, require considerable functionality on the FPGA, including frame detection, synchronization, demodulation, and a Viterbi decoder. Having these physical layer algorithms in hardware would contradict the software only approach and exceed the resources of the used Ettus N210 (p. 60).

Basically, this means that the available hardware from the engineering department is not sufficiently capable to be able to implement the CSMA in FPGA.

Consequences of Software Implementation

Instead of implementing CSMA in FPGA hardware, this paper includes pseudocode for implementing the functionality in software running in the computer, namely in GNU Radio. Implementing CSMA in software will have the following consequences. The results will not be standard compliant, and the transmission speeds will be slower. The programming may be easier, and the timing may cause unknown issues. For the signal samples to be processed in software, the samples must pass through the software defined radio, get encoded into gigabit ethernet packets, and get decoded back into samples at the computer all before even reaching the software. This latency in the execution of software will be too large to satisfy the timing requirements for the

CSMA in the 802.11 standard. The failure to meet timing requirements causes the project to fall out of compliance with the full standard.

The speed of transmission is also related to the latency of the software. More delay between each packet sent plus the time needed to run code instead of hardware implementation will slow down how fast each packet is able to be sent. There is also the potential that this slowdown will cause other unforeseen issues such as packet loss. Hopefully, however, implementing the CSMA algorithms will be easier in software than it would otherwise be in the FPGA.

Design Possibilities

The point of implementing CSMA will be to get multiple computers to talk to each other. First, the idea is to connect two computers and have them transmit only when the channel is idle. This will require adding some connecting logic between the transmitter and receiver portions of the flow graph in GNU Radio. Then, a third software defined radio and computer connect to the other computers as well. The fact that this third computer is able to connect to the others would fully demonstrate that the CSMA is working properly. It may work such that one of the software defined radios will act as a router, and the others will act as clients.

In order to Implement CSMA, according to Bloessl's (Personal communication, February 21, 2018) suggestion for a system that does not meet IEEE timing requirements, "You need some additional logic that connects the RX and TX chain. Either introduce a new block or extend, for example, decode MAC." Connecting the RX and TX chains might be the simplest option. It looks like someone is transmitting when we get samples from WiFi Sync Short, so maybe when we are not getting anything from there, we are good to send traffic. Since there are so many samples coming out of the WiFi Sync Short, it may be good to deal with them quickly, or simply check to see how many samples we get. We need some kind of trigger block to block the sender data from sending while we are getting data from there. Or, like Dr. Bae suggested, detect the power of the signal directly from incoming samples. This would cause way more computations though and could impact performance.

Proposed Design and Pseudo Code

To implement CSMA, create a GNU Radio block. This block would take in input from WiFi Sync Short, the transmitting IFFT, and would output to the OFDM Cyclic Prefixer. When there are OFDM vectors, wait for the samples from WiFi Sync Short to end, then sleep according to exponential back off and test again. It will only copy the vector once it receives the go ahead from this CSMA logic. When there are no more OFDM vectors, it will wait until there are more vectors. Once it gets these new vectors, the process repeats. This paper suggests this implementation with states and case statement: busy or back off state, free or idle state, and transmit or copy state. The pseudo code is shown.

```
private:
enum { BUSY, FREE, TRANSMIT } state_type;

csma (
      n_in_receiver_samples, in_receiver_samples,
      n_in_transmitter_samples, in_transmitter_samples,
      n_out_transmitter_samples, out_transmitter_samples
)
{
      // Constants to be set in block properties
      int COLLISION_LIMIT = 10;
      float TIME_UNIT = 1.6; //us
      int IDLE_WAIT = 5; // TIME_UNITs
```

```
        // States to be saved across executions of function.
        static state_type state;
        static int n_collisions = 0;
        static int wait_timer = 0;

        if(n_in_receiver_samples > 0)
        {
                if (state == TRANSMIT)
                {
                        log("Collision detected!");
                        n_collisions++;
                        if(n_collisions > COLLISION_LIMIT)
                        {
                                n_collisions = 0;
                        }
                }
                state = BUSY;
        }

        switch (state)
        {
                case BUSY:
                {
                        wait_timer = rand_between(0, pow(2, n_collisions)-1);
                        consume(receiver_input, n_in_receiver_samples);
                        //wait and check again next run of loop
                        wait(wait_timer*TIME_UNIT);
                        wait_timer = 0;
                        state = FREE;
                        return 0;
                }
                case FREE:
                {
                        wait_timer = IDLE_WAIT;
                        wait_us(wait_timer*TIME_UNIT);
                        state = TRANSMIT;
                        return 0;
                }
                case TRANSMIT:
                {
                        int n_copy_transmit_samples =
                                length_of_next_wifi_frame(n_in_receiver_samples,
                                                                in_receiver_samples);
                        for (int i=0; i<n_copy_transmit_samples; i++)
                        {
                                output_item[i] = input_item[i];
                        }
                        consume(transmitter_input, n_copy_transmit_samples);
                        return 0;
                }
        }
}
```

Lessons learned

The biggest non-technical lesson learned from this project was how helpful writing documentation is. Throughout the process of testing, failing and learning, notes were kept, and screenshots were taken. This helped pick up unfinished tasks and see progress throughout the project. This documentation also served as a huge help when referencing things for the research project in this honors thesis.

Flowgraph programming can be nice sometimes because it naturally produces a block diagram. However, every developer should understand that the block diagram turns into a very unreadable mess as the complexity of the program increases. The final flowgraph extends beyond a 1080p monitor, so a standard display must scroll just to make all of it visible. On computers that do not process graphics very well, these large flowgraphs (and even small ones) will end up lagging terribly and sometimes even crashing.

A program is almost always more complex than it initially seems. This project was bigger and more in-depth than imagined. There were more things that went wrong than expected, and there was also more to do than expected.

Future work on this project should have some rough prerequisites. Communication Systems (ENGE 341) and Digital Signal Processing (ENGE 312) were extremely helpful courses to even understand what was going on under the hood of the program. Even with that base knowledge, there are still many concepts that must be self-taught in order for a future researcher to be successful. Future work on this project will also require a knowledge of the Linux command line, how to make a block using C++ in GNU Radio, and C++ programming in general.

Conclusion

The outcomes of this research are the aforementioned implementations of portions of the 802.11ax WiFi standard, namely the OFDMA transmitter. Implementing the 802.11ax WiFi standard has not been attempted before in software, most likely because it is impossible to fully comply with the standard based on timing constraints of the CSMA for WiFi. The transmission speed is expected to be slower than an FPGA implementation would elicit since it is being implemented in software. Due to the padding to the WiFi frames in order to detect the frames in software, one packet can easily take more than 200ms to transmit. This paper demonstrated an attempt at implementing OFDMA and provided pseudo code for implementing CSMA.

References

Abdullah, A.N.M., Akbarpour, B., & Tahar, S. (2009). Error analysis and verification of an IEEE 802.11 OFDM modem using theorem proving. *Electronic Notes in Theoretical Computer Science*, *242*(2), 3-30. doi:10.1016/j.entcs.2009.06.020

Adlakha, S., Kulkarni, G., & Srivastava, M.B. (2005). Subcarrier allocation and bit loading algorithms for OFDMA-based wireless networks. *IEEE Transactions on Mobile Computing*, *4*, 652-662. doi:10.1109/TMC.2005.90

Bellalta, B. (2016). IEEE 802.11ax: High-efficiency WLANS. *IEEE Wireless Communications*, *23*, 38-46. doi: 10.1109/MWC.2016.7422404

Bloessl, B. (2017a) gr-foo [Computer software]. Retrieved from https://github.com/bastibl/gr-foo

Bloessl, B. (2017b) gr-ieee802-11 [Computer software]. Retrieved from https://github.com/bastibl/gr-ieee802-11

Bloessl, B., Dressler, F., Leitner, C., & Sommer, C. (2013). A GNU radio-based IEEE 802.15.4 testbed. 37-40. Retrieved from http://www.ccs-labs.org/bib/bloessl2013gnu/bloessl2013gnu.pdf

Bloessl, B., Dressler, F., Puschmann, A., & Sommer, C. (2014). Timings matter: Standard compliant IEEE 802.11 channel access for a fully software-based SDR architecture. *Mobile Computing and Communications Review*, 57-63. doi: 10.1145/2643230.2643240

Bloessl, B., Dressler, F., Segata, M., & Sommer, C. (2013). An IEEE 802.11a/g/p OFDM receiver for GNU radio. *Proceedings of the Second Workshop on Software Radio Implementation Forum*, 9-16. doi:10.1145/2491246.2491248

El-Ghoroury, H.S., McNeill, D., & Sourour, E. (2004). Frequency offset estimation and correction in the IEEE 802.11a WLAN. *IEEE 60th Vehicular Technology Conference, 2004. VTC2004-Fall. 2004*, *7*, 4923-4927. doi:10.1109/VETECF.2004.1405033

Fuxjäger, P., & Ricciato, F. (2010). IEEE 802.11p transmission using GNURadio. *6th Karlsruhe Workshop on Software Radios (WSR)*, 83-86.

Gadhiok, M. (2008). *Symbol Timing Synchronization for OFDM-based WLAN Systems* (Master's thesis). Retrieved from ProQuest Dissertations and Theses database. (UMI No. 1455238)

Introduction to 802.11ax high-efficiency wireless. (2017). Retrieved from http://www.ni.com/white-paper/53150/en/

Kuo, C.J., Morelli, M., & Pun, M. (2007). Synchronization techniques for orthogonal frequency division multiple access (OFDMA): A tutorial review. *Proceedings of the IEEE*, 95, 1394-1427. doi:10.1109/JPROC.2007.897979

Li, B., Qu, Q., Yang, M., & Yan, Z. (2015). An OFDMA based concurrent multiuser MAC for upcoming IEEE 802.11ax. *2015 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 136-141. doi:10.1109/WCNCW.2015.7122543

Linux Man Page. arp(8). Retrieved from https://linux.die.net/man/8/arp

Linux Man Page. ifconfig(8). Retrieved from https://linux.die.net/man/8/ifconfig

Linux Man Page. iptables(8). Retrieved from https://linux.die.net/man/8/iptables

Linux Man Page. python(1). Retrieved from https://linux.die.net/man/1/python

Linux Man Page. sysctl(8). Retrieved from https://linux.die.net/man/8/sysctl

Liu, C.-H. (2003). On the design of OFDM signal detection algorithms for hardware implementation. *Global Telecommunications Conference, 2003*, 596-599. doi:10.1109/GLOCOM.2003.1258308

Mrutu, S.I., Sam, A. & Mvungi, N.H. (2014). Trellis analysis of transmission burst errors in Viterbi decoding. *International Journal of Computer Science and Information Security*, *12*, 46-53.

Müller, M. (2015). Splitting a vector at the output of the FFT. Retrieved from https://lists.gnu.org/archive/html/discuss-gnuradio/2015-03/msg00375.html

Networking Kernel Parameters. (2018). Retrieved from https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

Pavel-Odintsov. (2012). Pavel-odintsov/drop_watch. Retrieved from https://github.com/pavel-odintsov/drop_watch/wiki/Ubuntu-14.04-LTS-kernel-with-drop_monitor-support

Time synchronization for OFDM applications. (n.d.). Retrieved from https://www.nutaq.com/blog/time-synchronization-ofdm-applications

Wang, D., & Zhang, J. (2007). Timing Synchronization for MIMO-OFDM WLAN Systems. *2007 IEEE Wireless Communications and Networking Conference*. doi:10.1109/wcnc.2007.223

Wime Project. Wireless Measurement and Experimentation. Retrieved from https://www.wime-project.net/

Wireshark (Version 2) [Computer Software]. Wireshark · Go Deep. Retrieved from https://www.wireshark.org