CUFE
CCE-E
Credit Hours System

Spring / 2017
Senior-2 Level
Graduation Project-2
CCEN481

Graduation Project-2

"V2V Communication"

PHY Layer Implementation

Final Report

Submitted by:

Habiba Tarek Al-Toudy

Dina Mohamed Magdy Zakaria Eissa

Salma Khaled Ismail Kamel

Samer Ahmed

Mohamed Hussein El-Naggar

Sherine Othman Salem

Zeinab Ahmed

Supervised by:

Dr. Hassan Moustafa

Dr. Yasmine Fahmy

Acknowledgement

Listed below are the names of the people who provided us with significant help in developing our graduation project in addition to our sponsor iGP-ASU and Consultix corporate. To all we extend our sincere thanks.

Dr. Hassan Mostafa Hassan

Dr. Yasmeen Fahmy

Dr. Maged Ghoneima

Eng. Ayman Hendawy

Eng. Ahmed el Menshawy

And a special thanks to Dr. Hassan Mostafa, it has been a great pleasure and honor being our supervisor. You were continuously encouraging us, even before we decided to work on this project.

Finally, though only our names appear on the cover of this thesis, but many have - knowingly or unknowingly – contributed to its production, and for that we are extremely thankful,

Dina, Habiba, Salma, Sherine, Zeinab, Samer, Mohamed

Abstract

Most accidents occur because the driver can only see, with the sensors and the current electronic driver aids, as far as the vehicles directly in front of him/her, behind him/her, or on either side. A competent driver might notice more than one car ahead or behind, notice the signal lights and act preemptively to prevent any sudden actions or accidents. However, sometimes this isn't enough. If any sudden action was taken faster than the driver's reaction such as a vehicle coming in a very high speed next to him/her or realizing there's a huge obstacle when the car is too near to take the needed precautions, this will lead to dangerous consequences. As a result, there has to be another solution that will car itself notice the sudden changes to take precautions if the driver couldn't. Also there has to be a solution to make the able to see more than 2 vehicles ahead or behind to alert the driver of the changes that happen a little further than his/her sight so that the driver can act smoothly and preemptively. Car accidents have risen to 14500 accident in 2015. A total of 63.3 percent of car accidents were caused by humans. A total of 6203 were killed and 19325 were injured due to such accidents in 2015.

One of the technological advances that could solve this problem is vehicle to vehicle communication. This report will include more information about V2V communication, its benefits and its market nowadays. Next, an overview of the IEEE standard that is used to implement the PHY layer of the V2V communication system is explained. After that, the project design is discussed along with the tools used as well as the importance of each tool in our project. Then, the actual implementation of our project along with the testing methods and results are furtherly explained. Finally, the lessons learned while working on our project as well as the next phases are discussed.

Table of *Contents*

List of Figures

List of Tables

List of Equations

List of Abbreviations

ADC	Analog to digital converter
AGC	Automatic gain control
BPSK	Binary phase-shift keying
CCS	Code composer studio
DAC	Digital to analog converter
DDC	Digital down converter
DSP kit	Digital signal processing kit
DUC	Digital up converter
FDM	Frequency-division multiplexing
FFT	Fast Fourier transform
FIR filter	Finite impulse response
FPGA	Field-programmable gate array
GI	Guard interval
ICI	Inter carrier interference
IEEE 802.11p	IEEE standard for wirless communications
IEEE	Institute of Electrical and Electronics Engineers
IFFT	Inverse furrier transform
ISI	Inter symbol interference
MIB	Management Information Base
NBPSC	Number of bits in each OFDM subcarrier
NCBPS	Number of coded bits per OFDM symbol
NDBPS	Number of data bits per OFDM symbol

OFDM	Orthogonal frequency-division multiplexing
PLCP sub-layer	Physical Layer Convergence Protocol
PLME	Physical layer management entity
PMD	Physical medium dependent
PPDU	PLCP protocol data unit
PSDU	PLCP Service Data Unit
QAM	Quadrature amplitude modulation
QPSK	Quadrature phase shift keying
SoM	System on module
STA	Spectral Temporal Averaging
UPP	Universal parallel port
USRP	Universal Software Radio Peripheral
V2I	Vehicles to infrastructure
V2V	Vehicle to vehicle communication.
V2X	V2I and V2V

Chapter 1

Introduction

1.1 About V2X

Vehicle to vehicle communication is, as its name describes, a way for vehicles to send and receive signals to each other to explain their location, speed and direction. If there was a car that decided to change lanes and the driver didn't pay attention to the other cars who want to do the same, the car that fall behind in line with 3-4 cars between will send signals to this vehicle to inform it to wait until it passes to prevent future possible accidents. That way, the car can know what other out-of-sight cars, are doing or about to do.

In addition to that, the communication will not be between vehicles only, but between vehicles and infrastructure as well (V2I), reducing any human errors that lead to accidents. V2V and V2I have become one name, V2X.

V2X communications are being standardized in various countries and are anticipated to be an important technology for achieving autonomous driving. Development of this technology by automotive manufacturers, chip manufacturers, and technology and solution providers is accelerating.

1.2 V2X market

V2X communication's market is growing every year due to the enhancement of technology use in vehicles. A lot of investment is done in this field nowadays. Middle Eastern countries are considered a great potential for this technology due to the increase in population as well as the focus of many automobile companies on regions such as the Middle East and Africa. The development of this technology by automotive manufacturers, chip manufacturers as well as technology and solution providers is accelerating.

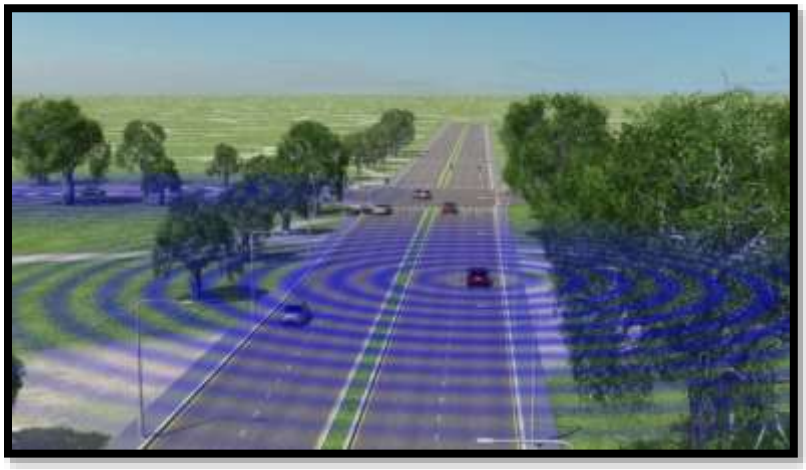

Figure 1-1 the range of signals sent by the car

Figure 1-2 the future of V2X systems

1.3 V2X competition landscape

There are many companies interested in this technology such as BMW, Audi, Daimler, Volvo, and Ford- Applink. Among the solution providers Etrans Systems, Qualcomm Technologies Inc., Cisco Systems Inc., Delphi Automotive PLC, Autotalks Ltd., Denso, Arada Systems, Kapsch Group and Savari Inc., are included in the vehicle to vehicle communication market.

1.4 Standardized V2X protocols

Since, V2X requires devices and vehicles of different manufacturers communicate with each other, there has to be a standard that all companies and manufacturers will follow. That's why IEEE developed the 802.11p standard which explains the physical and mac layers of vehicular transceivers. That way, any other European or American standards developed, will have to be based on the lower-level IEEE 802.11p standard, to ensure the compatibility of different devices communicating with each other.

1.5 Project description

Our project is to build a prototype of the V2V transceiver on the PHY layer to be used as a testbed of the actual V2V transceivers. That way we can test any other device by sending data to it or receiving data from it to make sure it's working properly and to measure how far the data can travel.

Chapter 2

OFDM PHY Layer Specification

This chapter includes the basic information that is needed to be known to implement OFDM PHY layer. The first part is the standard part which will go through OFDM PHY layer structure, its sublayers and the frame structure that is sent by the transmitter. The standard mainly helps in transmitter implementation, that's why it is needed to study some receiving concepts to implement the receiver which is discussed in the second part of this chapter.

2.1 Standard IEEE-802.11p overview

2.1.1 Introduction

This standard is developed by IEEE (Institute of electrical and electronics engineers) organization to describe telecommunications and information exchange between systems Local and metropolitan area networks— Specific requirements and it's mainly determines Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.

The standard has more than one amendment:

- IEEE Std 802.11k™-2008: Radio Resource Measurement of Wireless LANs
- IEEE Std 802.11r™-2008: Fast Basic Service Set (BSS) Transition (Amendment 2)
- IEEE Std 802.11y™-2008: 3650–3700 MHz Operation in USA (Amendment 3)
- IEEE Std 802.11w™-2009: Protected Management Frames (Amendment 4)
- IEEE Std 802.11n™-2009: Enhancements for Higher Throughput (Amendment 5)
- IEEE Std 802.11p™-2010: Wireless Access in Vehicular Environments
- IEEE Std 802.11z™-2010: Extensions to Direct-Link Setup (DLS) (Amendment 7)
- IEEE Std 802.11v™-2011: IEEE 802.11 Wireless Network Management
- IEEE Std 802.11u™-2011: Interworking with External Networks (Amendment 9)
- IEEE Std 802.11s™-2011: Mesh Networking (Amendment 10)

Our project is following mainly amendment IEEE Std 802.11p™-2010: Wireless Access in Vehicular Environments (Amendment 6). Specifically it is an implementation for **orthogonal frequency division multiplexing (OFDM) PHY specification** part in the standard.

2.1.2 Reasons of using OFDM

OFDM transceiver has a lot of advantages that enhance the communication systems and also solves main problems. There are two main problems that are solved using (OFDM)

First one: Multi-path problem in the channel, There is a lot of interacting objects in the channel that cause the problem of multi-path fading as shown in (Fig.2-1), But (OFDM) or mainly the family of (FDM) solves this problem as it divides the band to sub-bands or sub carriers which mean that the signal will be extended in time domain what leads to minimization of effect of the delay on the incoming signal as its time is much greater than the delay.

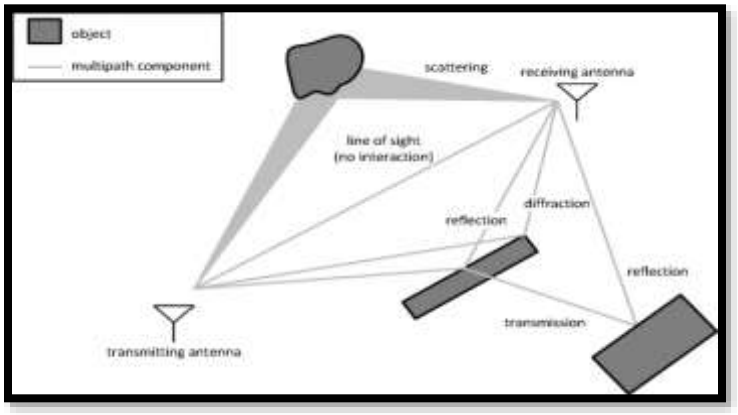

Figure 2-1 MPC effect

Second problem: Is the frequency selective nature of the channel and this was solved in (FDM) as the signal is carried over more than one channel, in only one channel carrier whole signal will be corrupted but in multi-carrier some of the channel will be corrupted not whole the signal and more over using the concepts of coding and frequency diversity will prevent these corrupted subcarriers from corrupting the original signal. Moreover (OFDM) is better than (FDM) as it is more efficient usage of the bandwidth as shown in (Fig.2-2)

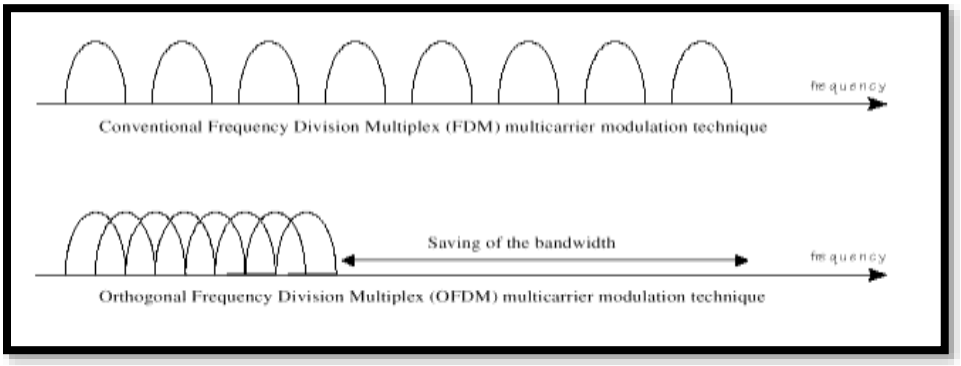

Figure 2-2 Illustration of Frequency selectivity in FDM and OFDM techniques

2.1.3 PHY layer structure in the standard

The physical layer consists of three main sub-layers as shown in the next figure

- PLCP sub-layer
- PMD sub-layer
- PLME sub-layer

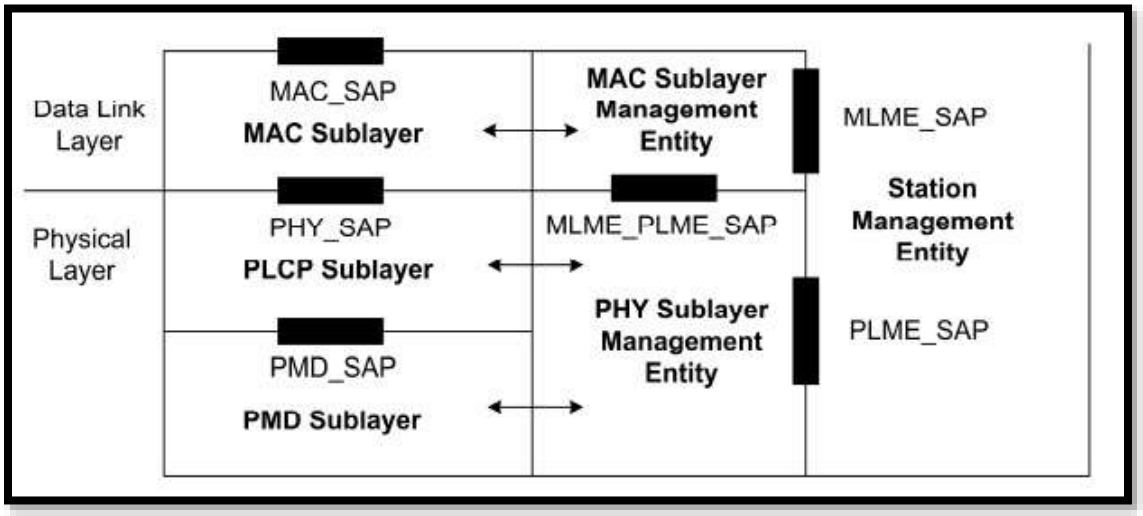

Figure 2-3 Sub-layers in PHY layer

2.1.3.1 PLCP sub-layer (Physical Layer Convergence Protocol)

Provides a convergence procedure in which PSDUs (PLCP Service Data Unit) are converted to and from PPDUs (PLCP protocol data unit).During transmission, the PSDU shall be provided with a PLCP preamble and header to create the PPDU. At the receiver, the PLCP preamble and header are processed to aid in demodulation and delivery of the PSDU.

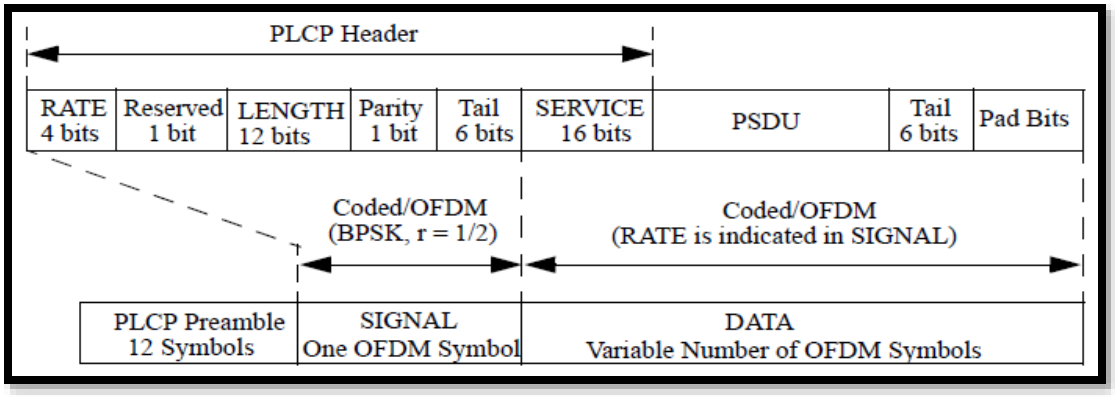

Figure 2-4 PPDU frame format

Overview of the PPDU encoding process

- Produce the PLCP Preamble field, composed of 10 repetitions of a "short training sequence" (used for AGC convergence, diversity selection, timing acquisition, and coarse frequency acquisition in the receiver) and two repetitions of a "long training sequence" (used for channel estimation and fine frequency acquisition in the receiver), preceded by a guard interval (GI)

- Produce the PLCP header field from the RATE, LENGTH, and SERVICE fields of the TXVECTOR by filling the appropriate bit fields. The RATE and LENGTH fields of the PLCP header are encoded by a convolutional code at a rate of R = 1/2, and are subsequently mapped onto a single BPSK encoded OFDM symbol, denoted as the SIGNAL symbol. In order to facilitate a reliable and timely detection of the RATE and LENGTH fields, 6 zero tail bits are inserted into the PLCP header. The encoding of the SIGNAL field into an OFDM symbol follows the same steps for convolutional encoding, interleaving, BPSK modulation, pilot insertion, Fourier transform, and prepending a GI as described subsequently for data transmission with BPSK-OFDM modulated at coding rate 1/2. The contents of the SIGNAL field are not scrambled as shown in the next figure.

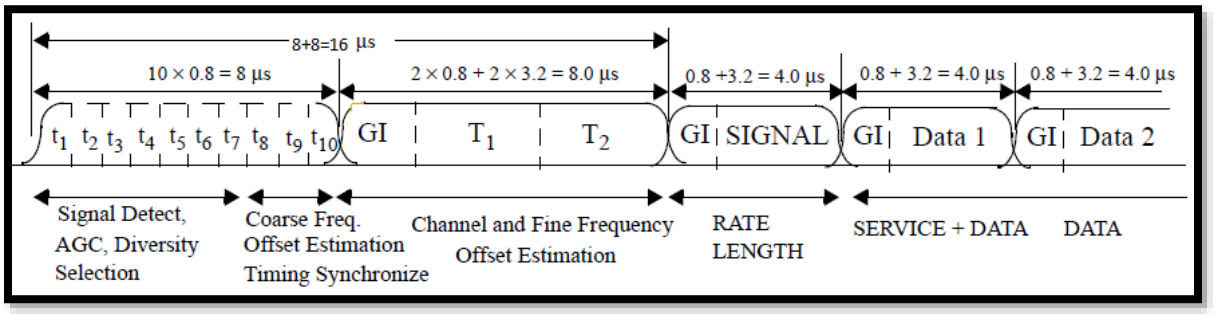

Figure 2-5 PLCP header field

- Calculate from RATE field of the TXVECTOR the number of data bits per OFDM symbol (*NDBPS*), the coding rate (R), the number of bits in each OFDM subcarrier (*NBPSC*), and the number of coded bits per OFDM symbol (*NCBPS*).

- Append the PSDU to the SERVICE field of the TXVECTOR. Extend the resulting bit string with zero bits (at least 6 bits) so that the resulting length is a multiple of *NDBPS*. The resulting bit string constitutes the DATA part of the packet.

- Initiate the scrambler with a pseudorandom nonzero seed, generate a scrambling sequence, and XOR it with the extended string of data bits.

- Replace the six scrambled zero bits following the data with six non-scrambled zero bits. (Those bits return the convolutional encoder to the zero state and are denoted as tail bits).

- Divide the encoded bit string into groups of *NCBPS* bits. Within each group, perform an "interleaving" (reordering) of the bits according to a rule corresponding to the desired RATE.

- Divide the encoded bit string into groups of *NCBPS* bits. Within each group, perform an "interleaving" (reordering) of the bits according to a rule corresponding to the desired RATE.

- Divide the complex number string into groups of 48 complex numbers. Each such group is associated with one OFDM symbol. In each group, the complex numbers are numbered 0 to 47 and mapped hereafter into OFDM subcarriers numbered –26 to –22, –20 to –8, –6 to –1, 1 to 6, 8 to 20, and 22 to 26. The subcarriers –21, –7, 7, and 21 are skipped and, subsequently, used for inserting pilot subcarriers. The 0 subcarrier, associated with center frequency, is omitted and filled with the value 0.

- For each group of subcarriers –26 to 26, convert the subcarriers to time domain using inverse Fourier transform. Prepend to the Fourier-transformed waveform a circular extension of itself thus forming a GI, and truncate the resulting periodic waveform to a single OFDM symbol length by applying time domain windowing.

- Up-convert the resulting "complex baseband" waveform to an RF according to the center frequency of the desired channel and transmit.

Declarations

Coding (rate): Every bit in the data stream is coded (repeated) to allow error correction.

E.g. 1 → 111 (“1/3” coding)
0 → 00 (“1/2” coding)

Interleaving: This is done to achieve frequency diversity to resist the frequecy selectivity nature of the channel.

Preamble: Samples known by the receiver to support in the process of retreiving the original data at it has three functionalities :

- Time sync
- Frequency offset determining
- Channel estimation

Pilot insertion: The preamble is not sufficient for retreiving the original data process as the channel is suffering from variations all the time, to make perfect estimation of the channel some known bits are sent inside the data over some subchannels called pilots.

Service field: Used to send the type of the modulation , number of symbols and the required information to correctly de-modulate the signal

Cyclic extension: It is added to overcome the problems of ISI and ICI, It must be removed from the received frame in order to have the information only.

2.1.3.2 OFDM PMD sublayer

The PMD sublayer accepts the PLCP sub layer primitives and provides the actual means by which data are transmitted or received from the medium.

The PMD sublayer primitives and services for the transmission and reception functions include data stream, timing information, and associated signal parameters being delivered to and from the PLCP sublayer.

THE OFDM sublayer primitives are divided into two different categories:

1- Service primitives that support PLCP peer-to-peer interactions

PMD_DATA.request: This primitive defines the transfer of data from the PLCP sublayer to the PMD entity. When generated, this primitive shall be generated by the PLCP sublayer to request transmission of one OFDM symbol. The data clock for this primitive shall be supplied by the PMD layer based on the OFDM symbol clock.

PMD_DATA.indication: This primitive defines the transfer of data from the PMD entity to the PLCP sublayer. When generated by the PMD, it forwards received data to the PLCP sublayer. The data clock for this primitive shall be supplied by the PMD layer based on the OFDM symbol clock.

2- Service primitives that have local significance and support sublayer-to-sublayer interactions

PMD_TXSTART.request: This primitive is generated by the PHY PLCP sublayer. It initiates PPDU transmission by the PMD layer.
PMD_TXEND.request: This primitive is generated by the PHY PLCP sublayer. It ends PPDU transmission by the PMD layer.
PMD_TXPWRLVL.request: This primitive is generated by the PHY PLCP sublayer to select the power level used by the PHY for transmission.
PMD_RATE.request: This primitive is generated by the PHY PLCP sublayer to select the modulation rate that shall be used by the OFDM PHY for transmission.
PMD_RSSI.indication: This primitive, generated by the PMD sublayer, provides the receive signal strength to the PLCP and MAC entity.
PMD_RCPI.indication: This primitive, generated by the PMD sublayer, provides the RCPI to the PLCP and MAC entity.

2.1.3.3 PLME sub layer

The PLME performs management of the local PHY functions in conjunction with the MLME. It also has the MIB (Management Information Base) attributes which are used in the communication process.

Its parameters are divided into 3 categories:

TXVECTOR parameters

Table 1 Transmitter vector parameters in PLME sublayer

Parameter	Description	Associated primitive	Value
LENGTH	This value is used by the PHY to determine the number of octet transfers that will occur between the MAC and the PHY after receiving a request to start the transmission.	PHY-TXSTART.request (TXVECTOR)	1–4095
DATATRATE	It describes the bit rate at which the PLCP shall transmit the PSDU.	PHY-TXSTART.request (TXVECTOR)	6, 9, 12, 18, 24, 36, 48, and 54 Mb/s for 20 MHz channel spacing (Support of 6, 12, and 24 Mb/s data rates is mandatory.) 3, 4.5, 6, 9, 12, 18, 24, and 27 Mb/s for 10 MHz channel spacing (Support of 3, 6, and 12 Mb/s data rates is mandatory.) 1.5, 2.25, 3, 4.5, 6, 9, 12, and 13.5 Mb/s for 5 MHz channel spacing (Support of 1.5, 3, and 6 Mb/s data rates is mandatory.)
SERVICE	The SERVICE parameter consists of 7 null bits used for the scrambler initialization and 9 null bits reserved for future use.	PHY-TXSTART.request (TXVECTOR)	Scrambler initialization; 7 null bits + 9 reserved null bits

TXPWR_LEVEL	This parameter is used to indicate which of the available TxPowerLevel attributes defined in the MIB shall be used for the current transmission.	PHY-TXSTART.request (TXVECTOR)	1-8
TIME_OF_ DEPARTURE_ REQUESTED	A parameter value of true indicates that the MAC sublayer is requesting that the PLCP entity provides measurement of when the first frame energy is sent by the transmitting port and reporting within the PHY-TXSTART.confirm(TXSTATUS) primitive	PHY-TXSTART.request (TXVECTOR)	False, true. When true, the MAC entity requests that the PHY PLCP entity measures and reports time of departure parameters corresponding to the time when the first frame energy is sent by the transmitting port; when false, the MAC entity requests that the PHY PLCP entity neither measures nor reports time of departure parameters.

RXVECTOR parameters

Table 2 Receiver vector parameters in PLME sublayer

Parameter	Description	Associated Primitive	Value
LENGTH	The MAC and PLCP use this value to determine the number of octet transfers that will occur between the two sublayers during the transfer of the received PSDU.	PHY-RXSTART.indication	1-4095
RSSI	RSSI shall be measured during the reception of the PLCP preamble.	PHY-RXSTART.indication (RXVECTOR)	0–RSSI maximum
DATARATE	DATARATE shall represent the data rate at which the current PPDU was received.	PHY-RXSTART.indication (RXVECTOR)	6, 9, 12, 18, 24, 36, 48, and 54 Mb/s for 20 MHz channel spacing (Support of 6, 12, and 24 Mb/s data rates is mandatory.) 3, 4.5, 6, 9, 12, 18, 24, and 27 Mb/s for 10 MHz channel spacing (Support of 3, 6, and 12 Mb/s data rates

			is mandatory.) 1.5, 2.25, 3, 4.5, 6, 9, 12, and 13.5 Mb/s for 5 MHz channel spacing (Support of 1.5, 3, and 6 Mb/s data rates is mandatory.)
SERVICE		PHY-RXSTART.indication (RXVECTOR)	Null
RCPI	This parameter is a measure by the PHY of the received channel power.	PHY-RXSTART.indication (RXVECTOR)	0-255
RX_START_OF_FRAME_OFFSET	An estimate of the offset from the point in time at which the start of the preamble corresponding to the incoming frame arrived at the receive antenna port to the point in time at which this primitive is issued to the MAC	PHY-RXSTART.indication (RXVECTOR)	0 to 2^32 – 1.

TXSTATUS parameters

Table 3 Transmitter status parameters in PLME sublayer

Parameter	Description	Associated Primitive	Value
TIME_OF_DEPARTURE	The locally measured time when the first frame energy is sent by the transmitting port, in units equal to 1/TIME_OF_DEPARTURE_ClockRate. This parameter is present only if TIME_OF_DEPARTURE_REQUESTED is true in the corresponding request.	PHY-TXSTART.confirm (TXSTATUS)	0 to 2^32 – 1.
TIME_OF_DEPARTURE_ClockRate	The clock rate, in units of MHz, is used to generate the TIME_OF_DEPARTUREvalue. This parameter is present only if TIME_OF_DEPARTURE_REQUESTED is true in the corresponding request.	PHY-TXSTART.confirm (TXSTATUS)	0 to 2^16 – 1
TX_START_OF_FRAME_OFFSET	An estimate of the offset (in 10 ns units) from the point in time at which the start of the preamble corresponding to the frame was transmitted at the transmit antenna port to the point in time at which this primitive is issued to the MAC.	PHY-TXSTART.confirm (TXSTATUS)	0 to 2^32 - 1

2.2 Receiver Overview

This part will go through some concepts in the PHY layer receiver. As noticed, what the standard mostly state about the receiver functionality that it is the opposite of transmitting functions, which is correct. However, it is needed to discover some receiving concepts to be able to implement those opposite operations such as synchronization, decoding …etc. This part will go through these concepts one by one and mention important notes related to the OFDM receiver.

To receive a frame, the following steps take place:

1. Start of frame is detected
2. Transition from short sequence to channel estimation sequence is detected and fine timing is established
3. Coarse and fine frequency offsets are estimated
4. The packet is then compensated with the estimated frequency offset
5. The complex channel response coefficients are estimated for each subcarrier
6. For each symbol inside the OFDM symbol, the symbol is transformed into subcarrier received values, then the phase is estimated using the four pilots and the subcarriers are compensated with this phase. After that, every subcarrier is divided with the complex estimated channel response coefficient.
7. The signal field is then further analyzed to find out the modulating technique, the parsing rate and the number of data octets
8. Finally, the output data is de-interleaved, de-scrambled and de-punctured and decoded to produce the message.

2.2.1 Frame detection

The first step in the receiver is to detect the start of the frame received. Each OFDM frame starts with a short preamble sequence followed by long training sequence then followed by the data as shown in the following figure. To detect the data, operations on each part in the frame should be made. Firstly, with the short preamble sequence which consists of a pattern of 16 samples and repeated 10 times.

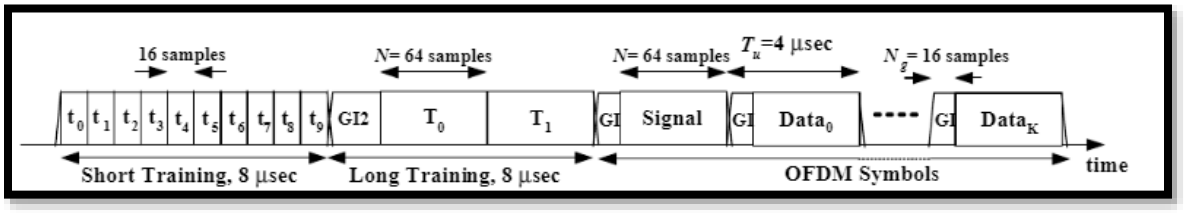

Figure 2-6 Detailed OFDM frame structure

The frame detection algorithm is based on autocorrelation of the short training sequence. Then in order the values to be independent of the absolute values, autocorrelation value will be divided by the average power. Firstly, the autocorrelation absolute value is calculated by the following equation, the value of a[n] results of the incoming sample stream s [n+k] multiplied with the complex conjugate of s lagged by 16. By summing up over an adjustable window we can get the auto correlated values. Secondly, to have independent correlated values of the absolute level of incoming samples, a[n] will be normalized with the average power p[n] and calculate the auto correlation coefficients c[n].

$$a[n] = \sum_{k=0}^{N_{win}-1} s[n+k]\,\overline{s}[n+k+16].$$

Figure 2-7 Autocorrelation calculation algorithm

$$\mathrm{p}[n] = \sum_{k=0}^{N_{\mathrm{win}}-1} \mathrm{s}[n+k]\,\overline{\mathrm{s}}[n+k];$$

$$\mathrm{c}[n] = \frac{|\mathrm{a}[n]|}{\mathrm{p}[n]}.$$

Figure 2-8 Normalized auto correlation values with the average power

Due to cyclic property of the short training sequence, the autocorrelation values will be high at the start of OFDM frame which will detect the start of the frame by comparing values with a threshold. Final thing to do to be sure that the frame start is detected is to leave the first three values (called plateau) more than the threshold value. And if the values after that still greater than the threshold, then the frame start is detected. If they are still less than threshold then the frame is not detected yet. Note that the size of plateau and the value of threshold can vary from a receiver to another. The following figure is an example of autocorrelation distribution in frame detection.

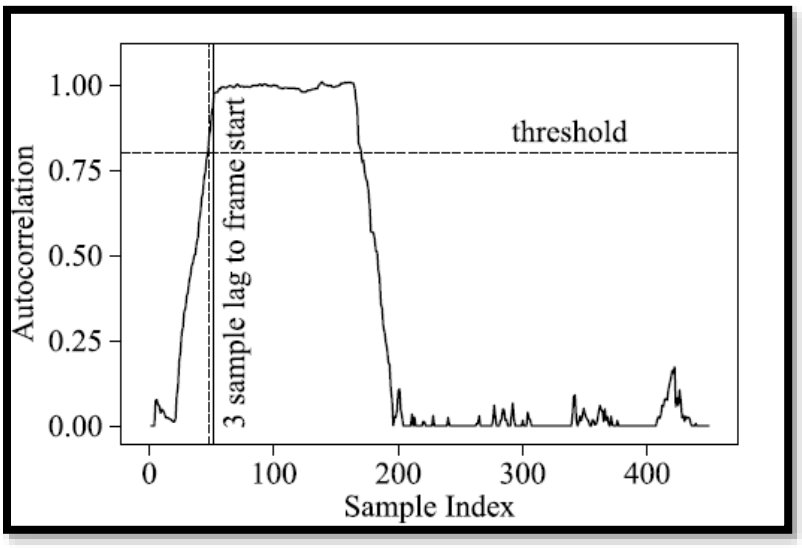

Figure 2-9 Autocorrelation function behavior in the frame detection

2.2.2 Frequency offset correction

Frequency offset correction is required due to the fact of receiving slightly different frequencies. To compensate these differences, there are many algorithms to recover this frequency offset. One of these algorithms is applicable on OFDM receiver which depends on the frame cyclic property. In another words, it's expected in the normal case that a sample s[n] should correspond to the sample s [n+16]. But, due noise and frequency offset occurrence this is no longer the case and s[n]*conj(s [n+16]) is not a real number as in the ideal case.

In order to neglect the noise, the argument of the product that corresponds to 16 times the rotation that is introduced by the frequency offset between samples. Then to estimate the final frequency offset value, averaging is applied (dividing by 16) as shown in df equation in the next figure. Where Nshort is the length of the short training sequence.

Using the argument of sum of the products is more robust against noise, as samples with small magnitudes which are more affected by noise are weighted less.

Finally, the frequency offset is applied to each sample as shown in the next figure.

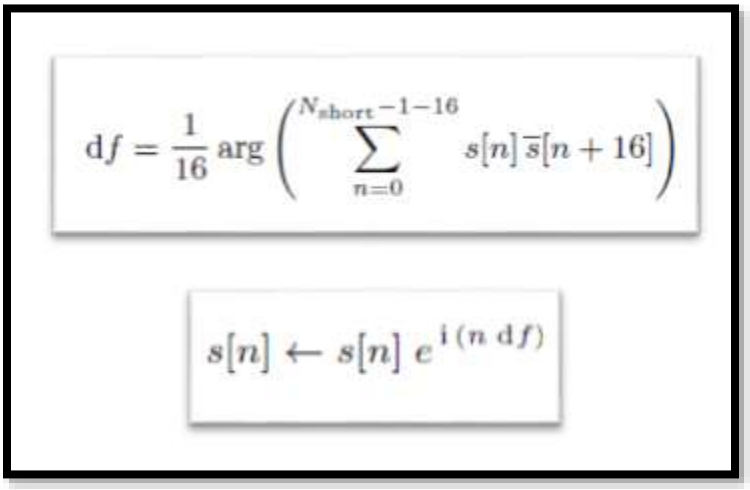

Figure 2-10 Frequency offset calculation equation and updating frame with the new phase

2.2.3 Symbol Alignment

After the frequency offset estimation. Symbol alignment is then performed. The main mission of the symbol alignment is to calculate where the symbol starts, extract the data symbols and send them to FFT algorithm to be transformed from time domain to frequency domain. This task is done with the help of the long training sequence which is composed of 64 samples that repeat 2.5 times. As the alignment have to be very precise, matched filtering is applied first for this operation.

In the next figure, a graph is showing the correlation of the input stream with the known sequence.

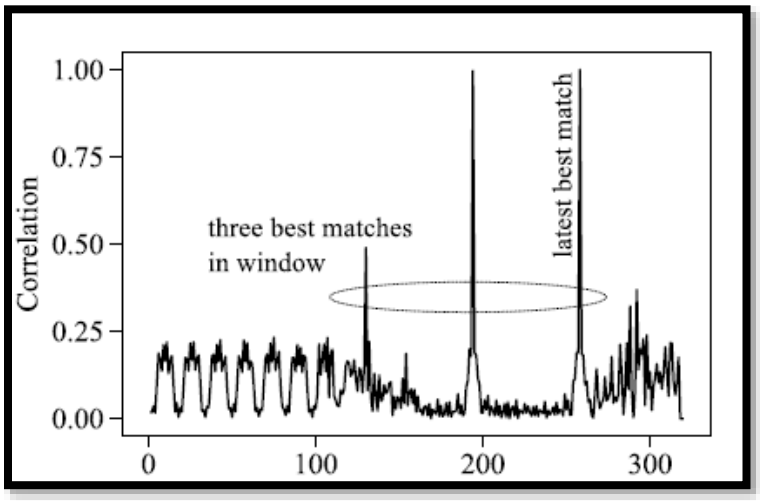

Figure 2-11 Sample Index

The indices of the highest three peaks are calculated using this equation

$$Np = \arg(max3) \sum_{k=0}^{63} s[n+k]LT[k] \qquad where\ n\ \in \{0,1,\dots.,Npreamble\}$$

Where N preamble represents the added length of the short and long preambles, LT is the long preamble pattern that spans 64 samples and arg (max3) return the top three indices maximizing this expression.

The first data symbol starts at the following sample index as the latest peak of the matched filter output is 64 samples before the end of the long training sequence.

$$np = \max(Np) + 64$$

After the relative frame start is detected, the data symbols are then extracted and passed to the FFT algorithm as samples multiples of 64 to perform the FFT with size 64.

In addition to that, knowing the start of the data symbols, the cyclic prefix can be removed by sub setting the data stream and grouping samples that correspond to individual data symbols

$$(s[np+16], \ldots\ldots, s[np+79], s[np+80+16], , , , , ,)$$

Where s [np+16] up to s [np+79] are considered the first symbol and the rest is the second symbol and so on.

2.2.4 Phase offset correction

After the symbol alignment, the output symbols are turned from time domain to frequency domain using the FFT algorithm of size 64. Then, the phase offset correction is done. This phase offset is calculated using the pilot symbols that are inserted inside each OFDM symbol. The phase correction is not only done using the pilots of each symbol independently, but the residual offset is also calculated through the phase offset between the pilot symbols of subsequent symbols. That way, the phase offset can be calculated, compensated and updated frequently to compensate with the fast channel changes.

2.2.5 Channel estimation

After the phase and frequency correction, the data is transformed from complex numbers to octets to be further decoded. This is done using channel coefficients that are extracted using different channel estimation techniques. They perform the same task but with different techniques that give them different efficiencies. These techniques are further discussed in the code description and code design.

2.2.6 Signal field decoding

The first step at the receiver after correct channel equalization and synchronization is to decode the signal field. In each frame, the short and long training sequences are followed by the signal field, which is a BPSK modulated OFDM symbol encoded with a rate of 1/2 that carries information about the length and encoding of the following symbols. In order to do so, we use the deinterleaver function to deinterleave the received signal field bits and a Viterbi decoder to decode the output of the deinterleaver.

If the signal field is decoded successfully, i.e., if the rate field contains a valid value and if the parity bit is correct, the Decode Signal Field return the type of encoding of the data and the number of symbols in each frame and pass it to the next block Frame Decode block.

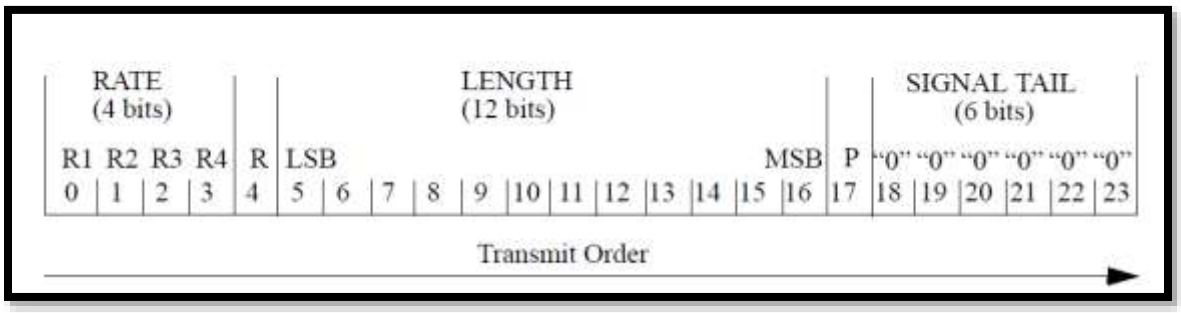

Figure 2-12 Signal field assignment

2.2.7 Frame decoding

The final step in the receiver after correct decoding of signal field is the decoding of the actual payload. It is performed in multiple sub-steps, as follows.

- **Demodulation:** The OFDM Decode block receives vectors of 48 constellation points in the complex plane, corresponding to the 48 data subcarriers per OFDM symbol. According to the used modulation scheme, these constellations are mapped to floating point values, representing the soft-bits of the employed modulation.
- **DE interleaving:** At which the bits of a symbol are permuted. The permutation is the same for all symbols of a frame.
- **Convolutional Decoding and Puncturing:** Depending on the coding rate we use Viterbi decoder for decoding a bit stream that has been encoded using Forward error correction.
- **Descrambling:** The final step in the decoding process is descrambling. In the encoder the initial state of the scrambler is set to a pseudo random value. As the scrambler is implemented with a seven bit feedback shift register, $2^7 = 128$ initial states are possible. The first 7 bit of the payload are part of the service field and always set to zero, in order to allow the receiver to deduce the initial state of the scrambler.

  The mapping from these first bits to the initial state is implemented via a lookup table.

Chapter 3

Project Design

In this chapter, we'll focus on how we designed the project to achieve its functionality and what were the needed components. We will state a quick overview on everything we used to create the picture of the project for the reader to understand the following chapters. Also we'll discuss our testing plan, project phases and cost.

3.1 Implementation Overview

Our implementation started by the need of a development kit to process the TX/RX code and a RF antenna to send and receive data stream. It was found that the model of the PHY layer can be represented by two ways:

3.1.1 Standalone device model

- Steps of model creation:

1- Develop codes of transmitter and receiver
2- Burn the code of transmitter or receiver to the DSP kit for processing data
3- Connect DSP kit with an RF tool to start sending
4- On the other side of reception, there will be the same components receiving data

Notes:

- The standalone model didn't work with USRP since its driver didn't work on DSP kit when we tried to make it. That's why we needed another RF tool that will be made by Consultix corporate (our sponsor).

3.1.2 Step by step model

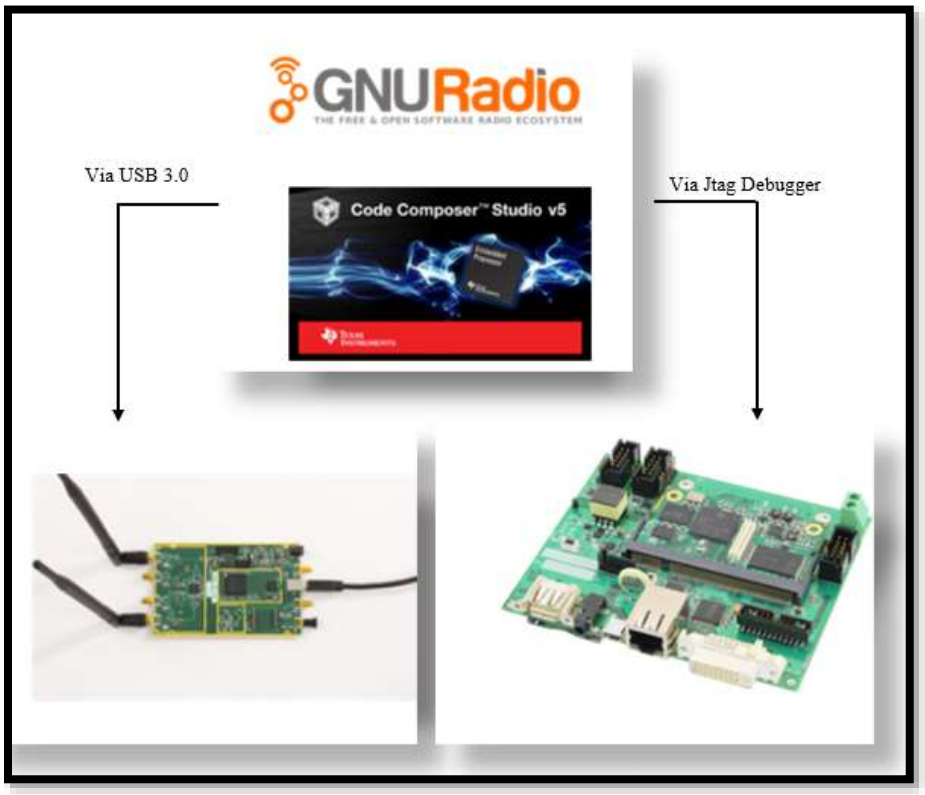

Figure 3-1 USRP and DSP kit connection

- Steps of model creation:

1- Develop codes of transmitter and receiver
2- Connect DSP kit with its emulator (debugger) to the laptop
3- Set the configuration file to deal with DSP emulator xdvs100
4- Start running transmitter code on CCS
5- Save Transmitter output to file
6- In GNUradio transceiver blocks (USRP channel), add the transmitter processed data
7- Connect USRP and save the data received on GNUradio
8- In CCS receiver, insert the data received and debug the code
9- Data received successfully

3.2 Project Testing

3.2.1 Functional Testing

We tested each function in the code by comparing its results by GNUradio block results using Octave tool.

3.2.2 Integration Testing

To test TX, we used GNUradio blocks and implemented the following:
1- Disabled the transmitter blocks from GNUradio
2- Add our transmitter results to be sent instead of disabled blocks
3- Receiver blocks is unchanged to check our transmitter functionality

To test the RX, we used CCS and GNUradio and implemented the following:
1- From GNUradio, take the input to the receiver by the help of octave
2- Read the file in CCS to the receiver code and run

3.3 Project phases

The project is divided into two main phases. Phase 1 which was done in the first half of the year and phase two in the second half.

3.3.1 Phase 1

- Standard IEEE802.11p understanding
- Adapting with GNU radio and studying IEEE blocks
- Standard verification with GNU radio blocks
- Studying USRP and try sending and receiving through two antennas and GNU radio
- DSP SDK understanding

3.3.2 Phase 2

- Setting up the environment either software or hardware tools
- Start implementing TX and RX C functions on DSP kit
- Testing
- Project Documentation

3.4 Project Cost

Table 4 Project Budget

Item	Specification	Price
USRP	Model: B200	686 $ (already available-Board only)
2 Antennas	Model: VERT900	36 $ each (already available)
DSP kit	Model:mityDSP OMAP-L138f	708 $
Emulator (Debugger)	Model: XDS100	79 $
Total Cost:		1545 $

Chapter 4

Tools Used

4.1 Software Tools

4.1.1 Gnu radio

In this section, we'll describe GNU radio a very important software that was used in the V2V PHY layer implementation, we'll first make an overview on it and describe its general usage, and then we'll go through how it was in the project.

4.1.1.1 Overview:

GNU Radio is a free software development toolkit that provides signal processing blocks to implement software-defined radios and signal-processing systems. It can be used with external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used in hobbyist, academic, and commercial environments to support both wireless communications research and real-world radio systems.

The GNU Radio software provides the framework and tools to build and run software radio or just general signal-processing applications. The GNU Radio applications themselves are generally known as 'flow graphs', which are a series of signal processing blocks connected together, thus describing a data flow. As with all software-defined radio systems, re-configurability is a key feature. Instead of using different radios designed for specific but disparate purposes, a single, general-purpose, radio can be used as the radio front-end, and the signal-processing software (here, GNU Radio), handles the processing specific to the radio application.

These flow graphs can be written in either C++or the Python programming language. The GNU Radio infrastructure is written entirely in C++, and many of the user tools are written in Python.

We used the GNU Radio Companion as a graphical UI used to develop GNU Radio applications. This is the front-end to the GNU Radio libraries for signal processing.

The main advantage of the gnu radio is that the standard 802.11is already implemented using C++and is an open source, so we used it as a test bed to verify that it's working with the 802.11p standard. Thus, we can use it for validation method when implementing the standard on the DSP kit.

4.1.1.2 Block diagrams of GNU radio

Wi-Fi Physical hierarchy

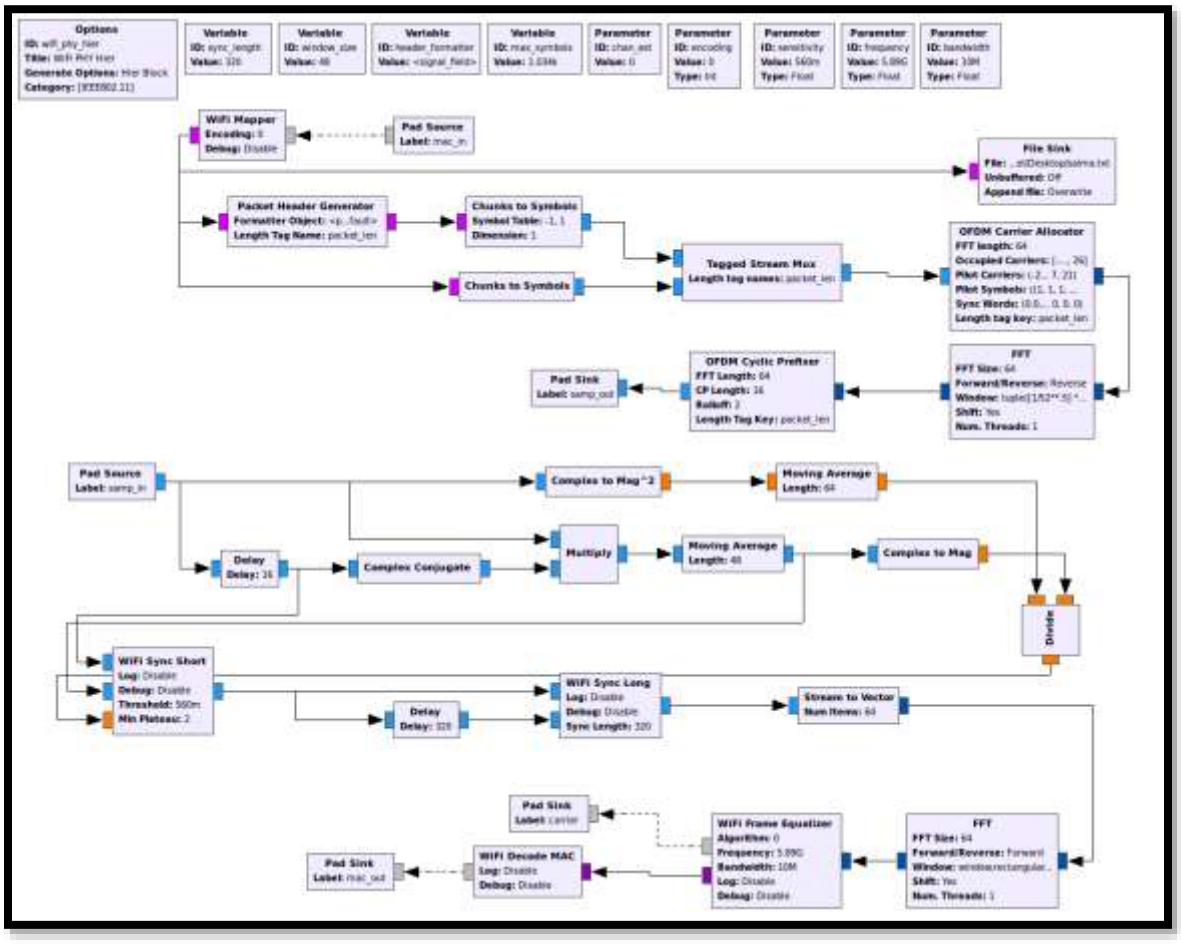

Figure 4-1 Physical hierarchy block diagram

The previous block diagram shows the blocks of the physical hierarchy in details and as explained in the standard. The Wi-Fi Mapper does the functions of the PLCP; scrambling, interleaving and splitting the data into symbols. The OFDM carrier allocator puts the symbols into the destined subcarriers, adds the pilots and prepares the OFDM symbol for the IFFT block to perform inverse fast Fourier transform. At the end, before transmitting, the cyclic prefix is added through the OFDM Cyclic pre-fixer block.

On the other half of the block diagram is the physical hierarchy of the receiver. This half reverses all the operations done on the transmitter after removing the cyclic prefix and recovering the signal from the traffic.

Wi-Fi transmitter

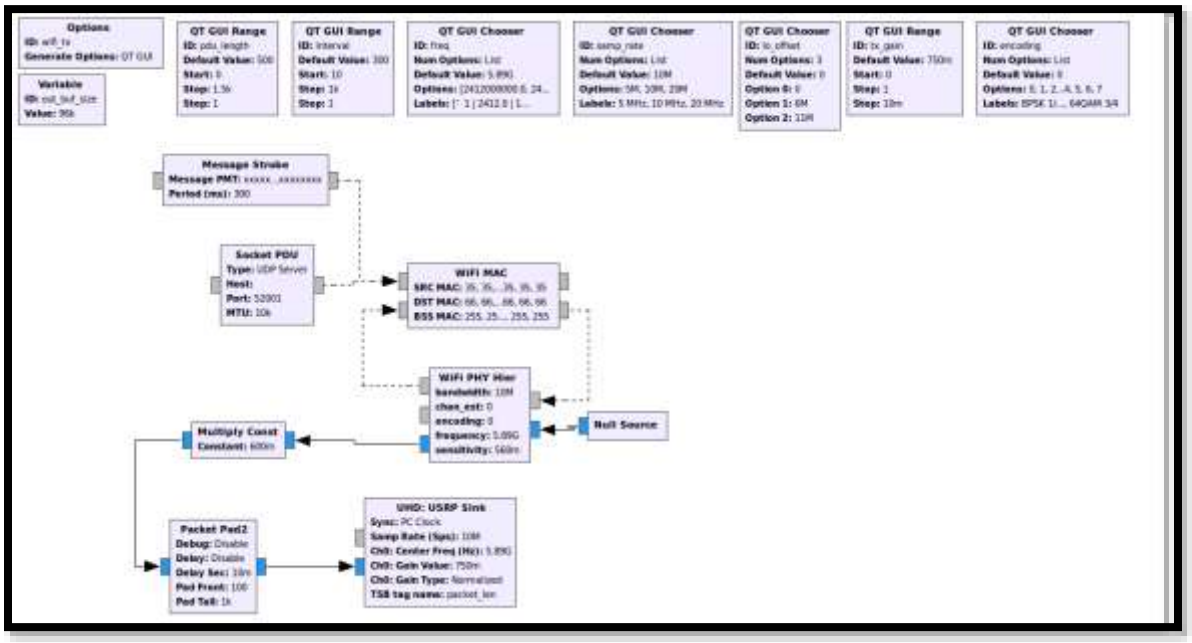

Figure 4-2 Transmitter block diagram

The physical hierarchy is all inserted into a single block called Wi-Fi PHY hierarchy. This block diagram shows the insertion of data into the MAC layer, then into the physical layer up to the USRP block which puts the data on the channel to be sent.

Wi-Fi Receiver

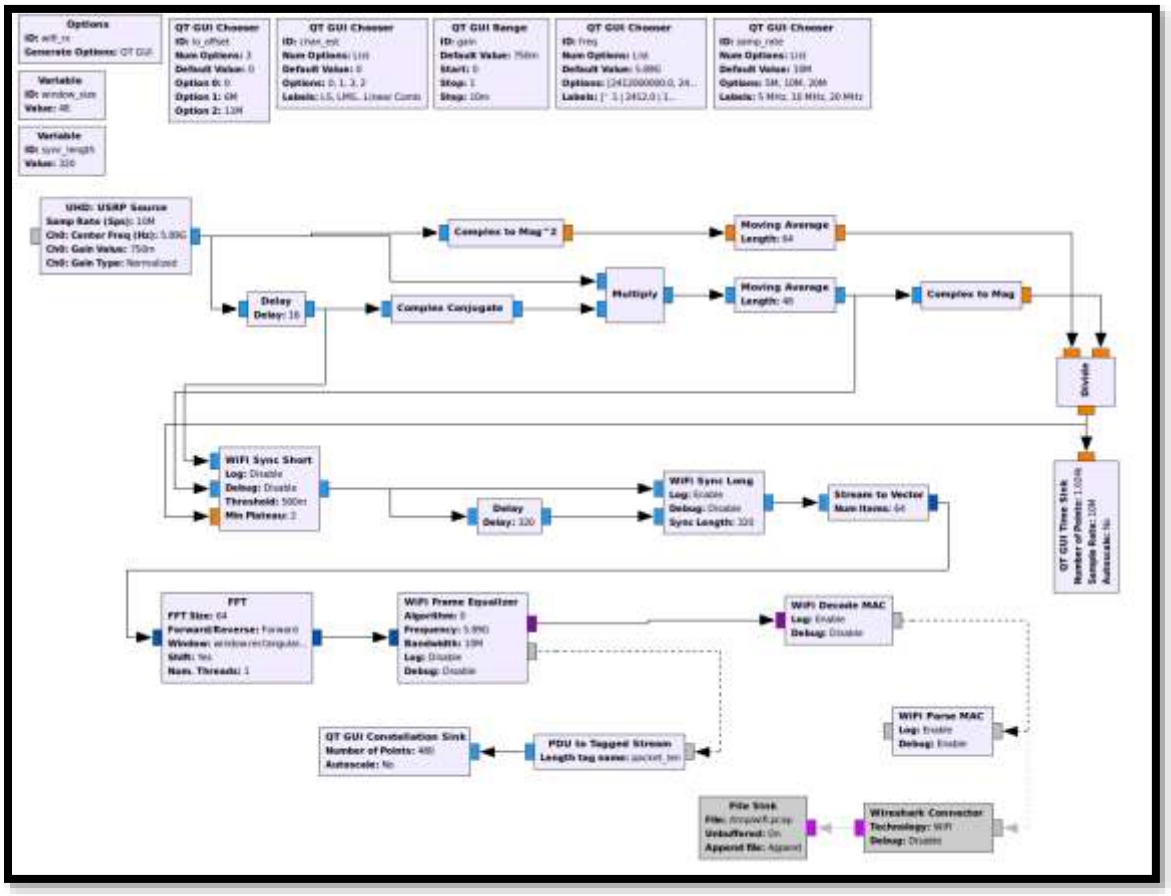

Figure 4-3 Receiver block diagram

This block diagram takes the receiver part of the physical hierarchy to receive the data from the channel first using the USRP block, then recover the data from the channel and start de-modulating, de-interleaving and de-scrambling the data.

Wi-Fi transceiver

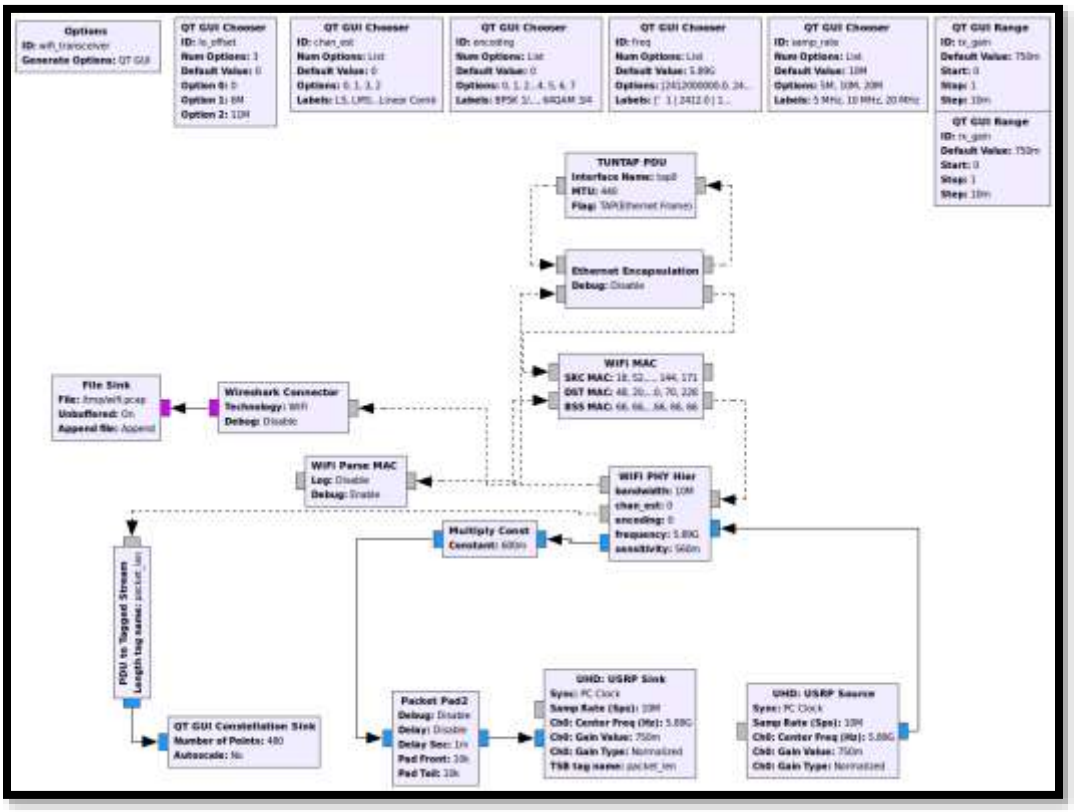

Figure 4-4 Transceiver block diagram

To show both sides in one block diagram, this block diagram shows the transmitter, the channel and the receiver. The channel here is created using USRP.

Wi-Fi loopback

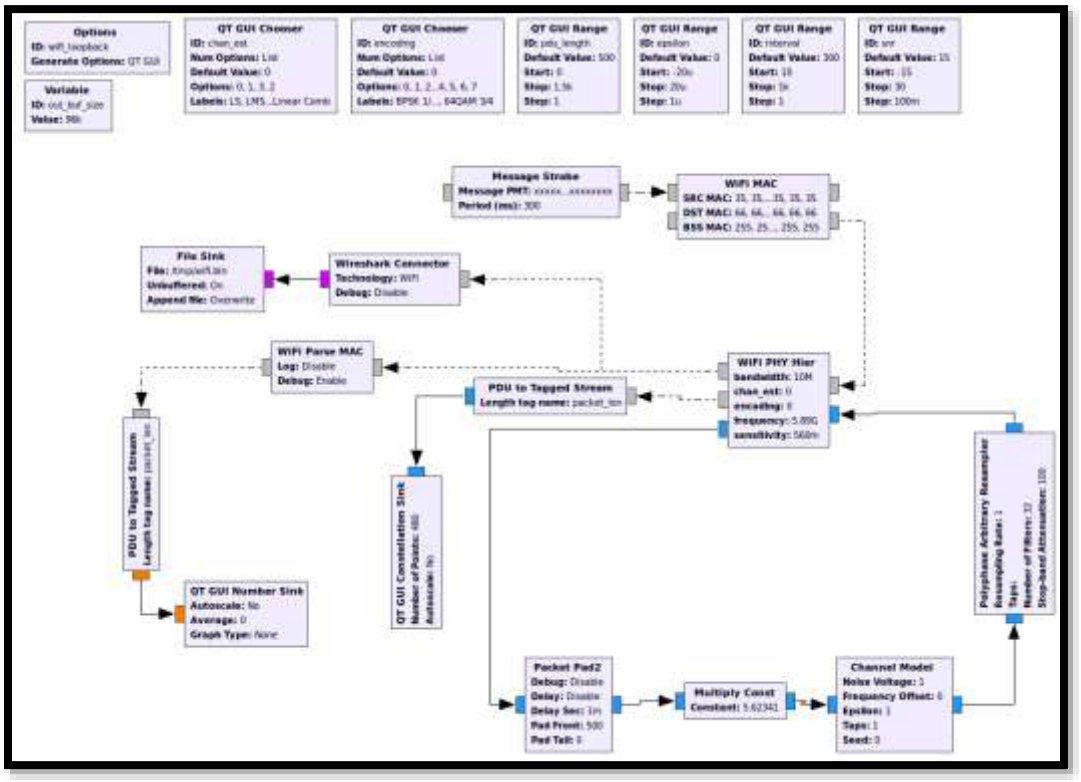

Figure 4-5 Loopback block diagram

Loopback also shows both sides in one block diagram, which are the transmitter, the channel and the receiver. The difference from transceiver is that the channel here is virtual channel model.

4.1.2 CCS (Code Composer Studio)

Code composer studio is an integrated development environment that supports TI's microcontrollers. It has tools to develop and debug embedded applications. These tools are an optimizing C/C++ compiler, source code editor, project build environment, debugger, profiler, and many other features. Code composer studio combines the advantages of Eclipse with advanced embedded debug capabilities of TI microcontrollers.

Code composer studio V7 is the latest version. It's efficient with the debugger used with the mitydsp kit (XDS 100v2). However, sometimes only a simulator is needed to make testing and trying easier. This is not available in v7 but it's available in CCS v5. CCS v5 has simulators that work for several DSP kits and processors. Since our project is DSP based, it was easy to find a simulator for (C674x processor) in CCS v5

In addition to that, there is a DSP library available for C674x processor. It has various useful functions that are rather used in our code such as FFT, IFFT and FIR filter. What's impressive about this library is that it's done using Assembly which increases its efficiency. Also, there are functions that are especially made for complex numbers which makes it a lot easier and more efficient for us to use this library.

Figure 4-6 CCS logo

4.1.3 GNU Octave

In this section, we'll describe GNU Octave a very important software that was used in the V2V PHY layer implementation, we'll first make an overview on it and describe its general usage, and then we'll go through how it was in the project.

4.1.3.1 Overview:

GNU Octave is software featuring a high-level programming language, primarily intended for numerical computations. Octave helps in solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with MATLAB. It also provides extensive graphics capabilities

Figure 4-7 Octave logo

for data visualization and manipulation. It is free software under the terms of the GNU General Public License.

4.1.3.2 The Octave language

The Octave language is an interpreted programming language. It is a structured programming language (similar to C) and supports many common C standard library functions, and also certain UNIX system calls and functions. However, it does not support passing arguments by reference.

Its syntax is very similar to MATLAB, and careful programming of a script will allow it to run on both Octave and MATLAB.

Because Octave is made available under the GNU General Public License, it may be freely changed, copied and used. The program runs on Microsoft Windows and most Unix and Unix-like operating systems, including macOS.

4.1.3.3 Usage in the project:

Since Octave can read the output file from any block in the gnu radio using a file sink, Octave has been used in this project to test the output from each block by comparing it to the output of the gnu radio block. It can read any type of data stored in a file with any size, with any format and convert to any type of data for displaying, here is a sample code for reading from a file.

1. PS1(">>")
2. addpath("/home/UserName/gnuradio/gr-utils/octave")
3. c=read_char_binary("File_Sink_Output.txt")

The second line of code is used to direct the path to the octave folder, the third line will read the file sink output that contains data in the form of characters and convert it to binary data and display it on the screen.

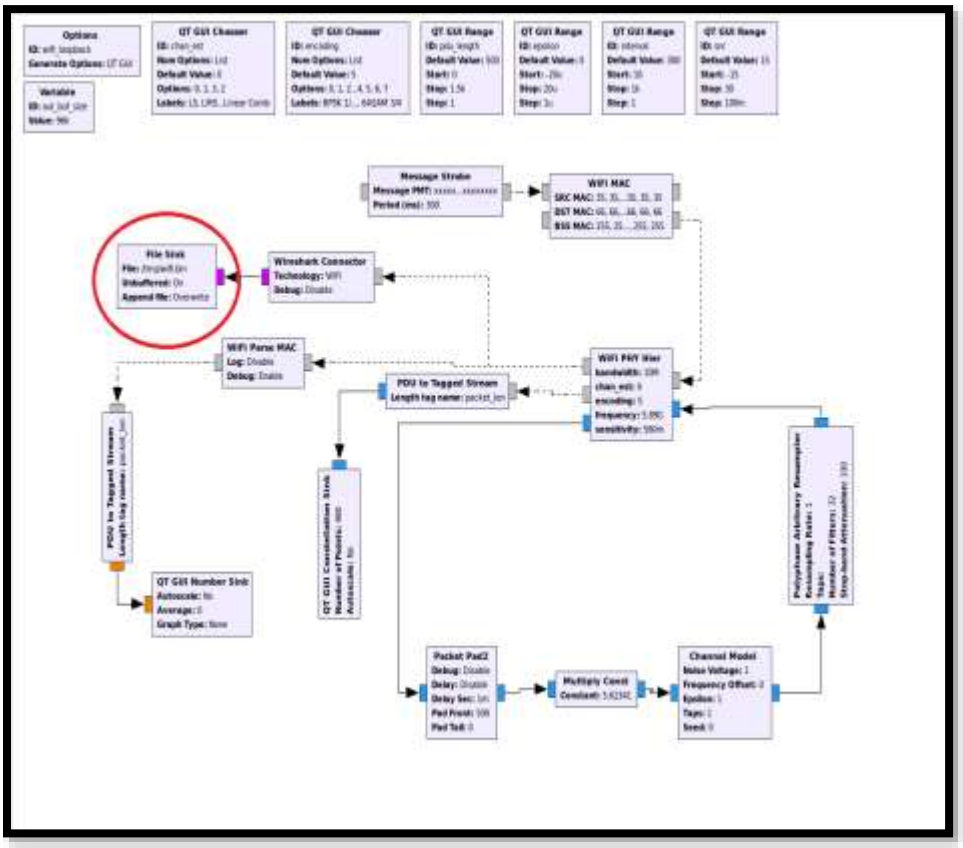

Figure 4-8 File Sink block

4.2 Hardware Tools

4.2.1 USRP

It stands for Universal Software Radio peripheral, the device is used as transceiver for radio frequency signals in wireless communication systems through the development of Software-defined radios.

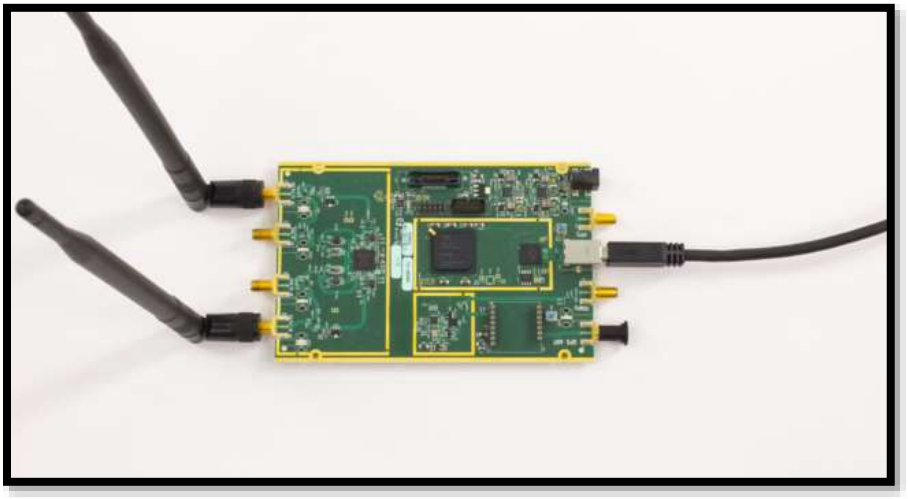

Figure 4-9 USRP B200

4.2.1.1 Hardware overview

"Following a common software-defined radio architecture, NI USRP hardware implements a direct conversion analog front end with high-speed analog-to-digital converters (ADCs) and digital-to-analog converters (DACs) featuring a fixed-personality FPGA for the digital down conversion (DDC) and digital up conversion (DUC) steps. The receiver chain begins with a highly sensitive analog front end capable of receiving very small signals and digitizing them using direct down conversion to in-phase (I) and quadrature (Q) baseband signals. Down conversion is followed by high-speed analog-to-digital conversion and a DDC that reduces the sampling rate and packetizes I and Q for transmission to a host computer using Gigabit Ethernet for further processing. The transmitter chain starts with the host computer where I and Q are generated and transferred over the Ethernet cable to the NI USRP hardware. A DUC prepares the signals for the DAC after which I-Q mixing occurs to directly up convert the signals to produce an RF frequency signal, which is then amplified and transmitted." *("What Is NI USRP Hardware? - National Instruments", 2017)*

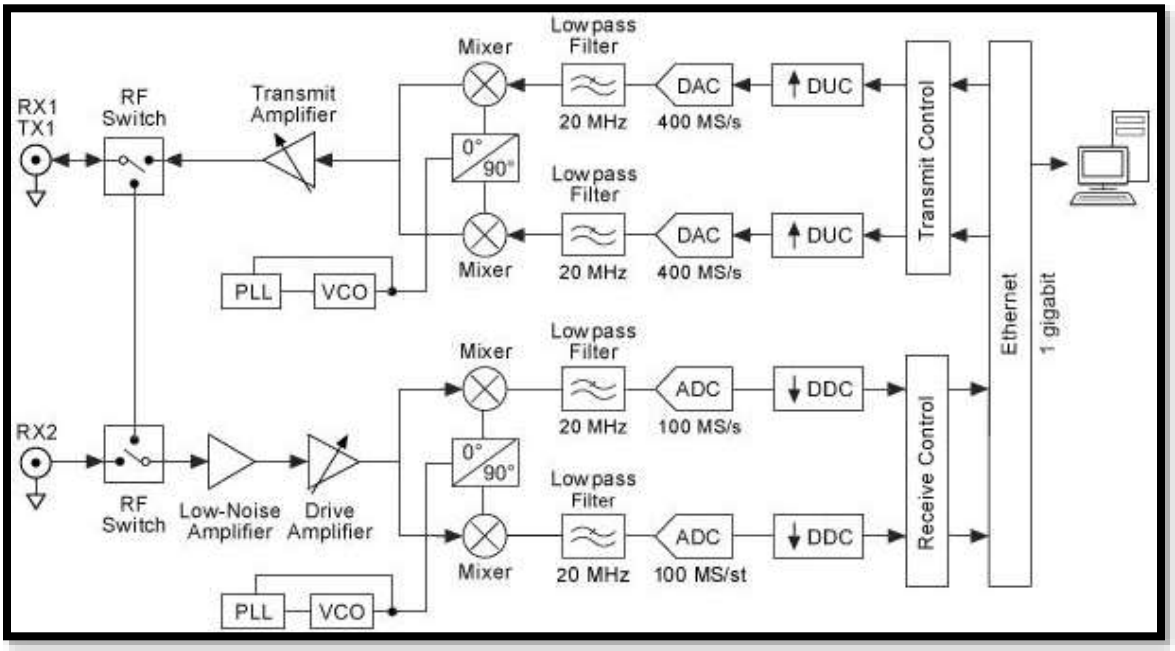

Figure 4-10 USRP 2920 system level diagram

4.2.1.2 USRP usage in our project

In this project the USRP B200 used to transfer the data from one end to the other end after setting the required antenna parameters and bandwidth occupied by the transferred data this happened through two principle blocks in GNU radio **USRP sink** is that responsible for adjusting the parameters at the transmitter side **USRP source** is that responsible for adjusting the parameters at the receiver side

In the following figure, the channel is represented by block called "channel model" in the loopback code for the IEEE standard 802.11, which is substituted by the USRP blocks for real transceiver connected through the USRP device.

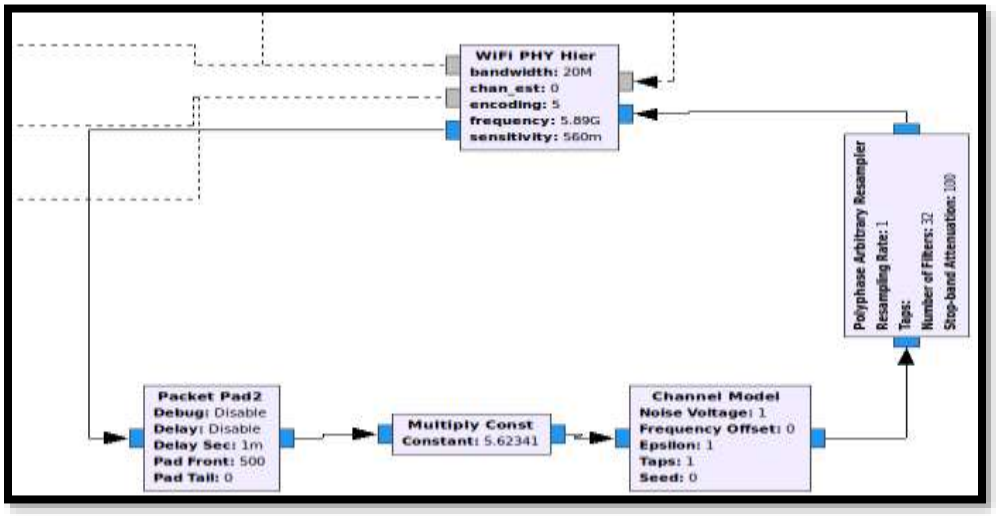

Figure 4-11 Simulated channel model using GNU radio

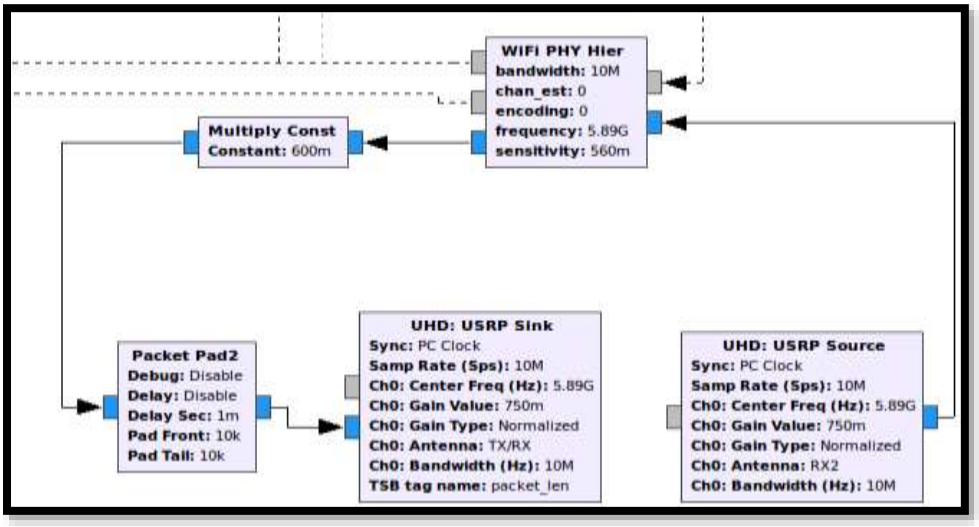

Figure 4-12 USRP channel using GNU radio

4.2.1.3 Challenges

One of our goals was to connect the USRP with the mitydsp kit directly. However, the DSP kit was considered a third party device with an operating system that is different than most operating systems that can install the USRP driver easily. A lot of challenges were faced while trying to install the libraries needed for the USRP driver. As a result, using the PC with Gnuradio as a communication host between the USRP and the DSP kit was a suitable satisfactory solution.

How we used the USRP and Gnuradio with our transmitter:

Gnu radio uses C++ language to make its blocks and Python to connect the blocks and make them communicate with each other. Using the C++ code of the last block of the transmitter, we managed to replace its code and make it write our own data that was made by our transmitter. That way, we can send our own data but by using Gnu radio's method of communication to send the data with no errors.

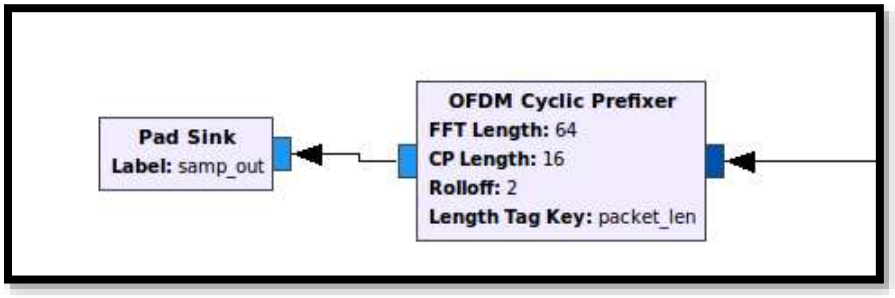

Figure 4-13 the last block in GNU Radio transmitter

```
/*----- COMMENT THIS-----------*/
    // 1) Figure out if we're in freewheeling or packet mode
//      if (!d_length_tag_key_str.empty()) {
//      symbols_to_read = ninput_items[0];
//      noutput_items = symbols_to_read * d_output_size + d_delay_line.size();
//      } else {
//      symbols_to_read = std::min(noutput_items / (int) d_output_size, ninput_items[0]);
//      noutput_items = symbols_to_read * d_output_size;
//
//
//      }
        /*-----------END OF COMMENT----------------*/
```

Figure 4-14 commenting code that is not needed

```
    /*-----------END OF COMMENT----------------*/

            noutput_items = 880; //Adding the number of output items coming from our own receiver

    // 2) Do the cyclic prefixing and, optionally, the pulse shaping
    for (int sym_idx = 0; sym_idx < symbols_to_read; sym_idx++) {
      memcpy((void *)(out + d_cp_size), (void *) in, d_fft_len * sizeof(gr_complex));
      memcpy((void *) out, (void *) (in + d_fft_len - d_cp_size), d_cp_size * sizeof(gr_complex));
      if (d_rolloff_len) {
        for (int i = 0; i < d_rolloff_len-1; i++) {
          out[i] = out[i] * d_up_flank[i] + d_delay_line[i];
          d_delay_line[i] = in[i] * d_down_flank[i];
        }
      }
      in += d_fft_len;
      out += d_output_size;
    }


    /*-------------------CCS TRANSMIT-------------------*/

    for(int i=0; i<880 ; i++)
    {
            out[i] = OutputNeeded[i]; // the output is eventually our own data. Any operation above is not important
    }

    /*----------------END OF TRANSMIT-------------*/
```

Figure 4-15 adding our own data

How we used the USRP and Gnuradio with our receiver:

Since Gnu radio has a block that communicates with the USRP, there wasn't any problems communicating with the USRP. However, the main concern was how to read the data after it's received and transfer it to our own receiver. So, we decided to add a file sink block after the USRP block. This block reads the received data and saves it into a file. After that, our receiver reads the data from this file and starts to analyze it and operate normally. That way, we facilitated the communication between Gnu radio, USRP and our own receiver using normal .txt files.

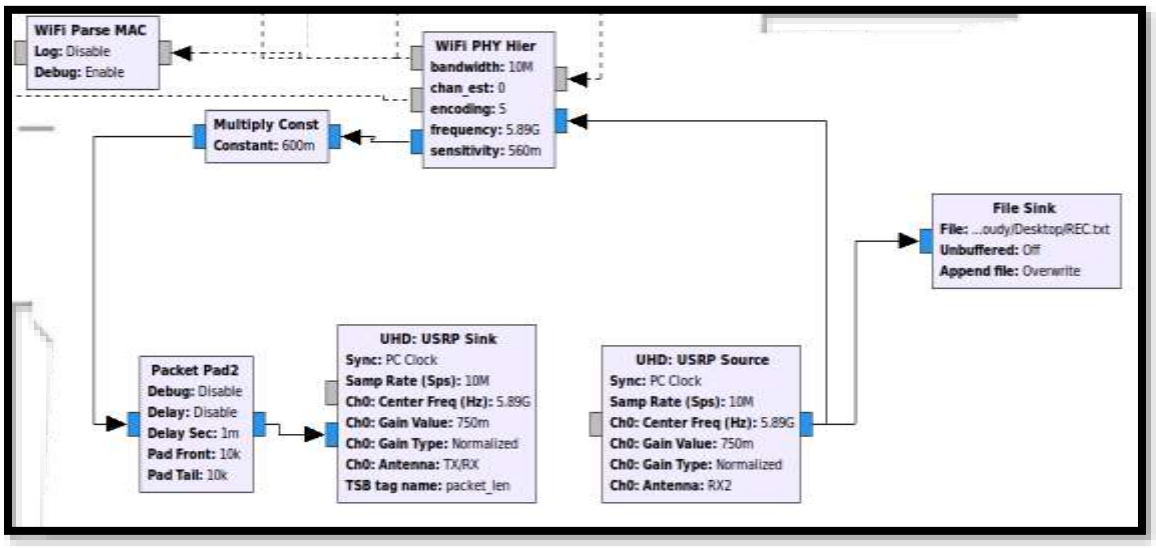

Figure 4-16 USRP blocks and File sink block

In this figure, the USRP sink is the block that takes the data to USRP to be sent. The USRP source is the block that receives the data from the receiving antenna and puts it to Gnu Radio. This received data is then written in the file that's path is written inside the block. FInally, our receiver reads the data from that same file and starts analyzing it.

4.2.2 MitydspL138F

In this section, we'll describe the kit that we used to create an environment applicable to test the V2V PHY layer implementation, we'll first make an introduction to DSP and its general usage, and then we will go through an overview on its Hardware and Software usage.

4.2.2.1 Introduction to DSP

It is an electronic board with Digital Signal Processor used for experiments, evaluation and development. A digital signal processor is a specialized microprocessor used to measure continuous real-world analog signals. Applications are developed in DSP Kits using a software usually referred as an Integrated Development Environment. Texas Instruments and Spectrum Digital are some of the companies who produce these kits.

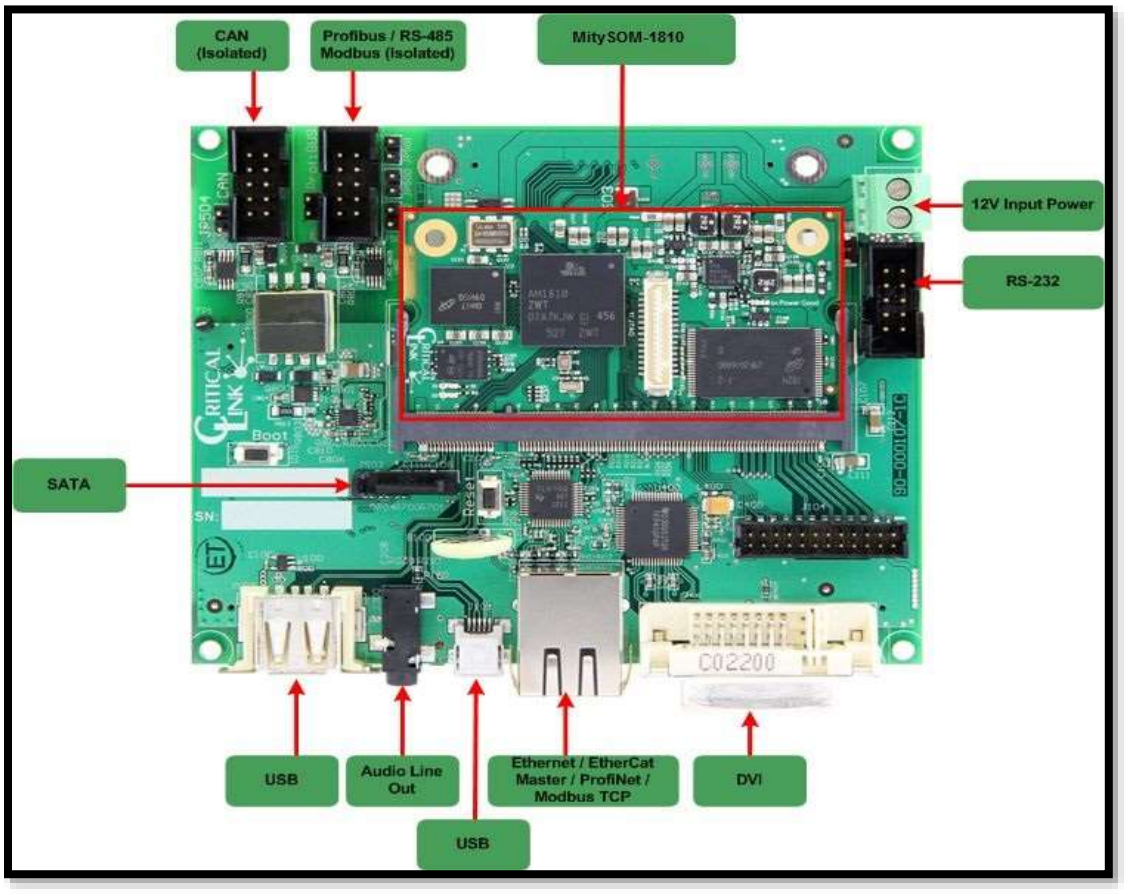

Figure 4-17 MityDsp-L138F

In our project, we are going to use a MitydspL138-F shown in the following figure, developed by Critical link an electronics product development company.

4.2.2.2 Overview on the MitydspL138-F:

The MityDSP-L138F System on Module (SoM) is the highest performance module in the OMAP-L138 family (a family of development kits) of MityDSPs. It features the dual-core OMAP-L138 CPU from Texas Instruments which provides both an ARM9 applications processor and a C674x Fixed / Floating Point DSP.

4.2.2.3 Applications:

1. Embedded Instrumentation
2. Industrial Automation
3. Industrial Instrumentation
4. Medical Instrumentation
5. Embedded Control Processing
6. Network Enabled Data Acquisition
7. Test and Measurement
8. Software Defined Radio
9. Bar Code Scanners
10. Power Protection Systems
11. Portable Data Terminals

4.2.2.4 Specifications:

- C674x Fixed / Floating point DSP

TI DSP Processor	C674x Fixed / Floating Point DSP
Max CPU Speed	456MHz
L1 Program Cache	32KB
L1 Data Cache	32KB
L2 Cache / Internal RAM	256KB

Table 4-1 C674x Fixed / Floating point DSP

- ARM processor

TI Applications Processor	ARM926EJ-S MPU
Max CPU Speed	456MHz
L1 Program Cache	16KB
L1 Data Cache	16KB
Internal RAM	8KB

Table 4-2 ARM processor

- Memory

System Memory	Available for Both CPUs
RAM	128MB to 256MB
NOR Flash	8MB
NAND Flash	256MB to 512MB

Table 4-3 Memory

- FPGA

FPGA Options	XC6SLX16	XC6SLX45
Slices	2,278	6,822
Logic Cells	14,579	43,661
Block RAM	576Kb	2,088Kb

Table 4-4 FPGA

4.2.2.5 Block diagram

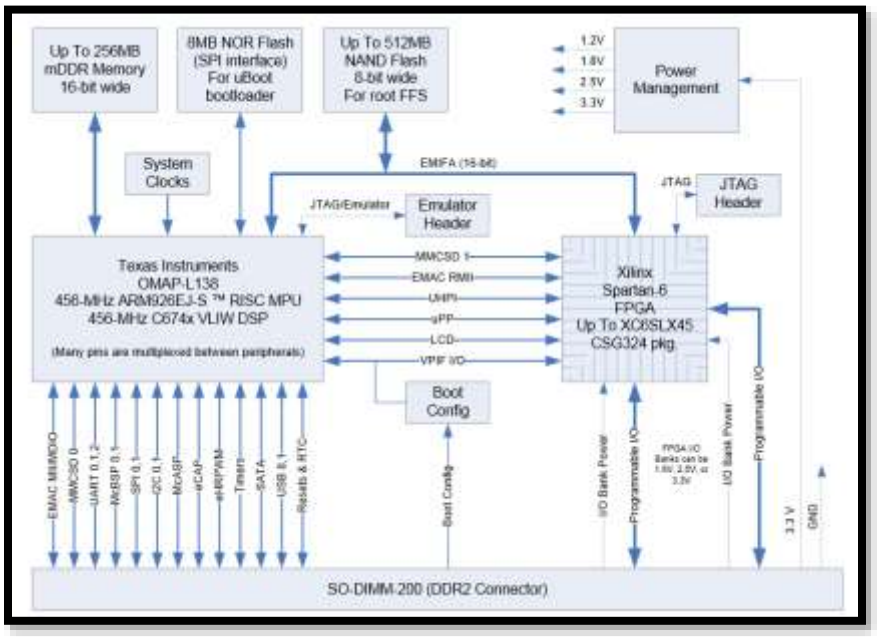

Figure 4-18 Block diagram

4.2.2.6 Interfaces

10/100 EMAC	SATA	eHRPWM
USB (2)	McASP	Timers
UART (3)	McBSP (2)	uPP
MMC/SD (2)	SPI (2)	uHPI
LCD	I2C (2)	
VPIF	eCAP	

Table 4-5 Interfaces

4.2.2.7 Mechanical

Dimensions	Length	Width
Inches	2.66	2.0
Millimeters	67.60	50.80

Table 4-6 Mechanical

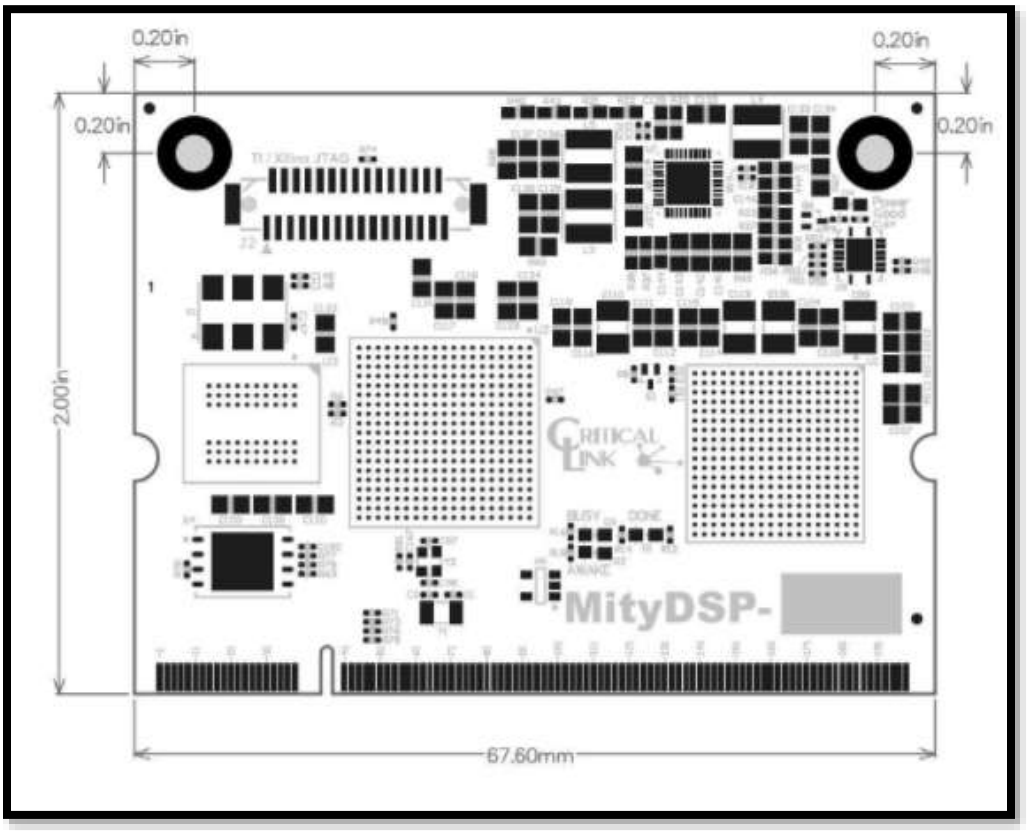

Figure 4-19 Dimensions

\

4.2.2.8 Development tools and software

Table 11 Software support

Software Support	CPU Subsystem
Real-time Linux Operating System	ARM9
ThreadX RTOS from ExpressLogic	ARM9
uBoot	ARM9
QNX Real-time Operating System	ARM9
Windows CE 6	ARM9
Qt Embedded Graphics	ARM9
DSP/BIOS Real-time Operating System	C674x DSP

Table 4-8 Development tools

Development Tool	Subsystem
Texas Instruments Code Composer Studio	ARM9 and C674x DSP
GNU Toolchain	ARM9
Xilinx ISE	Xilinx FPGA
Timesys LinuxLink	ARM9

4.2.2.9 UPP

The uPP (Universal Parallel Port) interface is one of the most important interfaces found in the mitydspL138F, It is particularly well suited to data acquisition through the on-board Xilinx FPGA.

It offers a very high-speed parallel data bus with several important features:

- Two independent channels with separate data buses
    - Channels can operate in same or opposing directions simultaneously
- I/O speeds up to 75 MHz with 8-16 bit data width per channel
- Internal DMA – leaves CPU EDMA free
- Simple protocol with few control pins (configurable: 2-4 per channel)
- Single and double data rates (use one or both edges of clock signal)
    - Double data rate imposes a maximum clock speed of 37.5 MHz
- Multiple data packing formats for 9-15 bit data widths
- Data interleave mode (single channel only)

4.2.2.10 Usage of the DSP kit in the project

The choice of this particular kit comes back to the fact that it's C674x Fixed / Floating Point DSP features an optimized general-purpose DSP function library as well as a MATH library for C Programmers typically used in computationally intensive applications. These libraries were very useful in the course of this project because many complex functions were needed through the implementation of the signal processing for both the transmitter and the receiver.

Another motive for choosing this particular kit, because of the UPP interface that was supposed to be used to transmit data with very high rates from the transmitter to the USRP. An XDS100v2 low cost JTAG debug probes (emulators) is used for the connection with the mitydsp kit in the software development using code composer studio. It provides the feature of debugging the code line by line without the need to download the code on the kit.

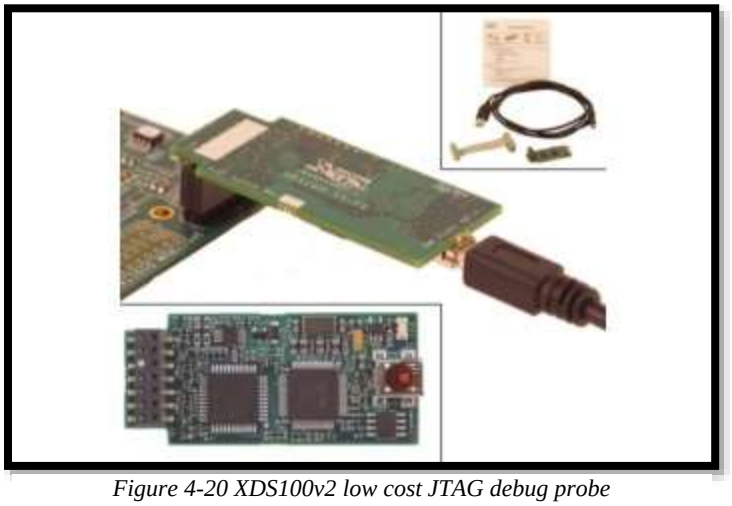

Figure 4-20 XDS100v2 low cost JTAG debug probe

Chapter 5

Code Description

5.1 Transmitter

5.1.1 Mapper

In this section, we will explain the Mapper block, the first block in the transmitter chain that performs all the signal processing on the data PSDU and append service bits,tail bits and pad bits to it to create multiple OFDM symbols depending on the length of the PSDU then pass it to the chunks to symbols for modulation.

5.1.1.1 Design:

The signal processing in the Mapper is composed of many detailed steps, which are described fully later in the implementation, the following overview intends to facilitate understanding the details of the design procedure sequence:

1. Calculate from the type of encoding the number of data bits per OFDM symbol (NDBPS), the coding rate (R), he number of bits in each OFDM subcarrier (NBPSC), and the number of coded bits per OFDM symbol (NCBPS). Refer to Table 1-1 for details.

2. Append the PSDU to the SERVICE field of the TXVECTOR. Extend the resulting bit string with zero bits (at least 6 bits) so that the resulting length is a multiple of NDBPS. The resulting bit string constitutes the DATA part of the packet. Refer to Figure 1-1for details.

3. Initiate the scrambler with a pseudorandom nonzero seed, generate a scrambling sequence, and XOR it with the extended string of data bits.

4. Replace the six scrambled zero bits following the data with six unscrambled zero bits (Those bits return the convolutional encoder to the zero state and are denoted as tail bits).

5. Encode the extended, scrambled data string with a convolutional encoder (R = 1/2). Omit (puncture) some of the encoder output string (chosen according to “puncturing pattern”) to reach the desired “coding rate”.

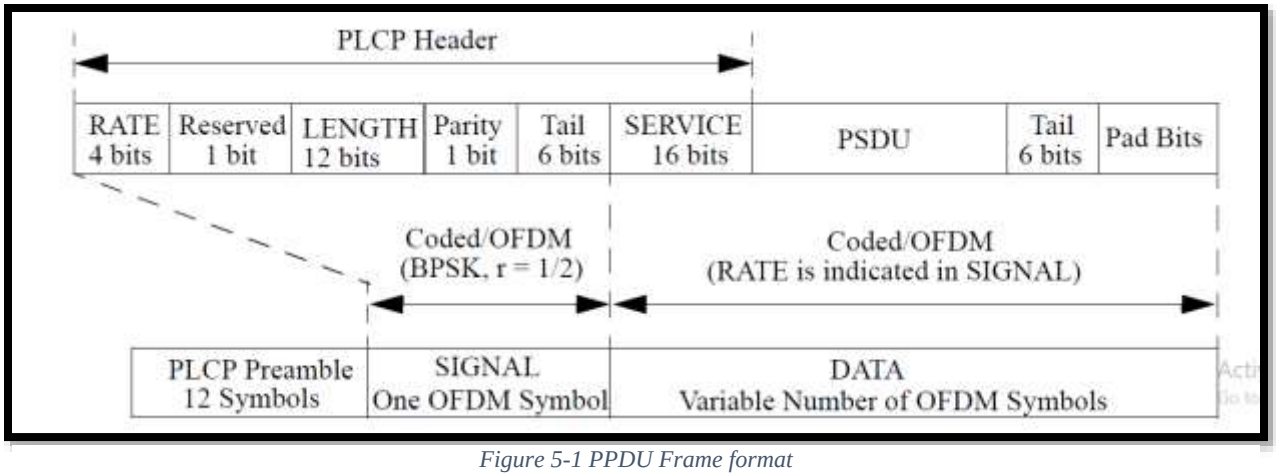

Figure 5-1 PPDU Frame format

Table 13 Modulation dependent parameters

Modulation	Coding rate (R)	Coded bits per subcarrier (N_{BPSC})	Coded bits per OFDM symbol (N_{CBPS})	Data bits per OFDM symbol (N_{DBPS})	Data rate (Mb/s) (20 MHz channel spacing)	Data rate (Mb/s) (10 MHz channel spacing)	Data rate (Mb/s) (5 MHz channel spacing)
BPSK	1/2	1	48	24	6	3	1.5
BPSK	3/4	1	48	36	9	4.5	2.25
QPSK	1/2	2	96	48	12	6	3
QPSK	3/4	2	96	72	18	9	4.5
16-QAM	1/2	4	192	96	24	12	6
16-QAM	3/4	4	192	144	36	18	9
64-QAM	2/3	6	288	192	48	24	12
64-QAM	3/4	6	288	216	54	27	13.5

5.1.1.2 Implementation:

The Mapper is divided into ten major functions, these functions contributes in the signal processing of the data and in the implementation of other blocks rather than the Mapper, and these functions are:

1. Void ofdm_param_intialization(Encoding e, ofdm_param* ofdm):

This function is used to initialize the OFDM parameters depending on the type of encoding as mentioned in Table 1-1, the OFDM parameters variable is implemented as a struct and consists of the following:

a. **Encoding** encoding; // Encoding type
b. **char** rate_field; // rate field of the SIGNAL header
c. **int** n_bpsc; // number of coded bits per sub carrier
d. **int** n_cbps; // number of coded bits per OFDM symbol
e. **int** n_dbps; // number of data bits per OFDM symbol

2. Void frame_param_intialization(ofdm_param* ofdm, frame_param* frame,int psdu_length):

This function is used to initialize the frame parameters depending on the parameter of the OFDM calculated in ofdm_param_intialization function and the PSDU size, the FRAME parameters variable is implemented as a struct and consists of the following:

a. **int** psdu_size; // PSDU size in bytes
b. **int** n_sym; // number of OFDM symbols (17-11)
c. **int** n_pad; // number of padding bits in the DATA field (17-13)
d. **int** n_encoded_bits;
e. **int** n_data_bits; // number of data bits, including service and padding (17-12)

3. Void generate_bits(const unsigned char *psdu, char *data_bits, frame_param* frame):

This function is used to append the 16 zero service bits before data, 6 tail bits and padding bits to the end of the PSDU, It also take a copy of every bit of the data into a separate byte to facilitate the signal processing later on.

The PSDU consists of an array of characters contain the data field, while the output of this function is an array of characters, the first 16 bits are the services bits ,then each character represent one bit of the data then 6 tail bits then pad bits.

4. Void scramble(const char *input, char *out, frame_param* frame,char initial_state):

This function implements the scrambler, the DATA field, composed of SERVICE, PSDU, tail, and pad parts that shall be scrambled with a length-127 frame-synchronous scrambler. The octets of the PSDU are placed in the transmit serial bit stream, bit 0 first and bit 7 last. The frame synchronous scrambler uses the generator polynomial S(x) as follows:

$$S(x) = x^7 + x^4 + 1$$

Equation 1 Scrambler

The 127-bit sequence generated repeatedly by the scrambler shall be (leftmost used first), 0000111011110010 11001001 00000010 00100110 00101110 10110110 00001100 11010100 11100111 1011010000101010 11111010 01010001 10111000 1111111, when the all ones initial state is used. The same scrambler is used to scramble transmit data and to descramble receive data. When transmitting, the initial state of the scrambler shall be set to a pseudorandom nonzero state. The seven LSBs of the SERVICE field shall be set to all zeros prior to scrambling to enable estimation of the initial state of the scrambler in the receiver.

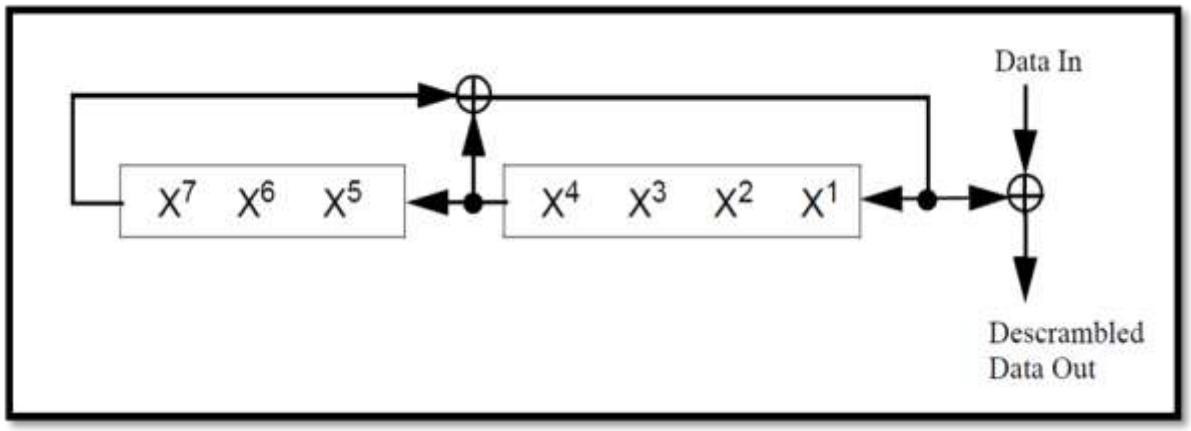

Figure 5-2 Data Scrambler

5. void reset_tail_bits(char *scrambled_data, frame_param* frame):

The 6 tails bits should be unscrambled as mentioned in the design process that follows the instructions of the standard of IEEE 802.11, since those bits return the convolutional encoder to the zero state. This function is used to reset these 6 tail bits to the zero state.

6. void convolutional_encoding(const char *input, char *out, frame_param* frame):

The DATA field, composed of SERVICE, PSDU, tail, and pad parts, shall be coded with a convolutional encoder of coding rate R = 1/2, 2/3, or 3/4, corresponding to the desired data rate. The convolutional encoder shall use the industry-standard generator polynomials, g0 = 1338 and g1 = 1718, of rate R = 1/2, as shown in Figure 1-3. The bit denoted as "A" shall be output from the encoder before the bit denoted as "B."

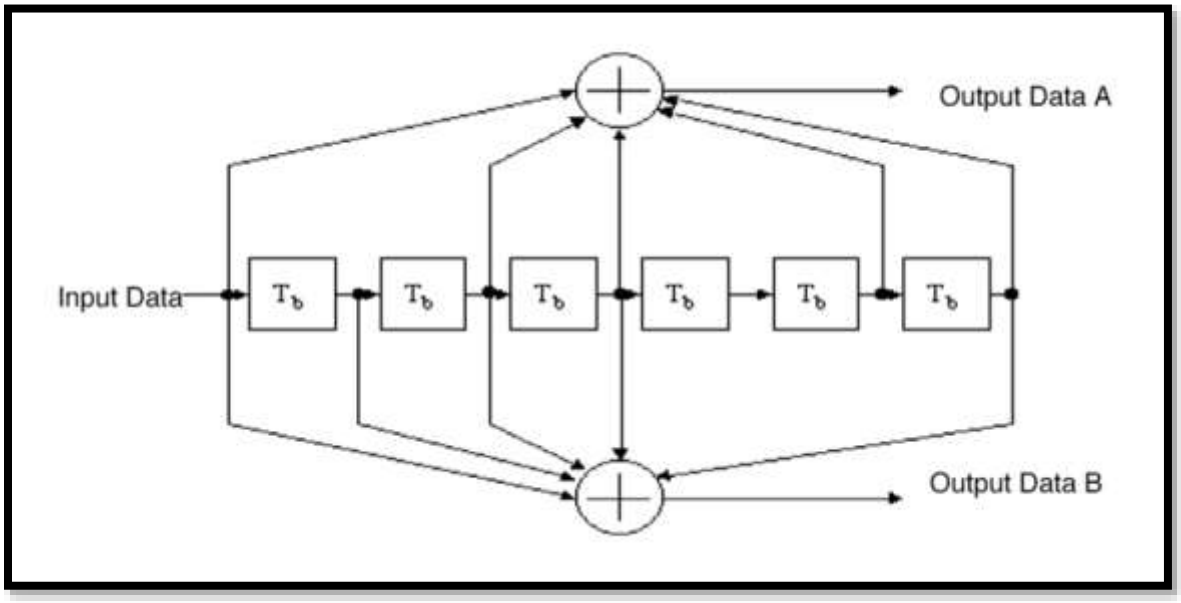

Figure 5-3 Convolutional encoding (k=7)

7. Void puncturing(const char *input, char *out, frame_param* frame, ofdm_param* ofdm):

Higher rates (2/3,3/4) are derived from convolutional encoding by employing "puncturing." Puncturing is a procedure for omitting some of the encoded bits in the transmitter (thus reducing the number of transmitted bits and increasing the coding rate) and inserting a dummy "zero" metric into the convolutional decoder on the receive side in place of the omitted bits.

8. Void interleave(const char *input, char *out, frame_param* frame,, ofdm_param* ofdm):

All encoded data bits shall be interleaved by a block interleaver with a block size corresponding to the number of bits in a single OFDM symbol, NCBPS. The interleaver is defined by a two-step permutation. The first permutation ensures that adjacent coded bits are mapped onto nonadjacent subcarriers. The second ensures that adjacent coded bits are mapped alternately onto less and more significant bits of the constellation and, thereby, long runs of low reliability (LSB) bits are avoided.

The index of the coded bit before the first permutation shall be denoted by k; i shall be the index after the first and before the second permutation; and j shall be the index after the second permutation, just prior to modulation mapping.

- The first permutation is defined by the rule:

$$i = (N_{CBPS}/16)\ (k \bmod 16) + \mathrm{Floor}(k/16)\ k = 0,1,\ldots,N_{CBPS} - 1$$

Equation 2 Interleaver first permutation

The function Floor (.) denotes the largest integer not exceeding the parameter.

- The second permutation is defined by the rule:

$$j = \mathrm{s} \times \mathrm{Floor}(i/\mathrm{s}) + (i + N_{CBPS} - \mathrm{Floor}(16 \times i/N_{CBPS})) \bmod \mathrm{s}\ i = 0,1,\ldots N_{CBPS} - 1$$

Equation 3 Interleaver second permutation

The value of s is determined by the number of coded bits per subcarrier, NBPSC, according to s = max (NBPSC/2, 1)

9. Void split_symbols(const char *input, char *out, frame_param* frame,ofdm_param* ofdm):

This function is used to split the data symbols according to the modulation type,e.g case of BPSK each symbol contains only one bit, so each element in the output array will contain only one bit,case QPSK each symbol contains 2 bits ,so each element in the output array will contain 2 bits,other Bits Per Symbol For Common Modulation Formats can be found in Table 1-2 ,this function is implemented using bit wise operations and shifting of the data bits according to the modulation type.

Table 10 Bits Per Symbol For Common Modulation Formats

Modulation Format	Bits/Symbol
BPSK	1
QPSK	2
8 PSK	3
8 QAM	3
16 QAM	4

10. unsigned char* mapper_general_work_function(const unsigned char* psdu,int psdu_length, ofdm_param* d_ofdm, frame_param * frame):

This function is the main function of the Mapper block, it calls all the functions stated above in the same order as they were mentioned to preform the signal processing mentioned in the design process.

5.1.1.3 Testing technique of the block:

In order to test this block, the correct output from each function is read from the Gnu radio by adding a print line in the code of the block after the function that needs to be tested, then the output is copied as an array to the CSS, subsequently compared to the output of the CSS by subtracting the two arrays.

5.1.2 Packet header generator

In this section, we will explain the packet header generator block, a parallel block to the Mapper in the transmitter chain, it produces the PLCP header excluding the service bits, it contains the LENGTH, RATE, reserved bit, and parity bit (with 6 zero tail bits appended) that constitute a separate single OFDM symbol, denoted SIGNAL, that is necessary for the demodulation, synchronization process at the receiver side.

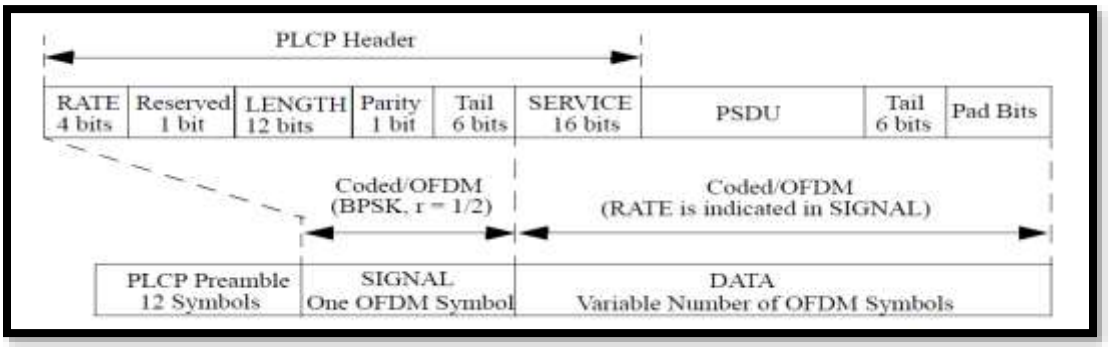

Figure 5-4 PPDU Frame Format

5.1.2.1 Design:

The OFDM training symbols shall be followed by the SIGNAL field, which contains the RATE and the LENGTH fields of the TXVECTOR. The RATE field conveys information about the type of modulation and the coding rate as used in the rest of the packet. The encoding of the SIGNAL single OFDM symbol shall be performed with BPSK modulation of the subcarriers and using convolutional coding at R = 1/2.

The encoding procedure of the signal field includes convolutional encoding, interleaving as used for transmission of data in the Mapper with BPSK-OFDM modulated at coding rate 1/2. The contents of the SIGNAL field are not scrambled.

The SIGNAL field shall be composed of 24 bits, as illustrated in Figure 1-2. The four bits 0 to 3 shall encode the RATE. Bit 4 shall be reserved for future use. Bits 5–16 shall encode the LENGTH field of the TXVECTOR, with the LSB being transmitted first.

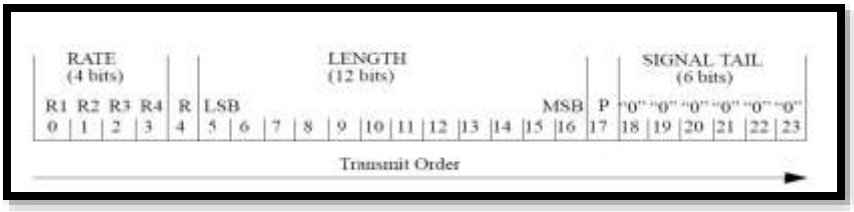

Figure 5-5 Signal field assignment

5.1.2.2 Implementation:

There is only **Two** new functions implemented for the signal field block:

1. **int get_bit(int b, int i) :**

This function returns the i^{th} bit in the int b variable,it is used to set the Rate field and the Length field consequently from the rate field in the OFDM parameters, and the psdu size in the FRAME parameters mentioned above in the explanation of the Mapper block.

```
int get_bit(int b, int i) {
return (b & (1 << i) ? 1 : 0);
}
```

2. **unsigned char * generate_signal_field(frame_param* signal_param, ofdm_param* signal_ofdm,frame_param* data_frame, ofdm_param* data_ofdm):**

This function calls the get_bit function to set the RATE field. Bit 4 is reserved. It shall be set to 0 on transmit and ignored on receive. Then the LENGTH field is set using the get_bit function, Bit 17 shall be a positive parity (even parity) bit for bits 0–16, finally 6 zero tail bits are inserted in order to facilitate a reliable and timely detection of the RATE and LENGTH fields.

It then preforms convolutional encoding with rate ½ then interleaving using the same functions implemented in the Mapper block in order to transmit the Signal field with the most robust combination of BPSK modulation and a coding rate of R = 1/2.

5.1.2.3 Testing technique of the block:

In order to test this block, the correct output from each function is read from the Gnu radio by adding a print line in the code of the block after the function that needs to be tested, then the output is copied as an array to the CSS, subsequently compared to the output of the CSS by subtracting the two arrays.

5.1.3 Chunks to symbols

5.1.3.1 Function:

The output divided Chunks of the encoded and interleaved binary serial input data bits as groups of (1, 2, 4, or 6) bits from the previous block are modulated by using BPSK, QPSK, 16-QAM, or 64-QAM, depending on the Encoding type and converted into complex numbers representing BPSK, QPSK, 16-QAM, or 64-QAM constellation points. The conversion is performed according to Gray-coded constellation mappings, illustrated in the standard.

Finally, it appends the modulated data field and to the modulated signal field in one array as an input for the next block.

5.1.3.2 Implementation:

To implement the modulation we use two functions:

1. The Constellation_implemenation function:

This function is responsible for generating the constellation by creating an array containing the complex numbers of this constellation and the size of this array is determined according to the coding type.

For example : if the encoding type is QPSK , then the array size will be four complex elements and as in this block we don't use the complex library for simplicity , we will have eight elements as shown in Figure 1 .

```
float *constellation_qpsk_impl() {
    const float level = sqrt((float) (0.5));
    float *d_constellation = calloc(4 * 2, sizeof(float));
    // Gray-coded
    d_constellation[0] = -level;
    d_constellation[1] = -level;
    d_constellation[2] = level;
    d_constellation[3] = -level;
    d_constellation[4] = -level;
    d_constellation[5] = level;
    d_constellation[6] = level;
    d_constellation[7] = level;

    return d_constellation;
}
```

Figure 5-6 QPSK constellation implementation function

Then this function returns a pointer to the created array to be used by the other function which is the Chunks to symbols implementation function.

2. The Chunks to symbols implementation function:

This function uses the constellation implementation function to create the array according to the encoding type and uses the created array to map each chunk of bits into the suitable complex number.

The idea of mapping is based on using the decimal value of the chunk bits as an index to the constellation array to get the suitable complex number corresponding to these bits.
For example: if we have this chunk of four binary bits [1000] , the chunks to symbols function maps these bits to the complex number at index =8 which is the equivalent decimal value of the chunk [1000] as shown in figure 2.

```
int i;
int index=0;
for (i=0 ; i < data_size ; i++ )
{
 index=(int)input_items[i];
 output_items[i*2]=d_mapping[index*2];
 output_items[(i*2)+1]=d_mapping[(index*2)+1];
}
```

Figure 5-7 Mapping of the chunks into the complex numbers

5.1.4 Tagged stream MUX

This block simply creates a frame that contains signal field followed by the rest of the frame.

5.1.5 OFDM carrier allocator

5.1.5.1 Standard requirements

According to the standard it is required to rearrange the 64 subcarriers entering the FFT with a certain sequence specified in the standard

The 64 subcarriers will be:

- Data carriers 48 subcarrier
- Pilot carriers 4 subcarriers
- Zero padding 12 subcarriers

Data carriers will be in {-26,-25,-24,-23,-22,-20,-19,-18,-17,-16,-15,-14,-13,-12,-11,-10,-9,-8,-6,-5,-4,-3,-2,-1, 26,25,24,23,22,20,19,18,17,16,15,14,13,12,11,10,9,8,6,5,4,3,2,1}

Pilot carriers will be in {-21,-7, 7, 21} subcarriers

5.1.5.2 Function Input

1. Data to be put on the OFDM frame
2. FFT-length which is 64 in the transceiver
3. Occupied_carriers 48 subcarrier positions specified in the standard
4. Pilot_carriers 4 subcarrier positions specified in the standard
5. Pilot_symbols the values of the pilot symbols
6. Sync word to be put in the beginning of the frame

5.1.5.3 Implementation

The sync words will be placed in the beginning of the frame directly. Then the input will be divided such that each part consists of 64 subcarriers to be delivered to the FFT.
The block consists of three objects for this, typically called `occupied_carriers` (for the data symbols), `pilot_carriers` and `pilot_symbols` (for the pilot symbols). `occupied_carriers` and `pilot_carriers` identify the position within a frame where data and pilot symbols are stored, respectively.

- Clarification example:

```
occupied_carriers = (-2, -1, 1, 3)
pilot_carriers = (-3, 2)
```

Every OFDM symbol carries 4 data symbols. They are on carriers -2, -1, 1 and 3. Carriers -3 and 2 are not used, so they are where the pilot symbols can be placed.

5.1.6 IFFT

5.1.6.1 Design

This block only performs the IFFT (Inverse Fast Fourier Transform) of the data to turn it from frequency domain to time domain to be sent to the channel after adding the cyclic prefix to it. The FFT size is 64. The only addition is that before it performs the IFFT to every 64 elements, it scales the data to the actual number of sub carriers (52) and performs a shift on the data before transforming it into time domain. Every 64 elements are divided into two arrays, the first array is shifted to be in the place of the second array and vice versa. Then the two arrays are combined again and an IFFT is performed on the data.

5.1.6.2 Implementation

One of the advantages of digital signal processing is the availability of a lot of DSP libraries in C language. Our DSP library has a ready–made function for the IFFT. However, this function doesn't perform the scaling or the shifting. So, we had to do both manually. First, every 64 elements are multiplied by 64 (to reverse the original normalization) and then divided by the square root of 52. Then the shifting is done using (memcpy()) function. Finally, the IFFT function is called to perform the Fourier transform. One of the great advantages of the IFFT function in the DSP library is that it's implemented in assembly to maximize the performance. Also, the function takes the complex numbers in the form of an array; the real numbers have an even index while the imaginary numbers have an odd index.

5.1.6.3 Testing

Since the window scale is not part of the normative specifications of the standard, there was no need to compare it to the standard. However, comparing it to Gnu Radio. It was easy to take the output of the IFFT block and store it into a file using File sink, then read the contents of this file using Octave. Comparing the ouput of IFFT to our own IFFT, the results were identical.

5.1.7 Cyclic prefix

5.1.7.1 Design

It consists of two parts:

First part cyclic prefix:

- Cyclic prefix usage:
  It increases the immunity to multipath fading.

Second part cyclic suffix and windowing:

- Cyclic suffix usage:
  It creates a smooth transition between the last sample of one symbol and the first sample of the next symbol.

5.1.7.2 Implementation

1. Cyclic prefix implementation:

It is implemented in CyclicPrefix function which exists in CyclicPrefix.c file as shown in figure1. It appends the last 16 samples of the 64 point IFFT to the front of the symbol, creating a composite symbol that is 80 samples long.

```
memcpy((out + (d_cp_size*2)),in, PtrToStruct->d_fft_len * sizeof(float)* 2);
memcpy(out,(in + (PtrToStruct->d_fft_len*2) - (d_cp_size*2)), d_cp_size * sizeof(float)* 2);
```

Figure 5-8 Implementation of cyclic prefix function in code

2. Cyclic suffix and windowing implementation:

It is implemented in CyclicPrefix function which exists in CyclicPrefix.c file as shown in figure2.

```
//---------- construct up flank and down flank ----------//
for (i = 1; i < PtrToStruct->d_rolloff_len; i++)
{
   d_up_flank[i-1]   = 0.5 * (1 + cos(M_PI *(float) i/(float)PtrToStruct->d_rolloff_len - M_PI));
   d_down_flank[i-1] = 0.5 * (1 + cos(M_PI *(float)(PtrToStruct->d_rolloff_len-i)/(float)PtrToStruct->d_rolloff_len - M_PI));
   d_delay_line[i-1]=0;
}
```

Figure 5-9 Implementation of cyclic suffix and windowing function in code

The window and the cyclic suffix length depend on the value of roll length of the raised cosine used.
Two windows are applied, one being the mathematical inverse of the other. The first raised cosine window is applied to the cyclic suffix of the previous symbol, and rolls off from 1 to 0 over its duration (down flank window). The second raised cosine window is applied to the cyclic prefix of the current symbol, and rolls on from 0 to1 over its duration (up flank window).
The cyclic suffix of the previous symbol multiplied by down flank window (delay line) is summed with the cyclic prefix of the next symbol multiplied by the up flank window as shown in figure3.

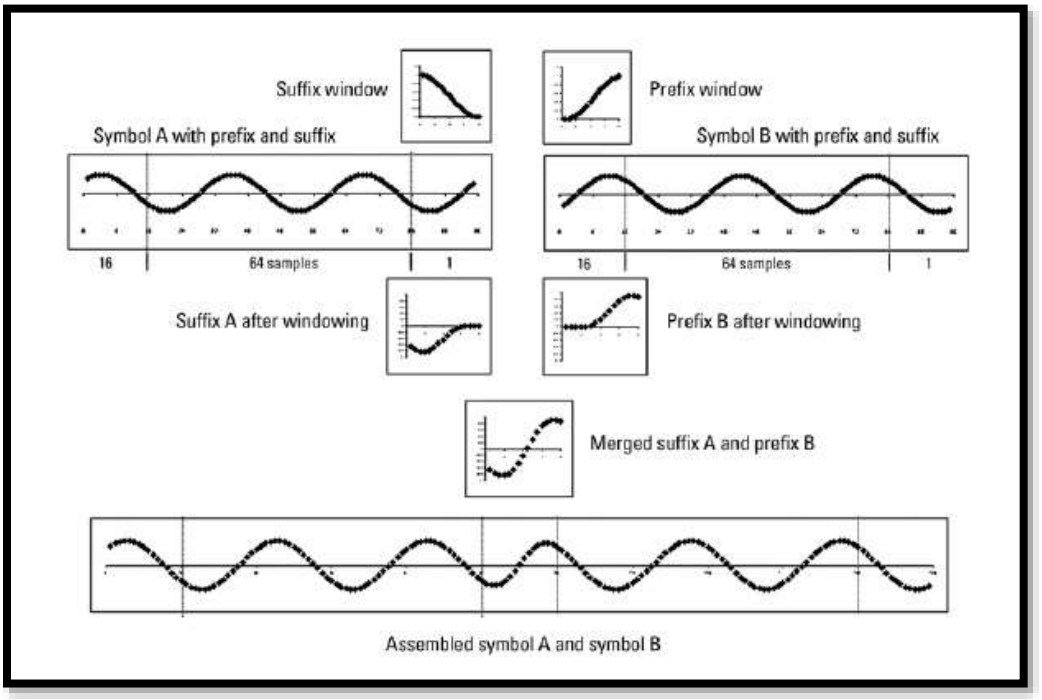

Figure 5-10 Cyclic prefix implementation

5.1.7.3 Testing technique of the block

First read input of cyclic prefix block in GNURADIO using OCTAVE and store it in an array called gnuradio_input in CCS then use it as input array of cyclic prefix function in CCS. After that read output of cyclic prefix block in GNURADIO using OCTAVE and store it in an array called gnuradio_output in CCS. Finally subtract gnuradio_output array from output of cyclic prefix function in CCS if we get an array of zeros then we succeed to implement the cyclic prefix function.

5.2 Receiver

5.2.1 The blocks before Synch short

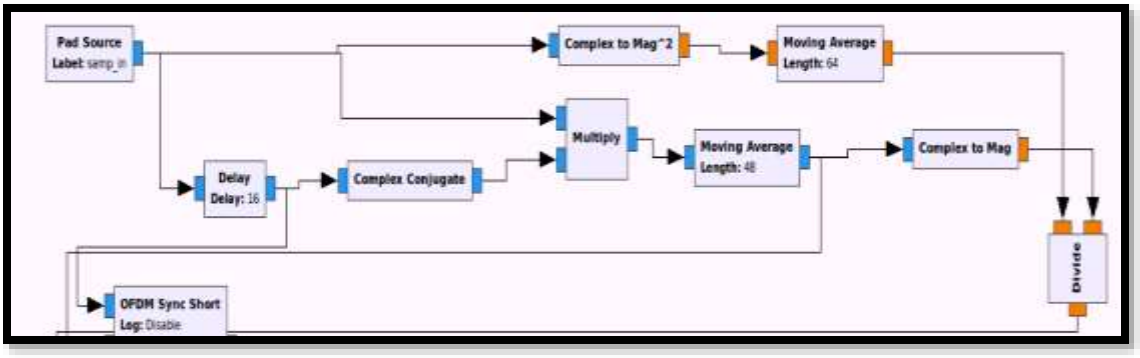

Figure 5-11 Blocks before sync short

5.2.1.1 Description

These blocks take the Input to the receiver which comes from the channel (either the channel model in GNuradio or USRP channel). As we discussed before, this part is responsible for frame detection and to detect the frame start we must make some calculations to reach to the auto correlated values

There are mainly three inputs to the sync short which are:

1. Delayed input
2. Window summation result
3. Auto-correlated values

5.2.1.2 Implementation

1. **void delayy(const float complex *input, float complex *output, short int delay , short int input_size, float complex delay_before_sync_short[])**

   - The function delay is used to pad zeros at the beginning of the frame by size delay
   - The output array size is input size + delay
   - Therefore, when the input is multiplied by delayed input, it will result to neglecting the 16 short sequence

2. **void movingAverage(float* arr,float* out ,short int size ,short int length,float*sum,float x[])**
   - Used to calculate the summation of each window frame
   - In the delayed path, we assume the window size to be (64-16 = 48)
   - In the non-delayed path, the window size is normally 64

5.2.2 Synch Short

5.2.2.1 Description

Completing the frame detection operation, sync short calculates the frame start, the frequency offset and calculates the indices that has some special conditions that will be discussed in the following part. Sync short mainly consists of two main cases which are:

Case Search:

It is used to search an index. This index describes the start of data that is more than 0.56 threshold. Known that there is a check which says that to return the index there must be three successive input more than threshold value (min plateau).

Consider the following example:

Assume that the following is the frame that is received from autocorrelation function

0.123	0.22	0.45	0.6	0.57	0.4	0.2	0.58	0.57	0.62	0.7	..	..	..

The start index that will come out in case of the previous table is 10 (value = 0.62)

Case Copy:

Once the index of the start of data is found. Case copy is used to copy all the rest of frame taking into consideration two things:

1- Minimum Gap case

Reaching min gap condition indicates that the case copy copied number of samples more than threshold which exceed the min gap value. In other words, if another frame arrives shortly after the first one, it won't be detected without minimum gap condition. Therefore when it reaches that condition it indicates that there is a start of a new frame (either it is a correct or wrong frame but it completed the size of min gap). The response to min gap case is normal, it will complete copying as it is and will break in two cases:

- If we reached input size (break and complete rest of the operations on the frames detected from min gap condition)
- If the counter copying (d_copied) reached max_samples

2- Maximum samples case

This case solves a limitation that may happen while copying. Which is the size of the frame that can be decoded is limited to a configurable number of OFDM symbols which doesn't contain a frame yet (all noise less than threshold after accessing case copy), Therefore, if we set a maximum number of samples which is a multiples of the OFDM frame size, we could set size limitation to solve the problem of not finding a frame (minimum gap case doesn't happen) which will let us return to Search case again and it will be stuck in sync short between search and copy if the minimum gap case doesn't happen.

5.2.3 Sync Long

5.2.3.1 Design

This block is responsible for frequency offset correction and the symbol alignment. As mentioned above, the symbol alignment as well as the frequency offset is calculated by getting the correlation of the received data with the long training sequence, getting the maximum peaks of this correlation and then detecting the frame start.
In our design, the correlation is extracted by using an FIR (Finite Impulse Response) filter.

5.2.3.2 Implementation

1. **Case SYNC:**

   In this case we are preparing the incoming data samples to detect the exact frame start of the frame and frequency offset of the samples by executing some functions that are described as follows

   1.1.FIR_Filter(…):

   It is used to calculate the correlation between the received samples with the well-known long training samples to calculate the exact frame start. this operation of the FIR filter is described as in Fig 5.12 such that:

   $$y[n] = h0.x[n] + h1.x[n-1] + \cdots + hN-1\,.x[n-N+1]$$

   Xn: nth element of the input data samples
   hn: nth element of the long training sequence array
   N: Number of complex long training sequence which is 64 in out case
   Yn: nth element of the FIR filter response

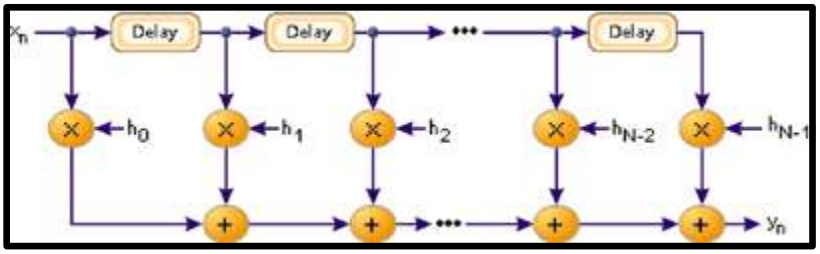

Figure 5-12 the logical structure of the FIR filter

1.2.Search_frame_start(…):

It is used to calculate the frequency offset and the index of the first training sample in the received frame to be fed to the FFT.
So first we reorder the resulted correlation from the FIR filter to get the maximal two correlated samples so that we can get the start of the frame and the frequency offset to be corrected.

2. Case COPY:

In this case the output of sync long is being constructed taking into account the beginning of long training sequence followed by the OFDM data symbols, removing the cyclic prefix and correcting the frequency by multiplying by frequency offset that was calculated previously.

5.2.3.3 Testing technique of the block

The testing process here is achieved through testing three parts:

1. **FIR_Filter testing:** The input and the output of the FIR filter is read through OCTAVE from the GNU Radio simulation results then the results is compared through showing the difference from that in GNU Radio and what is implemented.
2. **Frequency offset and frame start testing:** It is achieved by enforcing the same input array of samples obtained from the GNU Radio to the implemented Search_frame_start() and observing the frequency offset and the frame start achieved.
3. **Output array:** By observing the difference between the output array of sync long and that obtained from the GNU radio

5.2.4 FFT

5.2.4.1 Design

The FFT block is very similar to the IFFT. The only difference is that it doesn't need scaling. Also, the shifting is done after the FFT not before it; unlike the IFFT block.

5.2.4.2 Implementation

Using the FFT function from the DSP library, which is also written in Assembly to maximize the performance, the data was transformed to frequency domain. After that, the data was shifted using (memcpy()) function to return it to its original positions.

5.2.4.3 Testing

By reading the output of GNU radio's FFT block though Octave and comparing the data with our FFT output, the results were identical.

5.2.5 Frame Equalizer

5.2.5.1 Design

This block has four main important roles

1. Compensating the frequency and phase offsets through the Sync long and Sync short offsets and through the pilots
2. Calculating the model channel through various techniques (LS, LMS, STA or COMB)
3. Identifying the Data length, modulation technique and parsing technique used through analyzing the signal field
4. Compensating the data and transforming them from complex numbers to octets to be further analyzed by the next block

This block deals with each OFDM symbol before it goes to the other. After reading 64 complex numbers (1 OFDM symbol), it compensates every sample in the data with the frequency offsets calculated from the previous blocks (Sync short and Sync long) as well as the sampling offsets. Next, residual frequency offset is calculated by adding the phase of the four pilots of every symbol and compensating the OFDM symbol with this offset. To update this offset and correct the next symbol with it as well, the phase difference between the pilots of adjacent OFDM symbols is calculated as well to correct the next symbol. Then, the channel model is calculated using the long preamble sequence which are the first two OFDM symbols entering the equalizer. After that, the signal field is estimated using the channel model and de-modulated then further analyzed to know the data length, the modulating technique and the parsing rate. Finally, every data symbol is compensated like the others, estimated using the pre-calculated channel model, de-modulated and becomes the output of the block.

Signal field decoding

In each frame, the short and long training sequences are followed by the signal _field, which is a BPSK modulated OFDM symbol encoded with a rate of 1/2 that carries information about the length and encoding of the following symbols. The first step done in this function is to de-interleave the signal field and then decode the output bits using a Viterbi decoder.
If the signal _field is decoded successfully, i.e., if the rate_field contains a valid value and the parity bit is correct, the Decode Signal Field returns the type of encoding of the data and the number of symbols in each frame and passes it to the next block Frame Decode block.

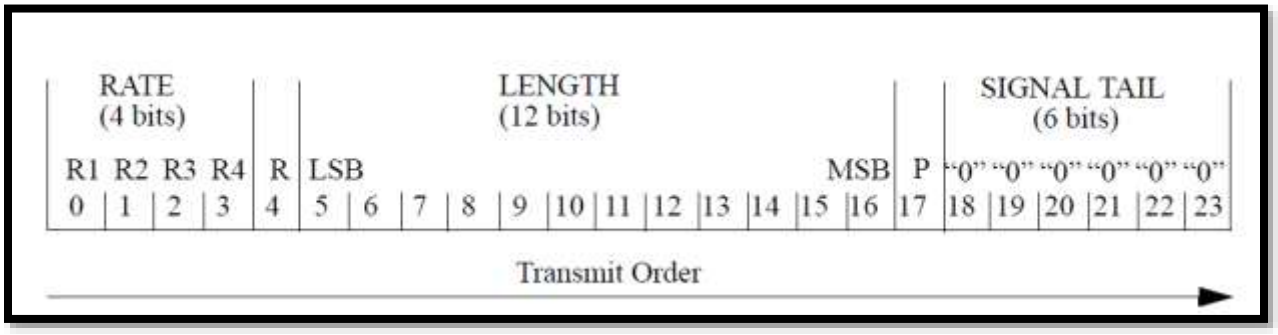

Figure 5-13 Signal Field Assignment

Channel estimation techniques

1- LS

This technique is called the least squares. It uses the received long preamble symbols and the saved version of the symbols to estimate the channel. Then, it uses the channel to estimate all 48 symbols inside the OFDM symbol.

$$H\hat{}(k) = \frac{Y1(k) + Y2(k)}{2Xlt(k)}$$

H(k) is the channel model. Y1(k) and Y2(k) are the received long training symbols while Xlt(k) is the saved long training sequence.

2- LMS (Least minimum squares)

The LS technique is very efficient. However, it suffers when the frame gets longer or the coherence time gets shorter. This technique solves the problems that the LS technique can't solve. Not only does it estimate the channel using the same way as LS, it also calculates the error percentage coming from the difference between the actual channel and the estimated channel. That way, it updates the channel in every single symbol to be more accurate than LS technique.

It updates the channel after the ith OFDM symbol using the constellation point Xi that the received Yi was de-mapped to.

$$\mathrm{H\hat{}i(k)} = (1-\alpha)\mathrm{H\hat{}i-1(k)} + \alpha\frac{Yi(k)}{Xi\hat{}(k)}$$

Where Hi(k) is the channel model used for the next symbol, Hi-1 is the channel model used for the current symbol. It's discovered that the best design is when alpha is 0.5.

The LS and LMS techniques use every subcarrier independently and they don't use averaging in frequency domain.

3-STA (The Spectral Temporal Averaging)

What makes this technique different is that it doesn't deal with every subcarrier independently, it correlates the channel coefficients in the frequency domain as well. First, the LS estimate is used as an initial estimate then data decision feedback is done by demodulating the first data symbol compensated by the LS initiate estimate.
After that, a more accurate channel estimation is done by dividing the received data with the demodulated data as follow

$$Hi(k) = \frac{Yi(k)}{Xi\hat{}(k)}$$

Then, the frequency domain correlation is done by using this equation

$$Hupdated(k) = \sum_{i=-\beta}^{\beta} WiH(k+i)$$

H updated is the channel estimate based on the correlation between the neighboring subcarriers, β is the window size where the weighted average happens and Wi is the weight of each in the window subcarrier. After the frequency domain averaging is done, the time domain averaging is done using the factor α.

$$H(t) = \left(1 - \frac{1}{\alpha}\right) H(t-1) + \frac{1}{\alpha} Hupdated(t)$$

4-COMB

This type is totally different than the other three techniques. Comb Type channel estimation uses the information about the channel at the pilots' location to be able to update the channel estimate to track channel variations during the same OFDM symbol. The Comb equalizer interpolates linearly in frequency domain using the four pilots and the mean of the pilots as well. The mean value of the pilots are used at the border of the vector used for interpolation [mp, P1, P2, P3, P4, mp], where P1..4 are the four comb pilots and mp is their mean. This interpolation is done for every OFDM symbol. Afterwards, a low-pass filter similar to the previous techniques is done over the channel in time domain.

$$H(t) = \left(1 - \frac{1}{\alpha}\right) H(t-1) + \frac{1}{\alpha} Hupdated(t)$$

Where H updated is the updated channel model using the linear interpolation of the four pilots.

5.2.5.2 Implementation

After compensating the frequency offsets and the sampling offsets by adding these offsets to the phase of the current symbols, there are two important functions to be discussed

1- Signal field implementation

There are **three** functions used in the signal field decoding:

a. deinterleave(uint8_t *rx_bits , uint8_t * d_deinterleaved):

This function takes the received bits and the deinterleaving sequence as an array and performs the inverse relation of the interleaving that is also defined by two permutations. Here, the index of the original received bit before the first permutation shall be denoted by *j*; *i* shall be the index after the first and before the second permutation and *k* shall be the index after the second permutation, just prior to delivering the coded bits to the convolutional (Viterbi) decoder.

The first permutation is defined by the rule:

$$i = s \times \text{Floor}(j/s) + (j + \text{Floor}(16 \times j/N_{CBPS})) \bmod s \; j = 0,1,\ldots N_{CBPS} - 1$$

Equation 4 First permutation

Where *s* is defined before in the interleaving function in the Mapper block.
The second permutation is defined by the rule:

$$k = 16 \times i - (N_{CBPS} - 1)\text{Floor}(16 \times i/N_{CBPS}) \; i = 0,1,\ldots N_{CBPS} - 1$$

Equation 5 Second permutation

b. bool parse_signal(uint8_t *decoded_bits):

This function takes the output decoded bits from the Viterbi decoder and finds the rate field and length field by shifting and using bit wise operation, it also computes the parity bit of the first 17 bits and if the parity bit is correct, then it uses a switch case to return the type of encoding and subsequently find the number of symbols in the frame.

Table 11 The rate field content

R1–R4	Rate (Mb/s) (20 MHz channel spacing)	Rate (Mb/s) (10 MHz channel spacing)	Rate (Mb/s) (5 MHz channel spacing)
1101	6	3	1.5
1111	9	4.5	2.25
0101	12	6	3
0111	18	9	4.5
1001	24	12	6
1011	36	18	9
0001	48	24	12
0011	54	27	13.5

c. bool decode_signal_field(uint8_t *rx_bits):

This function calls all the above function in the same order as they were mentioned, it also allocates memory for the output bits and the OFDM parameters and FRAME parameters for correct parsing of the signal field.

2- Channel estimation

The two most important functions to discuss are the following:

a. Unsigned char decision_maker(unsigned char[])

This function's main task is demodulating the estimated symbol from the channel response coefficient according to its modulation technique.

- For BPSK

The bits are estimated by observing the signs of the real part of the symbol.

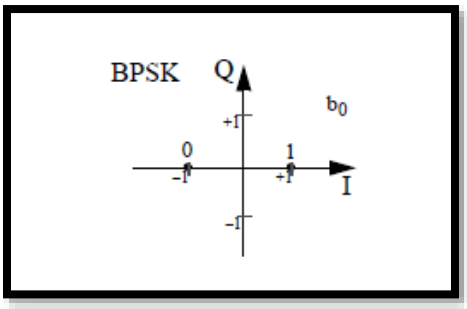

Figure 5-14 BPSK constellation

- For QPSK

The bits are estimated by observing the signs of the real and imaginary parts of the symbol

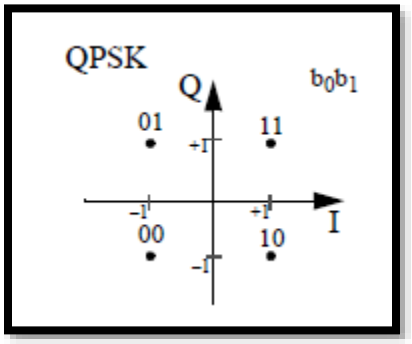

Figure 5-15 QPSK constellation

- For 16-QAM

This estimation is more complicated as the constellation is divided into more levels. Not only does it observe the signs of the real and imaginary numbers, it also the level of them.

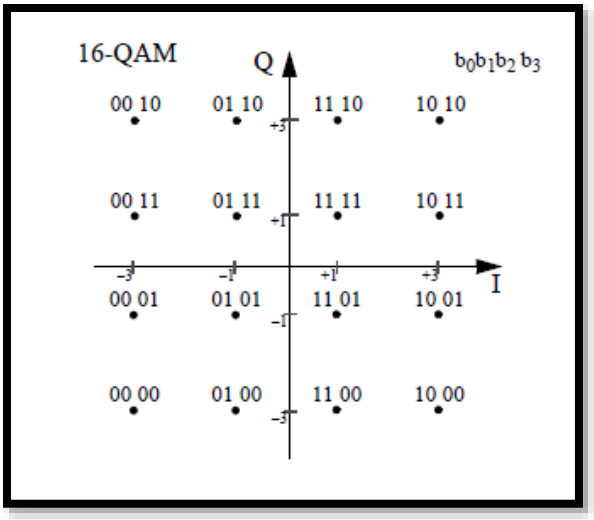

Figure 5-16 16-QAM constellation

- FOR 64-QAM

It's the same as 16-QAM constellations but with more constrictions to the level of the real and imaginary parts of the symbol.

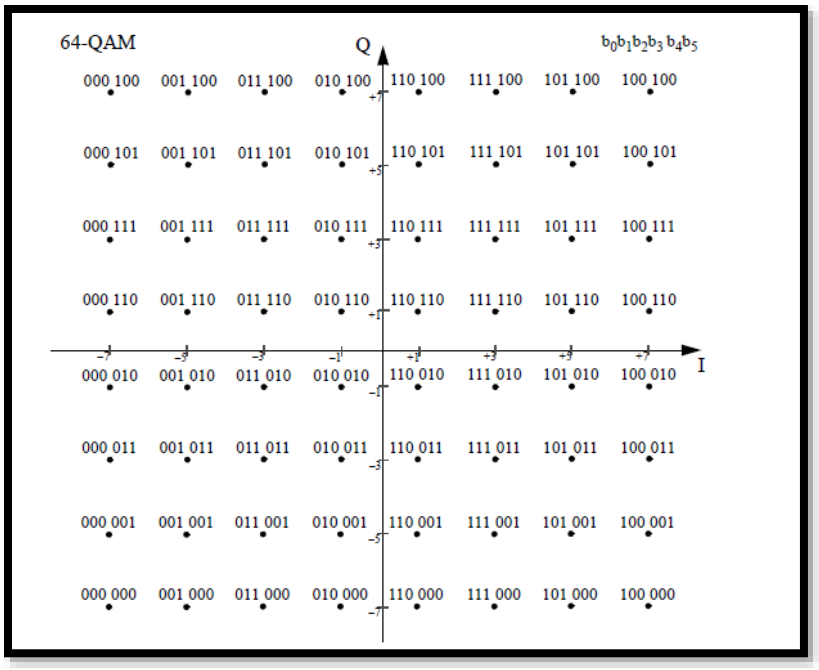

Figure 5-17 64-QAM constellation

b. Linear interpolation in COMB channel estimation technique

As mentioned before, the COMB channel estimation uses linear interpolation of the pilots of each symbol along with the mean value of the pilots. The linear interpolation is done as follow:

For the first 11 symbols the interpolation is done using the average value of pilots along with the first pilot. For the 12th symbol until the 25th symbol, the interpolation is done using the first two pilots. For the 26th symbol till the 39th symbol, the interpolation is done using the second and third pilots. For the 40th symbol up until the 53rd symbol, the interpolation is done using the 3rd and 4th pilots. Finally, the rest of the symbols (54th to 64th) use the last pilot along with the mean value of pilots.

5.2.5.3 Testing

The correct output from each function is read from the Gnu radio by adding a print line in the code of the block after the function that needs to be tested. Then, this output is copied as an array to the CCS, subsequently compared to the output of the CCS by subtracting the two arrays.

5.2.6 Frame Decoder

5.2.6.1 Design:

The final step in the receiver is the decoding of the actual payload. It is performed in multiple sub-steps, as follows, deinterleaving, convolutional, decoding and puncturing, depending on the coding rate we use a Viterbi decoder for decoding.

Viterbi decoder uses the Viterbi algorithm for decoding a bit stream that has been encoded using Forward error correction based on a convolution encoder shown in **Figure 1** where the following notations are used:

c = number of output bits.
x = number of input bits entering at a time.
m = number of stages of shift register.
K (constraint length) = (m + 1) digits.
R (bit rate) = x / c.

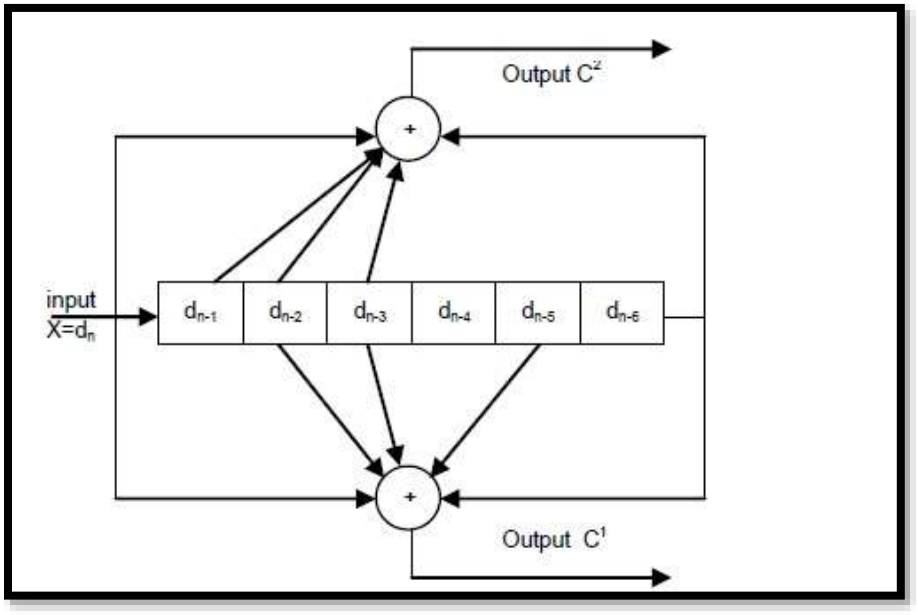

Figure 518- Convolution encoder for constraint length (k) = 7, bit rate (r) = 1/2

Viterbi decoder used to estimate the original sequence from the sequence of data received from the channel. It consists of the following functional units as shown in **Figure 2**:

- Branch Metric Unit (BMU)
- Add Compare and Select Unit (ACS)
- Survivor Memory Unit
- Trace Back Unit (TBU)

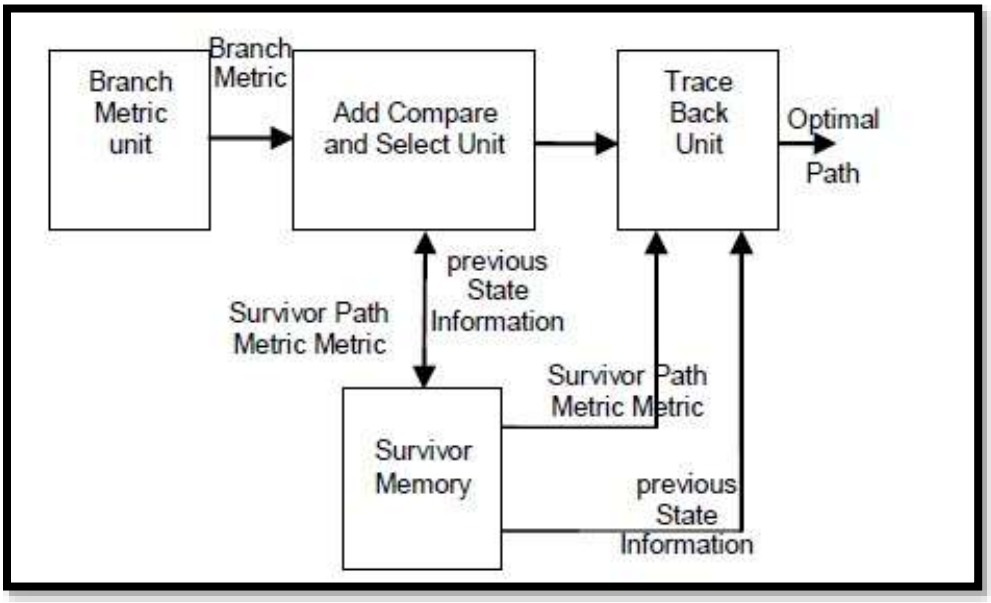

Figure 5-19 Block diagram of viterbi decoder

Basic definitions

- **State:**

The state of an encoder is defined as its shift register contents. Each new 'x' bit input results in a new state. Therefore for one bit entering the encoder there are 2 possible branches for every state. If the Constraint length k=7, then the size of shift register would be m=6 which results in 2^m states. Therefore 2^6 = 64 states are named from S0 to S63.

- **Branch metric:**

The branch metric is a measure of the "distance" between what was transmitted and what was received, and is defined for each arc in the trellis and the number on the arc shows the branch metric for that transition as shown in **Figure 3**.

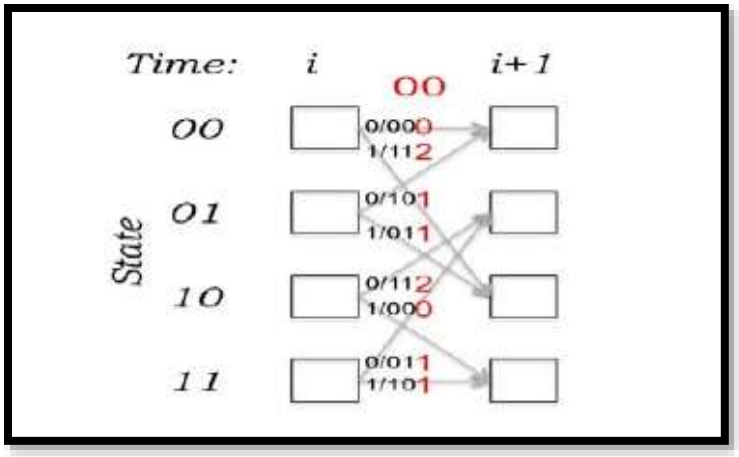

Figure 5-20 Trellis diagram for K = 3 and r = 1/2 in this example the received bits by decoder

- **Path metric:**

The path metric is a value associated with a state in the trellis. it corresponds to the Hamming *distance* with respect to the received parity bit sequence over the most likely path from the initial state to the current state in the trellis. The most likely path means the path with smallest Hamming distance between the initial state and the current state, measured over all possible paths between the two states.

The final step in the decoding process is descrambling. In the encoder the initial state of the scrambler is set to a pseudo random value.

5.3 Implementation

5.3.1 Deinterleaving

It is implemented in **interleave** function which exists in **utils.c** file. This function can operate as interleaver or deinterleaver depending on the value of the **Enum_reverse** parameter. If **Enum_reverse** = reverse function will work as deinterleaver, else it will work as interleaver.

5.3.2 Convolutional Decoding and Puncturing

5.3.2.1 Depuncture

Design

Higher rates are derived from convolutional encoder by employing "puncturing". Puncturing is a procedure for omitting some of the encoded bits in the transmitter In order to reduce the number of transmitted bits and increase encoder bit rate and in the receiver convolutional decoder side we insert dummy bits in place of the omitted bits.

Implementation

It is implemented in **depuncture** function which exists in **viterbi_decoder.c** file. Many methods can be used to perform puncturing operation, however, one of the puncture approach used in IEEE 802.11p is specified by a binary puncturing vector which consistent of two bit sequences 1110,111001 for rate 2/3, 3/4 consequently. So in the receiver side we use these two bit sequences to insert dummy "2" in place of the omitted bits as shown in **Figure 4**.

```
for (i = 0; i < d_frame->n_sym; i++) {
    for (k = 0; k < n_cbps; k++) {
        while (d_depuncture_pattern[count % (2 * d_k)] == 0) {
            depunctured[count] = 2;
            count++;
        }

        // Insert received bits
        depunctured[count] = in[i * n_cbps + k];
        count++;

        while (d_depuncture_pattern[count % (2 * d_k)] == 0) {
            depunctured[count] = 2;
            count++;
        }
    }
}
```

Figure 521-Implementation of depuncture function in code

5.3.2.2 .Viterbi decoder:

Implementation:

Branch metric unit:

Four parallel binary bits are passed to **viterbi_buterfly2_sse2** function which exists in **viterbi_decoder.c** file. This function processed over each two parallel bits at a time. It calculates sixty four set of hamming distance. Each set consists of two values because each current state can be reached by two possible paths. In order to calculate the hamming distance it compares the received codes with the expected codes of the current state by using xor bitwise operator as shown in **Figure 5**.

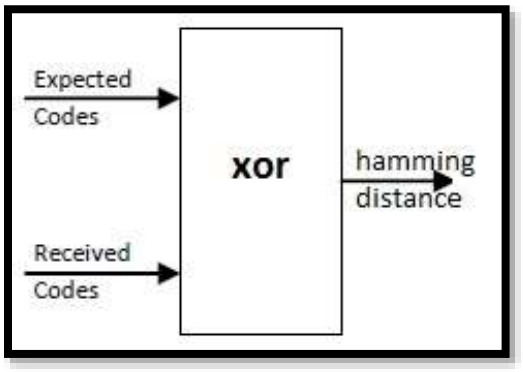

Figure 5-22 Block diagram of Branch Metric Unit

The expected codes are calculated in function called viterbi_chunks_init_sse2 which exists in **viterbi_decoder.c** file. Also this function used to reset all variables used by Viterbi decoder before starting to process on the received data bits.

At the decoder, when using a punctured code, missing parity bits don't participate in the calculation of branch metrics. Since we have replaced missing parity bits by 2 in the **depuncture** function which exists in **viterbi_decoder.c** file. So we will subtract one from calculated hamming distance if one of the processed bits is equal 2 as shown in **figure6**.

```
if (symbols[0] == 2) {
    for (j = 0; j < 16; j++) {
        metsvm[j] = d_branchtab27_sse2[1].c[(i * 16) + j] ^ sym1v[j];
        metsv[j] = 1 - metsvm[j];
    }
} else if (symbols[1] == 2) {
    for (j = 0; j < 16; j++) {
        metsvm[j] = d_branchtab27_sse2[0].c[(i * 16) + j] ^ sym0v[j];
        metsv[j] = 1 - metsvm[j];
    }
} else {
    for (j = 0; j < 16; j++) {
        metsvm[j] = (d_branchtab27_sse2[0].c[(i * 16) + j] ^ sym0v[j])
                + (d_branchtab27_sse2[1].c[(i * 16) + j] ^ sym1v[j]);
        metsv[j] = 2 - metsvm[j];
    }
}
```

Figure 5-23 Implementation of branch metric unit in code.

Add compare and select unit:

This unit is also implemented in **viterbi_buterfly2_sse2** as shown in **Figure7.** Path metric of the state is found by adding the path metric from the previous stage and the present branch metrics. Since there are two possible ways to reach any state two path metrics are obtained, these two are compared to select the one with the least path metric. The selected least path metric is sent for storage as well as it is used as benchmark for calculating the path metric of next stage as shown in **figure 8**.

```
for (j = 0; j < 16; j++) {
    m0[j] = metric0[(i * 16) + j] + metsv[j];
    m1[j] = metric0[((i + 2) * 16) + j] + metsvm[j];
    m2[j] = metric0[(i * 16) + j] + metsvm[j];
    m3[j] = metric0[((i + 2) * 16) + j] + metsv[j];
}

for (j = 0; j < 16; j++) {
    decision0[j] = ((m0[j] - m1[j]) > 0) ? 0xff : 0x0;
    decision1[j] = ((m2[j] - m3[j]) > 0) ? 0xff : 0x0;
    survivor0[j] = (decision0[j] & m0[j]) | ((~decision0[j]) & m1[j]);
    survivor1[j] = (decision1[j] & m2[j]) | ((~decision1[j]) & m3[j]);
}
```

Figure 5-24 Implementation of add compare and select unit in code

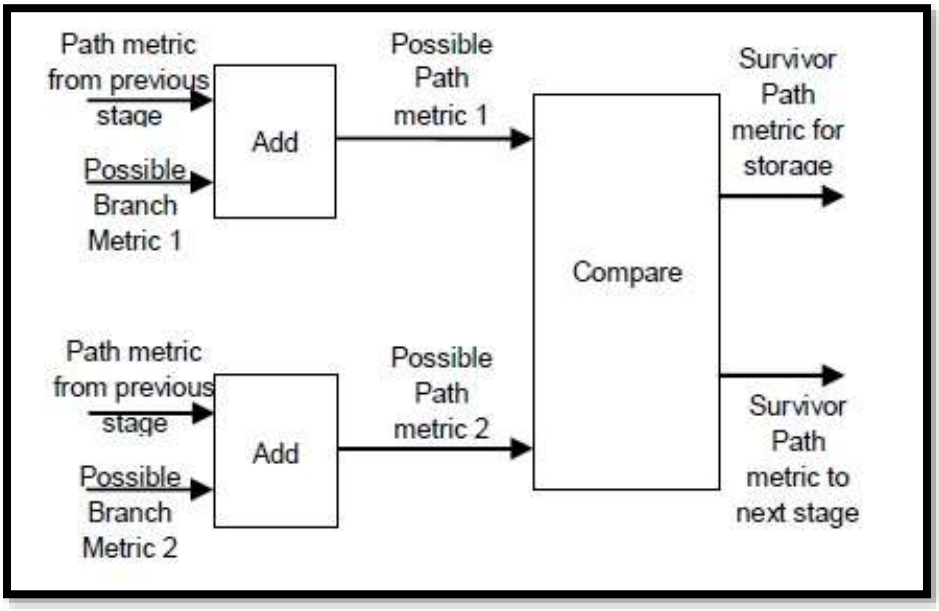

Figure 5-25 Block diagram of add compare and select unit

Survivor memory unit:

This unit is also implemented in **viterbi_buterfly2_sse2**.It is used for storing the survivor path values of the ACS unit. For each stage there are 64 survivor paths and number of stages varies depending on the length of encoded bits received.

Trace back unit:

This unit is implemented in **viterbi_get_output_sse2**. Once the minimum path metrics of all the states at each stage is calculated, the minimum path metric at the last stage is found. The state having the minimum path metrics at the last stage is given as input to Trace Back Unit and then it starts trace backing the survival paths from that node and outputs the corresponding bit which has caused the transition of that path as shown in **Figure 9**.

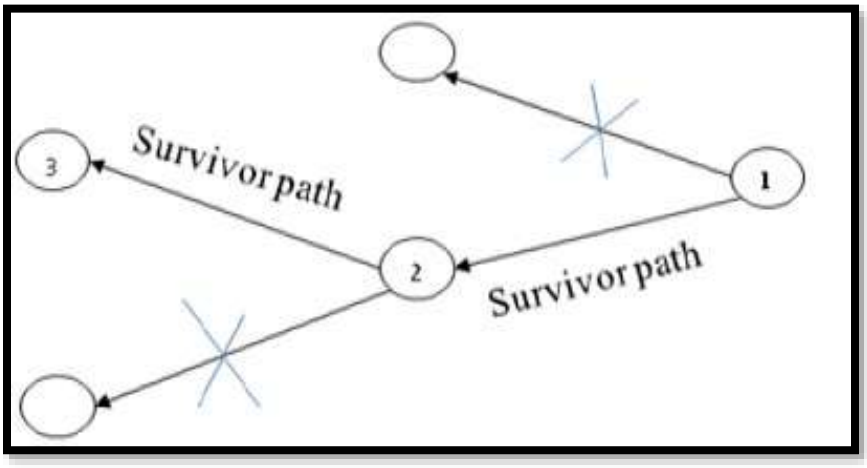

Figure 5-26 - Trace back procedure of optimal path

5.3.3 Descrambling:

It is implemented in **descramble** function which exists in **decoder_mac.c** file. This function uses first 7 bit of input data to deduce the initial state of the scrambler as shown in **Figure 10**.

```
void descramble (unsigned char *decoded_bits,unsigned char *out_bytes, frame_param *frame) {

    unsigned char state = 0;
    short int i;
    for( i = 0; i < 7; i++) {
        if(decoded_bits[i]) {
            state |= 1 << (6 - i);
        }
    }
    out_bytes[0] = state;

    unsigned char feedback;
    unsigned char bit;

    for( i = 7; i < frame->psdu_size*8+16; i++) {
        feedback = ((!!(state & 64))) ^ (!!(state & 8));
        bit = feedback ^ (decoded_bits[i] & 0x1);
        out_bytes[i/8] |= bit << (i%8);
        state = ((state << 1) & 0x7e) | feedback;
    }
}
```

Figure 5-27 Implementation of descrambling function in code

Chapter 6

Conclusion

To conclude, this report firstly discussed the need of V2V communication, V2X communication and their need in the market nowadays. No doubts that the new cars' generation is going through a massive development to self-driving cars which increases the importance of V2Vcommunication.

Our role in the project was the first step to implement this communication, the PHY layer which is the start of network creation between devices. Throughout our work, we took into consideration that the PHY layer is verified with the standard rules of IEEE802.11p. The implementation of the project after understanding all its technical aspects was coding transceiver blocks with the help of simulation tools such as GNUradio, octave..etc. Then by processing the transceiver on a DSP kit we was able to send and receive data through RF in USRP. We also stated that the PHY layer usage can have another application perspective which is testing transceiver modules. Finally, take into consideration that the next phases is very important to complete V2V as an application.

6.1 Lessons learned throughout the year

In this section, knowledge that we took from college which helped us to understand a lot of our research in the project throughout the year will be stated. Thanks to all that were reason for us to complement some of our academic knowledge with this graduation project. It was a great experience and responsibility.

Back to V2V PHY layer implementation, these were the topics that understanding it helped us a lot in the project:

a) Concept of OFDM technique
b) C programming
c) Linux usage
d) Some basic understanding of memory mapping and optimization

6.2 Future Work

There are two paths for this project as we stated at the beginning of the report. So the future work of the project will be divided into two paths which we'll discuss in this section; the first is connecting the DSP kit with C700 through FPGA and create the standalone device which can be used in testing, the second is the mac layer implementation.

All the above mentioned process in this report was only in order to implement the physical layer of the vehicle to vehicle communication, the next step is to implement the upper layers in the OSI model stated in the figure below that must be also verified with the standard of IEEE802.11 so as to implement a full optimized device at the end that could be inserted into a car to fulfill the project goal in the first place.

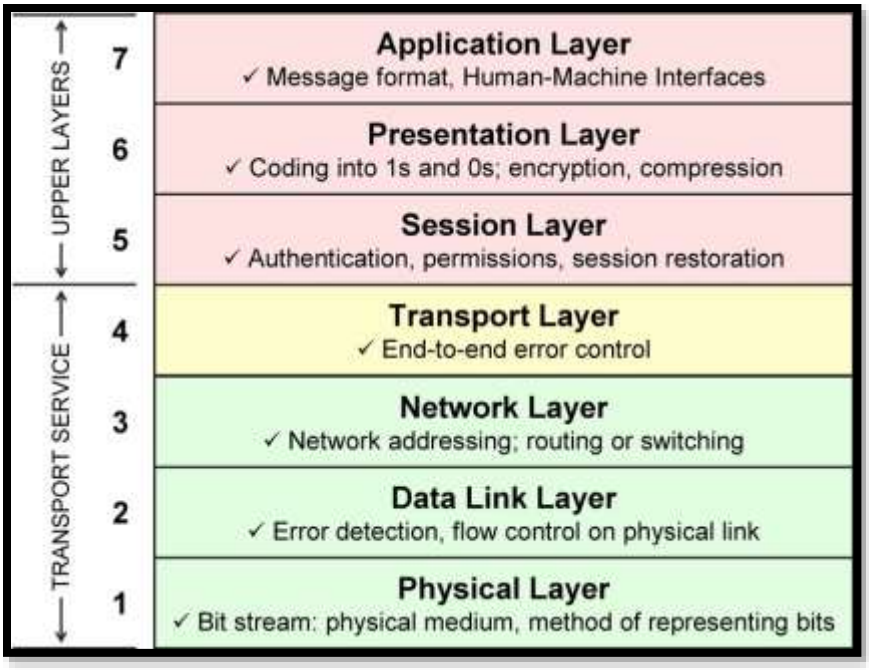

Figure 6-1 OSI model

Moreover, there are some optimizations and modifications that could be done to the hardware, that we are currently working on, this will include downloading the code on the Mitydsp kit, deriving the output to the UPP (Universal parallel port) and connecting the kit to C700, an alternative for the USRP that possess the same functionality and could be connected to the FPGA on the kit which couldn't be done with the USRP.

References

1- 14,500 road accidents in 2015, 63.3% attributed to human error: CAPMAS - Daily News Egypt. (2016). Daily News Egypt. Retrieved 2 January 2017, from http://www.dailynewsegypt.com/2016/05/08/14500-road-accidents-2015-63-3-attributed-human-error-capmas/ Connectivity. (2017). Renesas Electronics America. Retrieved 9 Jan 2017, from https://www.renesas.com/en-us/solutions/automotive/adas/v2x.html

2- Vehicle To Vehicle Communication Market Intelligence Report Offers Growth Prospects. (2016). www.linkedin.com. Retrieved 9 June 2017, from https://www.linkedin.com/pulse/vehicle-communication-market-intelligence-report-offers-musare

3- الجهاز المركزي للتعبئة العامة والإحصاء. (2017). Capmas.gov.eg. Retrieved 9 June 2017, from http://www.capmas.gov.eg/Pages/IndicatorsPage.aspx?page_id=6140&ind_id=2305

4- IEEE Standards Assosiation, IEEE Standard 802.11, IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks—Specific requirements, 2012

5- Bloessel, B., Segata, M., Sommer, C., & Dressler, F. (2013). An IEEE 802.11a/g/p OFDM Receiver for GNU Radio (1st ed.). Retrieved from http://conferences.sigcomm.org/sigcomm/2013/papers/srif/p9.pdf

6- GNU Radio. (2017). Wiki.gnuradio.org. Retrieved 9 June 2017, from https://wiki.gnuradio.org/index.php/Main_Page#Which-license-does-GNU-Radio-use

7- Home page - GNU Radio. (2017). GNU Radio. Retrieved 9 June 2017, from https://www.gnuradio.org/

8- bastibl/gr-ieee802-11. (2017). GitHub. Retrieved 9 June 2017, from https://github.com/bastibl/gr-ieee802-11

9- CCSTUDIO Code Composer Studio (CCS) Integrated Development Environment (IDE) | TI.com. (2017). Ti.com. Retrieved 9 June 2017, from http://www.ti.com/tool/ccstudio

10- GNU Octave Wiki - Octave. (2017). Wiki.octave.org. Retrieved 9 June 2017, from http://wiki.octave.org/GNU_Octave_Wiki

11- What Is NI USRP Hardware? - National Instruments. (2017). Ni.com. Retrieved 9 June 2017, from http://www.ni.com/white-paper/12985/en/

12- TMDSEMU100V2U-ARM XDS100v2 JTAG Debug Probe (ARM version) | TI.com. (2017). Ti.com. Retrieved 9 June 2017, from http://www.ti.com/tool/tmdsemu100v2u-arm

13- MityDSP-L138F - Critical Link. (2017). Critical Link. Retrieved 9 June 2017, from http://www.criticallink.com/product/mitydsp-l138f/

14- MityDSP-L138(F) Family Development Kit - Critical Link. (2017). Critical Link. Retrieved 9 June 2017, from http://www.criticallink.com/product/mitydsp-l138f-dev-kit/

15- Agilent N4010A WLAN Help. (2017). Rfmw.em.keysight.com. Retrieved 9 June 2017, from http://rfmw.em.keysight.com/rfcomms/n4010a/n4010aWLAN/onlineguide/default.htm#ofdm_raised_cosine_w

16- Blossel, B., Gerla, M., & Dressler, F. (2016). IEEE 802.11p in Fast Fading Scenarios: From Traces to Comparative Studies of Receive Algorithms (1st ed.). Retrieved from http://www.ccs-labs.org/bib/bloessl2016ieee/bloessl2016ieee.pdf

17- Sandesh, & Rambabu, K. (2013). Implementation of Convolution Encoder and Viterbi Decoder for Constraint Length 7 and Bit Rate 1/2 (1st ed.). ijera.com. Retrieved from https://pdfs.semanticscholar.org/bd87/b8d2b616c726c7987b682e166e57b68c707c.pdf

18- Viterbi Decoding of Convolutional Codes. (2012) (1st ed.). Retrieved from https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-02-introduction-to-eecs-ii-digital-communication-systems-fall-2012/readings/MIT6_02F12_chap08.pdf

Appendix A

Installation Guide

A.1 Installation Guide for Code Composer Studio

The installation steps for CCS v7 and v5 are almost the same.

For Windows

1- Open http://processors.wiki.ti.com/index.php/Download_CCS
2- Choose your operating system (Windows, Linux…etc)
3- Open the executable file that was downloaded
4- Mark "I accept the terms of the license agreement" and click next
5- Select your intsallation folder and then click next
6- Select the following processors (OMAP-L1x DSP + ARM9 processor, C6000 Power optimized DSP, C64x multicore DSP)
7- For the debug probes, select TI XDS Debug Probe Support and Spectrum Digital Debug Probes and Boards (it's better to select all if you're not sure about the debug probe you'll use yet)
8- Finally, select finish

For Ubuntu

Before the installation, make sure to install some dependencies. Open the command window and write this command

- sudo apt-get update

- sudo apt-get install libc6:i386 libx11-6:i386 libasound2:i386 libatk1.0-0:i386 libcairo2:i386 libcups2:i386 libdbus-glib-1-2:i386 libgconf-2-4:i386 libgdk-pixbuf2.0-0:i386 libgtk-3-0:i386 libice6:i386 libncurses5:i386 libsm6:i386 liborbit2:i386 libudev1:i386 libusb-0.1-4:i386 libstdc++6:i386 libxt6:i386 libxtst6:i386 libgnomeui-0:i386 libusb-1.0-0-dev:i386 libcanberra-gtk-module:i386 gtk2-engines-murrine:i386 unzip

After installing the dependencies, the same steps are applied. The only difference is that the executable file's format is .bin

To open this file, open the command window and go to the location where the downloaded file exists then write

sudo ./"file name".bin

The same steps are then applied.

A.1.1 Installing the DSP library

1- Open http://www.ti.com/tool/sprc265
2- Download C674x-DSPLIB
3- Click on the link suitable to your operating system (Windows 64- Windows 32- Ubuntu…etc)

For Windows

4- Click on the executable file and select your language
5- Select the installation folder of CCS (c:/ti/DSPlib folder)
6- Finally, select next and agree on the terms and conditions

For Linux

Same steps as Windows. To open the executable file, open the command window and run the .bin file

A.1.2 Make a new project on CCS

1- Open file -> new -> CCS project
2- Select the family c6000 for DSP based projects and ARM for ARM based projects
3- In Variant, select OMAPL138
4- Write the project name and click on Finish
5- Right click on the project -> new -> Target configuration file
6- In case of simulator, choose Texas Instruments Simulator, then choose C674x CPU Cyclic Accurate Simulator, Little Endian for DSP project or ARM9e CPU Cyclic Accurate Simulator, Little Endian for ARM based project
7- In case of emulator, choose your emulator (XDS 100v2 USB) then choose LCDKOMAPL138

A.1.3 Including the DSP library in the project

1- Right click on the project and select properties
2- Open compiler -> include options
3- In the "Add dir to #include search path" field -> add the path of your dsplib/packages
4- Open linker -> File search path
5- In the "Include library file or command file as input" field, add these two lines
   "dsplib.lib"
   "dsplib_cn.lib"
6- In the "Add dir to library search path" field, add the folder path of these 2 libraries, you will find it in the DSP library folder /packages/ti/dsplib/lib
7- Now you can use the DSPlib functions in your code.

A.2 USRP hardware driver Installation guide

A.2.1 Installation Requirements

- linux OS (ubuntu)

A.2.2 Installation guide

1) Open terminal window
2) Write these commands:
   - sudo apt -get install libuhd –dev libuhd uhd -host
   - sudo add-apt-repository ppa:ettusresearch/uhd
   - sudo apt-get update
   - sudo apt-get install libuhd-dev libuhd003 uhd-host

A.3 GNU Radio Installation guide

A.3.1 Installation Requirements

- linux OS
- GNU radio program
- IEEE 802.11 standerd blocks

A.3.2 Installation guide

3) Open terminal window
4) Write these commands:
   - (sudo apt-get update) then enter the username and password
   - wget http://www.sbrac.org/files/build-gnuradio && chmod a+x build-gnuradio && ./build-gnuradio
5) To open the gnu radio for the first time we need to open it through the terminal ,so we write (gnuradio-companion)
6) To get the blocks Write these commands:
   - sudo apt-get install liblog4cpp5-dev
   - sudo port install log4cpp
   - git clone https://github.com/bastibl/gr-foo.git
   - cd gr-foo
   - mkdir build
   - cd build
   - cmake ..
   - make

- sudo make install
- sudo ldconfig
- git clone git://github.com/bastibl/gr-ieee802-11.git
- cd gr-ieee802-11
- mkdir build
- cd build
- cmake ..
- make
- sudo make install
- sudo ldconfig
- sudo sysctl -w kernel.shmmax=2147483648

A.4 Octave Installation guide

A.4.1 Installation Requirements

- linux OS

A.4.2 Installation guide

7) Open terminal window
8) Write these commands:
   - sudo apt-add repository ppa:octave/stable
   - sudo apt -get update
   - sudo apt -get install octave
   - octave

A.4.3 Usage guide

After opening the program write these commands to open the files:

- PS1(">>")
- addpath("/home/username/gnuradio/gr-utils/octave")

Appendix B

CCS Code

B.1 Transmitter

B.1.1 Main function

```
/*
 * main.c
 */
#include  "IEEE802_11_Common_Variables.h"
#include  "utils.h"
#include "Mapper.h"
#include "signal_field_impl.h"
#include "constellations_impl.h"
#include "chunks_to_symbols_impl.h"
#include "ifft.h"
#include "CyclicPrefix.h"
#include  "ofdm_carr_alloc_func.h"

#define N 64
/* The length of the message received from the Mac layer */
#define psdu_length  100
#define signal_field_size 48
void main(void) {
    FILE *fp;
    /* This is where the main function will be called */
    Encoding e = QAM16_3_4;
    /* constructing an instant of the frame and the ofdm parameters */
    // This is the message : PSDU  generated by the mac-layer
    uint8 d_psdu[100] = { 4, 2, 0, 46, 0, 96, 8, 205, 55, 166, 0, 32, 214, 1,
                60, 241, 0, 96, 8, 173, 59, 175, 0, 0, 74, 111, 121, 44, 32,
98,
                114, 105, 103, 104, 116, 32, 115, 112, 97, 114, 107, 32,
111, 102,
                32, 100, 105, 118, 105, 110, 105, 116, 121, 44, 10, 68, 97,
117,
                103, 104, 116, 101, 114, 32, 111, 102, 32, 69, 108, 121,
115, 105,
                117, 109, 44, 10, 70, 105, 114, 101, 45, 105, 110, 115, 105,
114,
                101, 100, 32, 119, 101, 32, 116, 114, 101, 97, 103, 51, 33,
182 };
    float32 window[2 * N];
    int sizeof_input_sym;
    int test;
    int data_size;
    int i = 0;
    int loop = 1408 / (2 * N);
    ofdm_param* data_field_ofdm;
    frame_param* data_field_frame;
    ofdm_param* signal_field_ofdm;
    frame_param* signal_field_param;
    uint8 *out_processed_signal_field;
    uint8 * Output_Processed_Data;
    float32 *out_modulated_data;
```

```
	float32 *out_modulated_signal_field;
	float32 *out_tagged_mux;
	float32* Output_From_OFDMCarrierAllocator;
	float32* Output_From_IFFT;
	float32* Output_From_CyclicPrefix;
	StructCyclicPrefix_Init *CyclicPtr;
	while (1) {
		data_field_ofdm = (ofdm_param *) malloc(sizeof(ofdm_param));
		ofdm_param_intialization(e, data_field_ofdm);
		data_field_frame = (frame_param *) malloc(sizeof(frame_param));
		frame_param_intialization(data_field_ofdm, data_field_frame,
				psdu_length);
		/* Check the value of the frame_param */
		print_frame_param(data_field_frame);

		Output_Processed_Data = mapper_general_work_function(d_psdu,
				psdu_length, data_field_ofdm, data_field_frame);
		  /*generating the signal field and creating frame and ofdm
paramters*/
		signal_field_ofdm = (ofdm_param *) malloc(sizeof(ofdm_param));
		ofdm_param_intialization(BPSK_1_2, signal_field_ofdm);
		signal_field_param = (frame_param *) malloc(sizeof(frame_param));
		frame_param_intialization(signal_field_ofdm, signal_field_param,
0);
		out_processed_signal_field =
generate_signal_field(signal_field_param,
				signal_field_ofdm, data_field_frame,
data_field_ofdm);
		//Data modulation
		data_size = data_field_frame->n_sym * 48;
		out_modulated_data = malloc(data_size * 2 * sizeof(float32));
		chunks_to_symbols_impl(Output_Processed_Data, out_modulated_data,
				data_size, e);
		//Signal field modulation
		out_modulated_signal_field = malloc(
				signal_field_size * 2 * sizeof(float32));
		chunks_to_symbols_impl(out_processed_signal_field,
				out_modulated_signal_field, signal_field_size,
BPSK_1_2);
		//Tagged_stream_MUX
		out_tagged_mux = malloc(
				(data_size + signal_field_size) * 2 *
sizeof(float32));
		memcpy(out_tagged_mux, out_modulated_signal_field,
				signal_field_size * 2 * sizeof(float32));
		memcpy(out_tagged_mux + (signal_field_size * 2),
out_modulated_data,
				data_size * 2 * sizeof(float32));
		// part OFDM carrier allocater
		sizeof_input_sym = (2 * signal_field_size) + (2 * data_size);
		Output_From_OFDMCarrierAllocator = (float32 *) malloc(
				1408 * sizeof(float32));
		if (Output_From_OFDMCarrierAllocator == NULL) {
		printf("Not enough memory for Output_From_OFDMCarrierAllocator
\n");
```

```
            }
            test = ofdm_carr_alloc(occupied_carriers, pilot_carriers,
pilot_symbols,
                        sync_words, fft_len, output_is_shifted,
out_tagged_mux,
                        Output_From_OFDMCarrierAllocator, sizeof_input_sym);
            /*-----------------IFFT---------------------*/
            Output_From_IFFT = (float32 *) malloc(1408 * sizeof(float32));
            for (i = 0; i < loop; i++) {
                  ifft(Output_From_OFDMCarrierAllocator, Output_From_IFFT,
N, 52.0,
                              true, window);
                  Output_From_OFDMCarrierAllocator += (2 * N);
                  Output_From_IFFT += (2 * N);
            }
            Output_From_IFFT -= (loop * 2 * N);
            /*---------------------- Part cyclic prefix----------------------
---**/
            Output_From_CyclicPrefix = (float32*) malloc(
                        (1408 / 64) * 80 * 2 * sizeof(float32));
            CyclicPtr = (StructCyclicPrefix_Init *) malloc(
                        sizeof(StructCyclicPrefix_Init));
            ;
            CyclicPrefix_Init(CyclicPtr, 1408 / 64);
            CyclicPrefix(CyclicPtr, Output_From_IFFT,
Output_From_CyclicPrefix);
            /*---------------------- test cyclic prefix--------------------
--**/
            // Writing to a file
            if ((fp = fopen("Test2.txt", "a+")) == NULL) {
                  printf("Cannot open file.\n");
            }
            fseek(fp, 0, SEEK_END);
            if (fwrite(Output_From_CyclicPrefix, sizeof(float32), (1408 / 64)
* 80,
                        fp) != (1408 / 64) * 80)
                  printf("File read error.");
            fflush(fp);
            fclose(fp);
            free(Output_From_CyclicPrefix);
            free(Output_From_IFFT);
            free(Output_From_OFDMCarrierAllocator);
            free(out_tagged_mux);
            free(out_modulated_signal_field);
            free(out_modulated_data);
            free(out_processed_signal_field);
            free(signal_field_param);
            free(signal_field_ofdm);
            free(Output_Processed_Data);
            free(data_field_frame);
            free(data_field_ofdm);
      }
   }
```

B.1.2 Generic files used by more than one block

B.1.2.1 utils.h file

```
/*
 * utils.h
 *
 *  Created on: Feb 7, 2017
 *      Author: Salma Khaled
 */
#ifndef UTILS_H_
#define UTILS_H_
#include "IEEE802_11_Common_Variables.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#define MAX_PAYLOAD_SIZE 1500
#define MAX_PSDU_SIZE (MAX_PAYLOAD_SIZE + 28) // MAC, CRC
#define MAX_SYM (((16 + 8 * MAX_PSDU_SIZE + 6) / 24) + 1)
#define MAX_ENCODED_BITS ((16 + 8 * MAX_PSDU_SIZE + 6) * 2 + 288)
/**--------------------------------ofdm_param_implementation---------------
------**/
typedef enum {
   BPSK_1_2 = 0,
   BPSK_3_4 = 1,
   QPSK_1_2 = 2,
   QPSK_3_4 = 3,
   QAM16_1_2 = 4,
   QAM16_3_4 = 5,
   QAM64_2_3 = 6,
   QAM64_3_4 = 7,
} Encoding;
typedef struct {
   // data rate
   Encoding encoding;
   // rate field of the SIGNAL header
   char rate_field;
   // number of coded bits per sub carrier
   int n_bpsc;
   // number of coded bits per OFDM symbol
   int n_cbps;
   // number of data bits per OFDM symbol
   int n_dbps;

} ofdm_param;
/** This function is used to initialize the parameters of the ofdm */
void ofdm_param_intialization(Encoding, ofdm_param*);
/** This function print the values of the ofdm param */
void print_ofdm_param(const ofdm_param*);
```

```
   /**---------------------------frame_param_implementation-------------------
-------**/
   typedef struct {
      // PSDU size in bytes
      int psdu_size;
      // number of OFDM symbols (17-11)
      int n_sym;
      // number of padding bits in the DATA field (17-13)
      int n_pad;
      int n_encoded_bits;
      // number of data bits, including service and padding (17-12)
      int n_data_bits;
   } frame_param;
   /** This function is used to initialize the parameters of the ofdm */
   void frame_param_intialization(ofdm_param*, frame_param*, int);
   /** This function is used to print the value of the frame param */
   void print_frame_param(const frame_param*);
   /*---------------------------------PSDU_Processing-------------------------
-------**/
   void scramble(const uint8 *input, char unsigned *out,
            frame_param* frame, uint8 initial_state);
   void reset_tail_bits(uint8 *scrambled_data, frame_param* frame);
   void convolutional_encoding(const uint8 *input, char unsigned *out,
            frame_param* frame);
   void puncturing(const uint8 *input, uint8 *out,
            frame_param* frame, ofdm_param* ofdm);
   void interleave(const uint8 *input, uint8 *out,
            frame_param* frame, ofdm_param* ofdm);
   void split_symbols(const uint8 *input, uint8 *out,
            frame_param* frame, ofdm_param* ofdm);
   void generate_bits(const uint8 *psdu, uint8 *data_bits,
            frame_param* frame);
   #endif /* UTILS_H_ */
```

B.1.2.2 utils.c file:

```
   /*
    * utils.c
    *
    *  Created on: Feb 7, 2017
    *      Author: Salma Khaled
    */
   #include "utils.h"
   /**---------------------------ofdm_param_implementation-----------------
----**/
   void ofdm_param_intialization(Encoding e, ofdm_param* ofdm) {
      ofdm->encoding = e;
      switch (e) {
      case BPSK_1_2:
            ofdm->n_bpsc = 1;
            ofdm->n_cbps = 48;
            ofdm->n_dbps = 24;
            ofdm->rate_field = 0x0D; // 0b00001101
            break;
```

```
    case BPSK_3_4:
          ofdm->n_bpsc = 1;
          ofdm->n_cbps = 48;
          ofdm->n_dbps = 36;
          ofdm->rate_field = 0x0F; // 0b00001111
          break;
    case QPSK_1_2:
          ofdm->n_bpsc = 2;
          ofdm->n_cbps = 96;
          ofdm->n_dbps = 48;
          ofdm->rate_field = 0x05; // 0b00000101
          break;
    case QPSK_3_4:
          ofdm->n_bpsc = 2;
          ofdm->n_cbps = 96;
          ofdm->n_dbps = 72;
          ofdm->rate_field = 0x07; // 0b00000111
          break;
    case QAM16_1_2:
          ofdm->n_bpsc = 4;
          ofdm->n_cbps = 192;
          ofdm->n_dbps = 96;
          ofdm->rate_field = 0x09; // 0b00001001
          break;
    case QAM16_3_4:
          ofdm->n_bpsc = 4;
          ofdm->n_cbps = 192;
          ofdm->n_dbps = 144;
          ofdm->rate_field = 0x0B; // 0b00001011
          break;
    case QAM64_2_3:
          ofdm->n_bpsc = 6;
          ofdm->n_cbps = 288;
          ofdm->n_dbps = 192;
          ofdm->rate_field = 0x01; // 0b00000001
          break;
    case QAM64_3_4:
          ofdm->n_bpsc = 6;
          ofdm->n_cbps = 288;
          ofdm->n_dbps = 216;
          ofdm->rate_field = 0x03; // 0b00000011
          break;
    default:
          assert(false);
          break;
    }
}
/** a function to print the values of the ofdm_param */
void print_ofdm_param(const ofdm_param* ofdm) {
    printf("OFDM Parameters:   \n");
    printf("encoding       : %i\n", ofdm->encoding);
    printf("rate_field     : %i\n", ofdm->rate_field);
    printf("n_bpsc         : %i\n", ofdm->n_bpsc);
    printf("n_cbps         : %i\n", ofdm->n_cbps);
    printf("n_dbps         : %i\n", ofdm->n_dbps);
```

```
    }

    /**----------------------------frame_param_implementation-----------------
---**/
    void frame_param_intialization(ofdm_param* ofdm, frame_param* frame,
              int psdu_length) {
        frame->psdu_size = psdu_length;
        // number of symbols (17-11)
        frame->n_sym = (int) ceil(
                    (16 + 8 * (frame->psdu_size) + 6) / (double) ofdm-
>n_dbps);
        frame->n_data_bits = (frame->n_sym) * (ofdm->n_dbps);
        // number of padding bits (17-13)
        frame->n_pad = (frame->n_data_bits) - (16 + 8 * (frame->psdu_size) + 6);
        frame->n_encoded_bits = (frame->n_sym) * (ofdm->n_cbps);
    }
    /** a function to print the values of the frame_param */
    void print_frame_param(const frame_param* frame) {
        printf("FRAME Parameters :\n");
        printf("psdu_size          :%i\n", frame->psdu_size);
        printf("n_sym                    :%i\n", frame->n_sym);
        printf("n_pad                    :%i\n", frame->n_pad);
        printf("n_encoded_bits   :%i\n", frame->n_encoded_bits);
        printf("n_data_bits        :%i\n", frame->n_data_bits);
    }
    /**-------------------------------PSDU_Processing-------------------------
-----**/
    //1-Generate_bits:
    void generate_bits(const uint8 *psdu, uint8 *data_bits,
              frame_param* frame) {

        //printf(" This is the generate bits" );
        // first 16 bits are zero (SERVICE/DATA field)
        memset(data_bits, 0, 16);
        data_bits += 16;
        int i;
        int b;
        for (i = 0; i < frame->psdu_size; i++) {
              for (b = 0; b < 8; b++) {
                    data_bits[i * 8 + b] = !!(psdu[i] & (1 << b));
              }
        }
    }
    //2-Scrambling the data
    void scramble(const uint8 *input, uint8 *out,
              frame_param* frame, uint8 initial_state) {
        //printf("This is the scrambler \n");
        int state = initial_state;
        int feedback;
        int i;
        for (i = 0; i < frame->n_data_bits; i++) {
              feedback = (!!(state & 64)) ^ (!!(state & 8));
              out[i] = feedback ^ input[i];
              //printf("%i",out[i]);
              state = ((state << 1) & 0x7e) | feedback;
```

```
    }
}
//3-Reseting tail bits
void reset_tail_bits(uint8 *scrambled_data, frame_param* frame) {
    memset(scrambled_data + frame->n_data_bits - frame->n_pad - 6, 0,
                6 * sizeof(char));
}
//4-Convolutional encoding
int ones(int n) {
    int sum = 0;
    int i;
    for (i = 0; i < 8; i++) {
          if (n & (1 << i)) {
                sum++;
          }
    }
    return sum;
}
void convolutional_encoding(const uint8 *input, uint8 *out,
          frame_param* frame) {
    //printf(" This is the Convolutional encoder");
    int state = 0;
    int i;
    for (i = 0; i < frame->n_data_bits; i++) {
          assert(input[i] == 0 || input[i] == 1);
          state = ((state << 1) & 0x7e) | input[i];
          out[i * 2] = ones(state & 0155) % 2;
          out[i * 2 + 1] = ones(state & 0117) % 2;
          //printf("%i",out[i]);
    }
}
//5- Puncturing the data
void puncturing(const uint8 *input, uint8 *out,
          frame_param* frame, ofdm_param* ofdm) {
    int mod;
    int i;
    for (i = 0; i < frame->n_data_bits * 2; i++) {
          switch (ofdm->encoding) {
          case BPSK_1_2:
          case QPSK_1_2:
          case QAM16_1_2:
                *out = input[i];
                out++;
                break;
          case QAM64_2_3:
                if (i % 4 != 3) {
                      *out = input[i];
                      out++;
                }
                break;
          case BPSK_3_4:
          case QPSK_3_4:
          case QAM16_3_4:
          case QAM64_3_4:
                mod = i % 6;
```

```
                    if (!(mod == 3 || mod == 4)) {
                        *out = input[i];
                        out++;
                    }
                    break;
                default:
                    assert(false);
                    break;
                }
            }
        }
        //6-Interleaving data
        void interleave(const uint8 *input, uint8 *out,
                frame_param* frame, ofdm_param* ofdm) {
            int n_cbps = ofdm->n_cbps;
            ptoi first = (ptoi) calloc(n_cbps, sizeof(int));
            if (first == NULL) {
                printf("Not enough memory for first in the interleaver \n");
            }
            ptoi second = (ptoi) calloc(n_cbps, sizeof(int));
            if (second == NULL) {
                printf("Not enough memory for second in the interleaver \n");
            }
            int s = max(ofdm->n_bpsc / 2, 1);
            int j;
            for (j = 0; j < n_cbps; j++) {
                first[j] = s * (j / s) + ((j + (int) (floor(16.0 * j / n_cbps)))
% s);
            }
            int i;
            for (i = 0; i < n_cbps; i++) {
                second[i] = 16 * i - (n_cbps - 1) * (int) (floor(16.0 * i /
n_cbps));
            }
            int k;
            for (i = 0; i < frame->n_sym; i++) {
                for (k = 0; k < n_cbps; k++) {

                    out[i * n_cbps + k] = input[i * n_cbps +
second[first[k]]];
                    //printf("%i", out[i * n_cbps + k]);
                }
            }
            free(second);
            free(first);
        }
        //7-splitting the symbols according to the modulation type: BPSK, QPSK,
QAM16, ...
        void split_symbols(const uint8 *input, uint8 *out,
                frame_param* frame, ofdm_param* ofdm) {
            //printf(" This is the split symbols ");
            int symbols = frame->n_sym * 48;
            int i;
            int k;
            for (i = 0; i < symbols; i++) {
```

```
            out[i] = 0;
            for (k = 0; k < ofdm->n_bpsc; k++) {
                  assert(*input == 1 || *input == 0);
                  out[i] |= (*input << k);
                  input++;
                  //printf("%c",out[i]);
            }
      }
}
```

B.1.2.3 IEEE802_11_Common_Variables.h

```
/*
 * IEEE802_11_Common_Variables.h
 *
 *  Created on: Feb 7, 2017
 *      Author: Salma Khaled
 */
#ifndef IEEE802_11_COMMON_VARIABLES_H_
#define IEEE802_11_COMMON_VARIABLES_H_
typedef unsigned char uint8;
typedef signed char sint8;
typedef unsigned short uint16;
typedef signed short sint16;
typedef unsigned long uint32;
typedef signed long sint32;
typedef unsigned long longuint64;
typedef signed long longsint64;
typedef float float32;
typedef double float64;
typedef int* ptoi;
typedef int bool;
#define true              1
#define false             0
#define max(a,b) \
   ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a > _b ? _a : _b; })
#define min(a,b) \
   ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a < _b ? _a : _b; })
#endif /* IEEE802_11_COMMON_VARIABLES_H_ */
```

B.1.3 Code of the Mapper block

B.1.3.1 Mapper.h file

```
/*
 * Mapper.h
 *
 *  Created on: Feb 7, 2017
 *      Author: Salma Khaled
```

```
 */
#ifndef MAPPER_H_
#define MAPPER_H_
#ifndef MAPPER_IMP_H_
#define MAPPER_IMP_H_
#include  "IEEE802_11_Common_Variables.h"
#include "utils.h"
/* The Number of data carriers */
#define Data_Carriers 48
uint8 * mapper_general_work_function(const uint8* psdu,
          int psdu_length, ofdm_param* d_ofdm, frame_param * frame);
#endif /* MAPPER_IMP_H_ */
#endif /* MAPPER_H_ */
```

B.1.3.2 Mapper.c file

```
/*
 * Mapper.c
 *
 *  Created on: Feb 9, 2017
 *      Author: Salma Khaled
 */
#include "Mapper.h"
#include <inttypes.h>
// This is the general work function of the Mapper that is used to call the
function that will do all the PSDU processing
uint8* mapper_general_work_function(const uint8* psdu,
          int psdu_length, ofdm_param* d_ofdm, frame_param * frame) {
      char * d_symbols;
      int d_symbols_offset = 0;
      int d_symbols_len = 0;
      // calculate the length of the processed data
      d_symbols_len = frame->n_sym * 48;
      int i = d_symbols_len - d_symbols_offset;
      // Final output array
      uint8 * out = (uint8*) calloc(i, sizeof(char));
      // This is the final output from the Mapper without   offset
      d_symbols = (char*) calloc(d_symbols_len, 1);
      printf("MAPPER called offset: %i\n", d_symbols_offset);
      printf("length: %i\n", d_symbols_len);
      while (!d_symbols_offset) {
            printf("MAPPER: received new message \n");
            if (frame->n_sym > MAX_SYM) {

                  printf("packet too large, maximum number of symbols is
%i\n ",
                              MAX_SYM);
                  return 0;
            }
            //allocate memory for modulation steps
            uint8 *data_bits = (uint8*) calloc(frame->n_data_bits,
                        sizeof(uint8));
            if (data_bits == NULL) {
                  printf("Not enough memory for data_bits \n");
```

```
}
uint8 *scrambled_data = (uint8*) calloc(
            frame->n_data_bits, sizeof(uint8));
if (scrambled_data == NULL) {
      printf("Not enough memory for scrambled_data\n");
}
uint8 *encoded_data = (uint8*) calloc(
            frame->n_data_bits * 2, sizeof(uint8));
if (encoded_data == NULL) {
      printf("Not enough memory for encoded_data \n");
}
uint8 *punctured_data = (uint8*) calloc(
            frame->n_encoded_bits, sizeof(uint8));
if (punctured_data == NULL) {
      printf("Not enough memory for punctured_data \n");
}
uint8 *interleaved_data = (uint8*) calloc(
            frame->n_encoded_bits, sizeof(uint8));
if (interleaved_data == NULL) {
      printf("Not enough memory for interleaved_data \n");
}
uint8 *symbols = (uint8*) calloc(
            (frame->n_encoded_bits / d_ofdm->n_bpsc),
            sizeof(uint8));
if (symbols == NULL) {
      printf("Not enough memory for symbols \n");
}
//generate the WIFI data field, adding service field and pad bits
generate_bits(psdu, data_bits, frame);
// scrambling
// Initial state of the scrambler is set  to : 93
static uint8_t scrambler = 93;
scramble(data_bits, scrambled_data, frame, scrambler);
if (scrambler > 127) {
      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);
// interleaving
interleave(punctured_data, interleaved_data, frame, d_ofdm);
// one byte per symbol
split_symbols(interleaved_data, symbols, frame, d_ofdm);
memcpy(d_symbols, symbols, d_symbols_len);
// free the allocated memory
free(symbols);
free(interleaved_data);
free(punctured_data);
free(encoded_data);
free(scrambled_data);
free(data_bits);
break;
```

```
	}
	// if there was an offset copy it to the out data
	memcpy(out, d_symbols + d_symbols_offset, i);
	d_symbols_offset += i;
	if (d_symbols_offset == d_symbols_len) {
		d_symbols_offset = 0;
		free(d_symbols);
		d_symbols = 0;
	}
	return out;

}
/* Function to print the output data from the mapper can be used after the
split symbols */
void print_Output_bits(char* output, frame_param* frame, ofdm_param*
d_ofdm) {
	int i;
	int k;
	int symbols_length = frame->n_sym * 48;
	for (i = 0; i < symbols_length; i++) {
		for (k = 0; k < d_ofdm->n_bpsc; k++) {
			printf("%i", output[i]);
		}
	}
}
```

B.1.4 Code of the Packet header generater block

B.1.4.1 signal_field_impl.h

```
/*
 * signal_field_impl.h
 *
 *  Created on: Feb 21, 2017
 *      Author: Dina Mohamed
 */
#ifndef SIGNAL_FIELD_IMPL_H_
#define SIGNAL_FIELD_IMPL_H_
#include  "IEEE802_11_Common_Variables.h"
#include "utils.h"
uint8 * generate_signal_field(frame_param* signal_param,
		ofdm_param* signal_ofdm, frame_param* data_frame,
		ofdm_param* data_ofdm);
int get_bit(int b, int i);
#endif /* SIGNAL_FIELD_IMPL_H_ */
```

B.1.4.2 signal_field_impl.c

```
/*signal_field_impl.c
 *  Created on: Feb 21, 2017
 *      Author: Dina Mohamed
```

```
 */
#include "signal_field_impl.h"
#include  "IEEE802_11_Common_Variables.h"
// This function returns the ith bit in the int b variable
int get_bit(int b, int i) {
   return (b & (1 << i) ? 1 : 0);
}
/** This is the general work function of the Packet header generator block
that is used to produce the signal field**/
uint8 * generate_signal_field(frame_param* signal_param,
          ofdm_param* signal_ofdm, frame_param* data_frame, ofdm_param*
data_ofdm) {
      //output frame of 48 bits (24*2) 0->47
      uint8 * out = (uint8 *) malloc(sizeof(uint8) * 48);
      //data bits of the signal header
      uint8 *signal_header = (uint8 *) malloc(
                  sizeof(uint8) * 24);
      //convolutional encoding
      uint8 *encoded_signal_header = (uint8 *) malloc(
                  sizeof(uint8) * 48);
      //interleaving
      uint8 *interleaved_signal_header = (uint8 *) malloc(sizeof(uint8) * 48);
      //length of the psdu coming from the mac layer
      int length = data_frame->psdu_size;
      // first 4 bits represent the modulation and coding scheme
      signal_header[0] = get_bit(data_ofdm->rate_field, 3);
      signal_header[1] = get_bit(data_ofdm->rate_field, 2);
      signal_header[2] = get_bit(data_ofdm->rate_field, 1);
      signal_header[3] = get_bit(data_ofdm->rate_field, 0);
      // 5th bit is reserved and must be set to 0
      signal_header[4] = 0;
      // then 12 bits represent the length
      signal_header[5] = get_bit(length, 0);
      signal_header[6] = get_bit(length, 1);
      signal_header[7] = get_bit(length, 2);
      signal_header[8] = get_bit(length, 3);
      signal_header[9] = get_bit(length, 4);
      signal_header[10] = get_bit(length, 5);
      signal_header[11] = get_bit(length, 6);
      signal_header[12] = get_bit(length, 7);
      signal_header[13] = get_bit(length, 8);
      signal_header[14] = get_bit(length, 9);
      signal_header[15] = get_bit(length, 10);
      signal_header[16] = get_bit(length, 11);
      //18-th bit is the parity bit for the first 17 bits
      int sum = 0;
      int i;
      for (i = 0; i < 17; i++) {
            if (signal_header[i]) {
                  sum++;
            }
      }
      signal_header[17] = sum % 2;
      // last 6 bits must be set to 0
      for (i = 0; i < 6; i++) {
```

```
            signal_header[18 + i] = 0;
        }
        //allocating an OFDM parameter and a FRAME parameter objects
        ofdm_param_intialization(BPSK_1_2, signal_ofdm);
        frame_param_intialization(signal_ofdm, signal_param, 0);
        //convolutional encoding (scrambling is not needed)
        convolutional_encoding(signal_header, encoded_signal_header,
signal_param);
        // interleaving
        interleave(encoded_signal_header, out, signal_param, signal_ofdm);
        free(interleaved_signal_header);
        free(encoded_signal_header);
        free(signal_header);
        return out;
    }
```

B.1.5 Code of the Chuncks to symbols block

B.1.5.1 chuncks_to_symbols.h

```
    /*
     * chunks_to_symbols_impl.h
     *
     *  Created on: Feb 23, 2017
     *      Author: Shereen Othman
     */
    #ifndef CHUNKS_TO_SYMBOLS_IMPL_H_
    #define CHUNKS_TO_SYMBOLS_IMPL_H_
    #include "utils.h"
    void chunks_to_symbols_impl(uint8 *input_items, float32 *output_items,
              int data_size, Encoding encoding);
    #endif /* CHUNKS_TO_SYMBOLS_IMPL_H_ */
```

B.1.5.2 chuncks_to_symbols.c

```
    /*
     * chunks_to_symbols_impl.c
     *
     *  Created on: Feb 23, 2017
     *      Author: Shereen Othman
     */
    #include "chunks_to_symbols_impl.h"
    #include "constellations_impl.h"
    // This is the general work function of the Chunks to symbols block that is
used to // modulate the output bits from the Mapper and Packet header
generator according to // the frame parameters
    void chunks_to_symbols_impl(uint8 *input_items, float32 *output_items,
              int data_size, Encoding encoding) {
        float32 *d_mapping;
         switch (encoding) {
        case BPSK_1_2:
        case BPSK_3_4:
              d_mapping = constellation_bpsk_impl();
```

```
            break;
        case QPSK_1_2:
        case QPSK_3_4:
            d_mapping = constellation_qpsk_impl();
            break;
        case QAM16_1_2:
        case QAM16_3_4:
            d_mapping = constellation_16qam_impl();
            break;
        case QAM64_2_3:
        case QAM64_3_4:
            d_mapping = constellation_64qam_impl();
            break;
        default:
            printf("wrong encoding");
            assert(false);
            break;
        }
        int i;
        int index = 0;
        for (i = 0; i < data_size; i++) {
            index = (int) input_items[i];
            output_items[i * 2] = d_mapping[index * 2];
            output_items[(i * 2) + 1] = d_mapping[(index * 2) + 1];
        }
        free(d_mapping);
    }
```

B.1.5.3 constellation_impl.h

```
/*
 * constellations_impl.h
 *
 *  Created on: Feb 23, 2017
 *      Author: Shereen Othman
 */
#ifndef CONSTELLATIONS_IMPL_H_
#define CONSTELLATIONS_IMPL_H_
#include  "IEEE802_11_Common_Variables.h"
float *constellation_bpsk_impl();
float *constellation_qpsk_impl();
float *sconstellation_16qam_impl();
float *constellation_64qam_impl();
#endif /* CONSTELLATIONS_IMPL_H_ */
```

B.1.5.4 constellation_impl.c

```
/*
 * constellations_impl.c
 *
 *  Created on: Feb 23, 2017
 *      Author: Shereen Othman
 */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

```
#include "chunks_to_symbols_impl.h"
#include "constellations_impl.h"
/**------------------------------FUNCTIONS IMPLEMENTATIONS-------------------
----**/
// This function is used to implement the BPSK constellation using gray coding
float32 *constellation_bpsk_impl() {
	float32 *d_constellation = calloc(2 * 2, sizeof(float32)); //as each
complex number will take 2 successive bytes
	d_constellation[0] = -1; //first real
	d_constellation[1] = 0; //first imag
	d_constellation[2] = 1; //second imag
	d_constellation[3] = 0; //second imag
	return d_constellation;
}
// This function is used to implement the QPSK constellation using gray coding
float32 *constellation_qpsk_impl() {
	const float32 level = sqrt((float32) (0.5));
	float32 *d_constellation = calloc(4 * 2, sizeof(float32));

	d_constellation[0] = -level;
	d_constellation[1] = -level;
	d_constellation[2] = level;
	d_constellation[3] = -level;
	d_constellation[4] = -level;
	d_constellation[5] = level;
	d_constellation[6] = level;
	d_constellation[7] = level;
	return d_constellation;
}
// This function is used to implement the QAM16 constellation using gray
coding
float32 *constellation_16qam_impl() {
	const float32 level = sqrt((float32) (0.1));
	float32 *d_constellation = calloc(16 * 2, sizeof(float32));
	d_constellation[0] = -3 * level;
	d_constellation[1] = -3 * level;
	d_constellation[2] = 3 * level;
	d_constellation[3] = -3 * level;
	d_constellation[4] = -1 * level;
	d_constellation[5] = -3 * level;
	d_constellation[6] = 1 * level;
	d_constellation[7] = -3 * level;
	d_constellation[8] = -3 * level;
	d_constellation[9] = 3 * level;
	d_constellation[10] = 3 * level;
	d_constellation[11] = 3 * level;
	d_constellation[12] = -1 * level;
	d_constellation[13] = 3 * level;
	d_constellation[14] = 1 * level;
	d_constellation[15] = 3 * level;
	d_constellation[16] = -3 * level;
	d_constellation[17] = -1*level;
	d_constellation[18] = 3 * level;
	d_constellation[19] = -1 * level;
	d_constellation[20] = -1 * level;
```

```
        d_constellation[21] = -1 * level;
        d_constellation[22] = 1 * level;
        d_constellation[23] = -1 * level;
        d_constellation[24] = -3 * level;
        d_constellation[25] = 1 * level;
        d_constellation[26] = 3 * level;
        d_constellation[27] = 1 * level;
        d_constellation[28] = -1 * level;
        d_constellation[29] = 1 * level;
        d_constellation[30] = 1 * level;
        d_constellation[31] = 1 * level;
        return d_constellation;
}
// This function is used to implement the QAM64 constellation using gray
coding
float32 *constellation_64qam_impl() {
        const float32 level = sqrt((float32) (1 / 42.0));
        float32 *d_constellation = calloc(16 * 2, sizeof(float32));
        d_constellation[0] = -7 * level;
        d_constellation[1] = -7 * level;
        d_constellation[2] = 7 * level;
        d_constellation[3] = -7 * level;
        d_constellation[4] = -1 * level;
        d_constellation[5] = -7 * level;
        d_constellation[6] = 1 * level;
        d_constellation[7] = -7 * level;
        d_constellation[8] = -5 * level;
        d_constellation[9] = -7 * level;
        d_constellation[10] = 5 * level;
        d_constellation[11] = -7 * level;
        d_constellation[12] = -3 * level;
        d_constellation[13] = -7 * level;
        d_constellation[14] = 3 * level;
        d_constellation[15] = -7 * level;
        d_constellation[16] = -7 * level;
        d_constellation[17] = 7 * level;
        d_constellation[18] = 7 * level;
        d_constellation[19] = 7 * level;
        d_constellation[20] = -1 * level;
        d_constellation[21] = 7 * level;
        d_constellation[22] = 1 * level;
        d_constellation[23] = 7 * level;
        d_constellation[24] = -5 * level;
        d_constellation[25] = 7 * level;
        d_constellation[26] = 5 * level;
        d_constellation[27] = 7 * level;
        d_constellation[28] = -3 * level;
        d_constellation[29] = 7 * level;
        d_constellation[30] = 3 * level;
        d_constellation[31] = 7 * level;
        d_constellation[32] = -7 * level;
        d_constellation[33] = -1 * level;
        d_constellation[34] = 7 * level;
        d_constellation[35] = -1 * level;
        d_constellation[36] = -1 * level;
```

```
d_constellation[37] = -1 * level;
d_constellation[38] = 1 * level;
d_constellation[39] = -1 * level;
d_constellation[40] = -5 * level;
d_constellation[41] = -1 * level;
d_constellation[42] = 5 * level;
d_constellation[43] = -1 * level;
d_constellation[44] = -3 * level;
d_constellation[45] = -1 * level;
d_constellation[46] = 3 * level;
d_constellation[47] = -1 * level;
d_constellation[48] = -7 * level;
d_constellation[49] = 1 * level;
d_constellation[50] = 7 * level;
d_constellation[51] = 1 * level;
d_constellation[52] = -1 * level;
d_constellation[53] = 1 * level;
d_constellation[54] = 1 * level;
d_constellation[55] = 1 * level;
d_constellation[56] = -5 * level;
d_constellation[57] = 1 * level;
d_constellation[58] = 5 * level;
d_constellation[59] = 1 * level;
d_constellation[60] = -3 * level;
d_constellation[61] = 1 * level;
d_constellation[62] = 3 * level;
d_constellation[63] = 1 * level;
d_constellation[64] = -7 * level;
d_constellation[65] = -5 * level;
d_constellation[66] = 7 * level;
d_constellation[67] = -5 * level;
d_constellation[68] = -1 * level;
d_constellation[69] = -5 * level;
d_constellation[70] = 1 * level;
d_constellation[71] = -5 * level;
d_constellation[72] = -5 * level;
d_constellation[73] = -5 * level;
d_constellation[74] = 5 * level;
d_constellation[75] = -5 * level;
d_constellation[76] = -3 * level;
d_constellation[77] = -5 * level;
d_constellation[78] = 3 * level;
d_constellation[79] = -5 * level;
d_constellation[80] = -7 * level;
d_constellation[81] = 5 * level;
d_constellation[82] = 7 * level;
d_constellation[83] = 5 * level;
d_constellation[84] = -1 * level;
d_constellation[85] = 5 * level;
d_constellation[86] = 1 * level;
d_constellation[87] = 5 * level;
d_constellation[88] = -5 * level;
d_constellation[89] = 5 * level;
d_constellation[90] = 5 * level;
d_constellation[91] = 5 * level;
```

```
        d_constellation[92] = -3 * level;
        d_constellation[93] = 5 * level;
        d_constellation[94] = 3 * level;
        d_constellation[95] = 5 * level;
        d_constellation[96] = -7 * level;
        d_constellation[97] = -3 * level;
        d_constellation[98] = 7 * level;
        d_constellation[99] = -3 * level;
        d_constellation[100] = -1 * level;
        d_constellation[101] = -3 * level;
        d_constellation[102] = 1 * level;
        d_constellation[103] = -3 * level;
        d_constellation[104] = -5 * level;
        d_constellation[105] = -3 * level;
        d_constellation[106] = 5 * level;
        d_constellation[107] = -3 * level;
        d_constellation[108] = -3 * level;
        d_constellation[109] = -3 * level;
        d_constellation[110] = 3 * level;
        d_constellation[111] = -3 * level;
        d_constellation[112] = -7 * level;
        d_constellation[113] = 3 * level;
        d_constellation[114] = 7 * level;
        d_constellation[115] = 3 * level;
        d_constellation[116] = -1*level;
        d_constellation[117] = 3 * level;
        d_constellation[118] = 1 * level;
        d_constellation[119] = 3 * level;
        d_constellation[120] = -5 * level;
        d_constellation[121] = 3 * level;
        d_constellation[122] = 5 * level;
        d_constellation[123] = 3 * level;
        d_constellation[124] = -3 * level;
        d_constellation[125] = 3 * level;
        d_constellation[126] = 3 * level;
        d_constellation[127] = 3 * level;
        return d_constellation;
}
```

B.1.6 Code of the OFDM carrier allocator

B.1.6.1 ofdm_carr_alloc_func.h file

```
/*
 * ofdm_carr_alloc_func.h
 *
 *  Created on: Feb 24, 2017
 *      Author: Mohamed Elnaggar
 */
#ifndef OFDM_CARR_ALLOC_FUNC_H_
#define OFDM_CARR_ALLOC_FUNC_H_
#include  "IEEE802_11_Common_Variables.h"
/**----------------------All are form the standard definition----------------
----**/
```

```
// Occupied carriers
 int occupied_carriers[48] = { -26, -25, -24, -23, -22, -20, -19, -18, -17, -
                                    16,
    -15, -14, -13, -12, -11, -10, -9, -8, -6, -5, -4, -3, -2, -1, 1, 2, 3,
    4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24,
                                25, 26 };
// positions of the pilot symbols
int pilot_carriers[4] = { -21, -7, 7, 21 };
// values of the pilot symbols
 float32 pilot_symbols[] = { 1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1,
      0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1,
     0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1,
      0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0,
     -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0,
      1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
     -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0,
      1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0,
      1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0,
     1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1,
      0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0,
      1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1,
     0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1,
     0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1,
      0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1,
      0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0,
     1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1,
      0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1,
     0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1,
     0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0,
      1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0,
     1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0,
       -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0,
      -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1,
     0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1,
      0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1,
     0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0,
     -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0,
      1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0,
     -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1,
     0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1,
      0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1,
      0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1,
     0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0,
     -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0,
     -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0,
     -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0,
      1, 0, 1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1, 0, -1, 0, -1, 0,
      1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1,
       0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0, 1, 0,
```

```
                                -1, 0, -1, 0, -1, 0, 1 };
// synchronization words
float32 sync_words[] = { 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0,
     0.0, 0, 1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
         -1.4719601443879746, -1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
          1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
         -1.4719601443879746, -1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
         -1.4719601443879746, -1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
     1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0,
      0.0, 0, 0.0, 0, 0.0, 0, -1.4719601443879746, -1.4719601443879746, 0.0,
        0, 0.0, 0, 0.0, 0, -1.4719601443879746, -1.4719601443879746, 0.0, 0,
     0.0, 0, 0.0, 0, 1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0,
     0.0, 0, 1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
         1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
     1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0,
      0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0,
     0.0, 0, 0.0, 0, 1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0,
      0.0, 0, -1.4719601443879746, -1.4719601443879746, 0.0, 0, 0.0, 0, 0.0,
        0, 1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
         -1.4719601443879746, -1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
         -1.4719601443879746, -1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
     1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0,
      0.0, 0, 0.0, 0, 0.0, 0, -1.4719601443879746, -1.4719601443879746, 0.0,
        0, 0.0, 0, 0.0, 0, -1.4719601443879746, -1.4719601443879746, 0.0, 0,
     0.0, 0, 0.0, 0, 1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0,
     0.0, 0, 1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
         1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0,
     1.4719601443879746, +1.4719601443879746, 0.0, 0, 0.0, 0, 0.0, 0, 0.0, 0,
     0.0, 0, 0.0, 0, 0.0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 1,
      -1, 0, 0, 1, -1, 0, 0, 1, -1, 0, 0, -1, 1, 0, 0, 1, 1, 0, 0, -1, -1, 0,
       0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, -1, 0, -0, -1, 1, 0, 0, -1,
      -1, 0, 0, 1, 0, 0, 0, -1, 1, 0, -0, -1, 1, 0, 0, -1, 1, 0, 0, 1, -1, 0,
       0, -1, 1, 0, -0, -1, -1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1,
     -1, 0, 0, -1, 1, 0, 0, 1, 1, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, -1, 0, -1, 0, 1, 0,
     1, 0, -1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, -1, 0, -1,
      0, 1, 0, 1, 0, -1, 0, 1, 0, -1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0,
     -1, 0, -1, 0, 1, 0, 1, 0, -1, 0, 1, 0, -1, 0, 1, 0, -1, 0, -1, 0, -1, 0,
      -1, 0, -1, 0, 1, 0, 1, 0, -1, 0, -1, 0, 1, 0, -1, 0, 1, 0, -1, 0, 1, 0,
                 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

   // size of FFT
   int fft_len = 64;
   // This boolean is used to return the OFDM carrier allocater to the start
   int output_is_shifted = 1;
   int ofdm_carr_alloc(int *occupied_carriers, int *pilot_carriers,
             float32 *pilot_sympols, float32 *sync_words, int fft_len,
             int output_is_shifted, float32 *input, float32 *output,
             int sizeof_input_sym);
   #endif /* OFDM_CARR_ALLOC_FUNC_H_ */
```

B.1.6.2 Ofdm_carr_alloc.c file

```
   /*
    * ofdm_carr_alloc.c
```

```
 *
 *  Created on: Feb 24, 2017
 *      Author: Mohamed Elnaggar
 */
#ifndef ofdm
#define ofdm
#include  "IEEE802_11_Common_Variables.h"
// The general work function used for OFDM carrier allocation
int ofdm_carr_alloc(int *occupied_carriers, int *pilot_carriers,
          float32 *pilot_symbols, float32 *sync_words, int fft_len,
          int output_is_shifted, float32 *input, float32 *output,
          int sizeof_input_sym) {
      int i = 0;
      int j = 0;
      int sizeof_occ_carr = 48;
      int sizeof_pilot_carr = 4;
      int sizeof_sync_words = 512;
      //this part changes the zero values in the occupied_carriers to positive
to  indicate real positions in array
      for (i = 0; i < sizeof_occ_carr; i++) {
            if (occupied_carriers[i] < 0) {
                  occupied_carriers[i] += fft_len;
            }
            if (occupied_carriers[i] > fft_len || occupied_carriers[i] < 0) {
                  break;
            }
            if (output_is_shifted) {
                  occupied_carriers[i] = (occupied_carriers[i] + fft_len /
2)
                              % fft_len;
            }
      }
      //This part changes the zero values in the pilot_carriers to positive to
indicate real positions in array
      for (i = 0; i < sizeof_pilot_carr; i++) {
            if (pilot_carriers[i] < 0) {
                  pilot_carriers[i] += fft_len;
            }
            if (pilot_carriers[i] > fft_len || pilot_carriers[i] < 0) {
                  break;
            }
            if (output_is_shifted) {
                  pilot_carriers[i] = (pilot_carriers[i] + fft_len / 2) %
fft_len;
            }
      }
      // Copy Sync word
      for (i = 0; i < sizeof_sync_words; i++) {
            output[i] = sync_words[i];

      }
      // Copy data symbols
      float32 *out_data;
      out_data = 512 + output;
      long n_ofdm_symbols = 0; // Number of output items
```

```
	int symbols_to_allocate = 48;
	int symbols_allocated = 0;
	int k;
	for (i = 0; i < sizeof_input_sym; i = i + 2) {
		if (symbols_allocated == 0) {
			n_ofdm_symbols++;
		}
		k = occupied_carriers[symbols_allocated];
		k = k * 2;
		out_data[k] = input[i];
		out_data[k + 1] = input[i + 1];
		symbols_allocated++;
		if (symbols_allocated == symbols_to_allocate) {
			symbols_to_allocate = 48;
			symbols_allocated = 0;
			out_data = out_data + ((fft_len) * 2);
		}
	}
	// Copy pilot symbols
	float32 *out_pilot;
	out_pilot = 512 + output;
	for (i = 0; i < n_ofdm_symbols; i++) {
		for (j = 0; j < 8; j = j + 2) {
			k = pilot_carriers[j / 2];
			k = k * 2;
			out_pilot[k] = pilot_symbols[j + (i * 8)];
			out_pilot[k + 1] = pilot_symbols[j + 1 + (i * 8)];
		}
		out_pilot = out_pilot + ((fft_len) * 2);
	}
	return 0;
}
#endif
```

B.1.7 Code of the IFFT block

B.1.7.1 Ifft.h

```
/*
 * ifft.h
 *
 *  Created on: Feb 7, 2017
 *      Author: Habiba Tarek
 */
#ifndef IFFT_H_
#define IFFT_H_
#include "IEEE802_11_Common_Variables.h"
extern void gen_twiddle_fft_sp (float32 *w, int n);
extern void shiftF(float32* before, float32* after, int N);
extern void ifft(float32* input, float32* output, int N, float32
WindowScale, int shift, float32* window);
extern void seperateRealImg(float32* input, float32* real, float32*img, int
N);
#endif /* IFFT_H_ */
```

B.1.7.2 Ifft.c

```
/* ifft.c
 * Created on: Feb 7, 2017
 *      Author: Habiba Tarek*/
#include "ifft.h"
#include <math.h>
#include <ti/dsplib/dsplib.h>
#include <stdlib.h>
extern uint8 brev[64] = {
    0x0, 0x20, 0x10, 0x30, 0x8, 0x28, 0x18, 0x38,
    0x4, 0x24, 0x14, 0x34, 0xc, 0x2c, 0x1c, 0x3c,
    0x2, 0x22, 0x12, 0x32, 0xa, 0x2a, 0x1a, 0x3a,
    0x6, 0x26, 0x16, 0x36, 0xe, 0x2e, 0x1e, 0x3e,
    0x1, 0x21, 0x11, 0x31, 0x9, 0x29, 0x19, 0x39,
    0x5, 0x25, 0x15, 0x35, 0xd, 0x2d, 0x1d, 0x3d,
    0x3, 0x23, 0x13, 0x33, 0xb, 0x2b, 0x1b, 0x3b,
    0x7, 0x27, 0x17, 0x37, 0xf, 0x2f, 0x1f, 0x3f
};
// since complex here is written in the terms of a float32 array, with real
in even // indices and imaginary in odd indices, this function separates real
and imaginary // numbers in different arrays*/
void seperateRealImg(float32* input, float32* real, float32*img, int N) {
    int i, j;
    for (i = 0, j = 0; j < N; i+=2, j++) {
        real[j] = input[i];
        img[j] = input[i + 1];
    }
}
/* this ready function generates the twiddle factors that will be used in
IFFT function*/
void gen_twiddle_fft_sp (float32 *w, int n)
{
    int i, j, k;
    double x_t, y_t, theta1, theta2, theta3;
    const double PI = 3.141592654;
    for (j = 1, k = 0; j <= n >> 2; j = j << 2)
    {
        for (i = 0; i < n >> 2; i += j)
        {
            theta1 = 2 * PI * i / n;
            x_t = cos (theta1);
            y_t = sin (theta1);
            w[k] = (float32) x_t;
            w[k + 1] = (float32) y_t;

            theta2 = 4 * PI * i / n;
            x_t = cos (theta2);
            y_t = sin (theta2);
            w[k + 2] = (float32) x_t;
            w[k + 3] = (float32) y_t;

            theta3 = 6 * PI * i / n;
            x_t = cos (theta3);
            y_t = sin (theta3);
```

```
                w[k + 4] = (float32) x_t;
                w[k + 5] = (float32) y_t;
                k += 6;
            }
        }
    }
    /* this function shifts the input so that it swaps the two halves of the
input (gnuradio uses shift so we did it here as well) */
    void shiftF(float32* before, float32* after, int N)
    {
              int n;
              for(n = 0; n < (2*N)/2 ; n++)
              {
                    after[n] = before[(2*N)/2 + n];
              }
              n = (2*N)/2;
              for(n = (2*N)/2; n < (2*N) ; n++)
              {
                    after[n] = before[n - (2*N)/2];
              }
    }
    void ifft(float32* input, float32* output, int N, float32 WindowScale, int
shift, float32* window)
    {
         gen_twiddle_fft_sp(window, N);
         /* this small part multiples the input by N to reverse the
normalization and divides it by the the value of WindowScale to scale the
input */
            int k = 0;
            for (k = 0; k<2*N; k++)
            {
              input[k] = input[k] * ((float32) N /
(sqrt((float32)WindowScale)));
            }
            if(shift  == true)
            {
              short int len = (unsigned int)(ceil(2*N/2.0));
              float32* temp = calloc(2*N, sizeof(float32));
                memcpy(temp, &input[len],sizeof(float32)*(2*N - len));
                memcpy(&temp[2*N - len], &input[0],sizeof(float32)*len);
             memcpy(input,temp,2*N*sizeof(float32));
              free(temp);
            }
            if(N%4 == 0)
            {
              DSPF_sp_ifftSPxSP(N, input, window, output, brev, 4, 0, N);
            }
            else
            {
              DSPF_sp_ifftSPxSP(N, input, window, output, brev, 2, 0, N);
            }
    }
```

B.1.8 Code of the cyclic prefix block

B.1.8.1 CyclicPrefix.h

```
/*CyclicPrefix.h
 *Created on: 1 Feb 2017
 *Author: User Zeinab Ahmed*/
#ifndef CYCLICPREFIX_H_
#define CYCLICPREFIX_H_
#include  "IEEE802_11_Common_Variables.h"
typedef struct
{
   int d_fft_len; // initialized in gnuradio by 64 (size of IFFT block)
   int d_output_size; // output size = d_fft_len + cyclic prefix
size(initialized // in gnuradio by 16)
   int d_rolloff_len; // initialized in gnuradio by 2
   int symbols_to_read; // number of OFDM symbols input to this block
}StructCyclicPrefix_Init;
void CyclicPrefix(StructCyclicPrefix_Init*,float32*,float32*);
void CyclicPrefix_Init(StructCyclicPrefix_Init*,int);
#endif /* CYCLICPREFIX_H_ */
```

B.1.8.2 CyclicPrefix.c

```
/*CyclicPrefix.c
 *Created on: 1 Feb 2017
 *Author: User Zeinab Ahmed*/
#include "CyclicPrefix.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h> // added as I use cosine function
#ifndef M_PI
#    define M_PI 3.14159265358979323846
#endif
// The initialization of the Cyclic prefix
void CyclicPrefix_Init(StructCyclicPrefix_Init *Cyclic,int num_ofdm_sym)
{
   Cyclic->d_fft_len= 64;
   Cyclic->d_output_size=80;
   Cyclic->d_rolloff_len=2;
   Cyclic->symbols_to_read=num_ofdm_sym;
}
// The general work function of the cyclic prefix block
void CyclicPrefix(StructCyclicPrefix_Init *PtrToStruct,float32
*data_ptr,float32 *out_ptr)
{
   int i;
   int d_cp_size = PtrToStruct->d_output_size - PtrToStruct->d_fft_len;
   float32 *d_up_flank,*d_down_flank,*d_delay_line;
    if (PtrToStruct->d_rolloff_len == 1)
    {
          PtrToStruct->d_rolloff_len = 0;
    }
```

```
        if (PtrToStruct->d_rolloff_len)
        {
              d_up_flank=(float32*) malloc((PtrToStruct->d_rolloff_len-
1)*sizeof(float32));
              d_down_flank=(float32*) malloc((PtrToStruct->d_rolloff_len-
1)*sizeof(float32));
              d_delay_line=(float32*) malloc((PtrToStruct->d_rolloff_len-
1)*sizeof(float32));
              //---------- construct up flank and down flank ----------//
              for (i = 1; i < PtrToStruct->d_rolloff_len; i++)
              {
                    d_up_flank[i-1]   = 0.5 * (1 + cos(M_PI *(float32)
i/(float32)PtrToStruct->d_rolloff_len - M_PI));
                    d_down_flank[i-1] = 0.5 * (1 + cos(M_PI
*(float32)(PtrToStruct->d_rolloff_len-i)/(float32)PtrToStruct->d_rolloff_len -
M_PI));
                    d_delay_line[i-1]=0;
              }
        }
       //---------- cyclic prefix implementation ----------//
       float32 *in=data_ptr;
       float32 *out=out_ptr;
       int sym_idx;
        for (sym_idx = 0; sym_idx < PtrToStruct->symbols_to_read;sym_idx++)
        {
              memcpy((out + (d_cp_size*2)),in, PtrToStruct->d_fft_len *
sizeof(float32)* 2);
              memcpy(out,(in + (PtrToStruct->d_fft_len*2) - (d_cp_size*2)),
d_cp_size * sizeof(float32)* 2);
       if (PtrToStruct->d_rolloff_len)
       {
         for (i = 0; i < PtrToStruct->d_rolloff_len-1; i+=2)
         {
           out[i] = out[i] * d_up_flank[i/2] + d_delay_line[i/2]; //real part
           out[i+1] = out[i+1] * d_up_flank[i/2] + d_delay_line[i/2];
           d_delay_line[i/2] = in[i] * d_down_flank[i/2]; //real part
           d_delay_line[i/2] = in[i+1] * d_down_flank[i/2]; //imaginary part
         }
       }
              in += (PtrToStruct->d_fft_len*2);
              out += (PtrToStruct->d_output_size*2);
         }
   //---------- adding delay line for the last OFDM symbol ----------//
        if (PtrToStruct->d_rolloff_len)
        {
               for (i = 0; i < PtrToStruct->d_rolloff_len-1; i++)
               {
                     *out++ = d_delay_line[i/2]; // real part
                   *out++ = d_delay_line[i/2]; // imaginary part
               }
              free(d_delay_line);
              free(d_down_flank);
              free(d_up_flank);
       }
}
```