Software Defined Radio implementation of IEEE 802.11a Physical and MAC layer transceiver on General Purpose Processors

Manolis Surligas

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, Heraklion, GR-70013, Crete, Greece

Thesis Advisors: Prof. *Apostolos Traganitis*
Co-advisor: Dr. *Stefanos Papadakis*

This work was partially supported by the Rapid Emergency Deployment mobile Communication infrastructure (REDComm) project with the financial support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme. **European Commission - Directorate-General Home Affairs.**

University of Crete
Computer Science Department

Software Defined Radio implementation of IEEE 802.11a Physical and MAC layer transceiver on General Purpose Processors

Thesis submitted by
Manolis Surligas
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: Manolis Surligas

Committee approvals:

Apostolos Traganitis
Professor, Thesis Supervisor

Angelos Bilas
Professor, Committee Member

Athanasios Mouchtaris
Assistant Professor, Committee Member

Stefanos Papadakis
Researcher - Visiting Professor, Committee Member

Departmental approval:

Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, October 2014

Abstract

Over the last years the number of wirelessly connected devices has exponentially increased, along with the user need in bandwidth. Consequently, new technologies are under development to fulfill the user demands. However, the development and adaption of these new technologies has become a business critical issue.

Software Defined Radios (SDRs) provide great flexibility, reconfigurability and cost efficiency, aspects that can contribute to an increased rate of new technologies adaption. However, SDR implementations also introduce challenges, often related with the high computational resources that are required. Realtime performance of high bandwidth systems like IEEE 802.11a can be a challenging task for a software implementation. Towards this goal, this work exploits all the available optimization techniques and technologies including multi-threading and Single Instruction, Multiple Data (SIMD) instructions.

In this work we present a fully software implementation of an IEEE 802.11a standard compliant transceiver based on the GNU Radio SDR platform. Going beyond existing previous works, which fell short to provide a unified and fully operational system, our approach provides a realtime IEEE 802.11a Media Access Control (MAC) and physical layer system that can be handled as a normal wireless network interface by the host Operating System.

The performance of the system is evaluated both in terms of 802.11a capabilities, as well as in computational resources requirements. The results manifest a quite robust physical layer implementation, which is on par with the best commercial hardware. The MAC layer capacity is only limited by the SDR hardware latencies and the computing demands can be met by many modern CPUs. Moreover many of the optimization techniques and technological or theoretical limitations explored and addressed in this work are quite generic, and may set a road-map for future SDR implementations and new technologies development.

Περίληψη

Τα τελευταία χρόνια ο αριθμός των ασύρματων συσκευών έχει αυξηθεί εκθετικά, παράλληλα με τις απαιτήσεις των χρηστών σε όγκο δεδομένων. Παρόλο που νέες τεχνολογίες βρίσκονται διαρκώς υπό ανάπτυξη για να εκπληρώσουν τις ανάγκες των χρηστών, τόσο η υλοποίηση όσο και η αποδοχή αυτών των νέων τεχνολογιών είναι πλέον ένα φλέγον επιχειρηματικό ζήτημα.

Οι πομποδέκτες διαμορφούμενοι από λογισμικό, γνωστοί και ως Software Defined Radios (SDRs), παρέχουν υπέρμετρη ευελιξία, δυνατότητα επαναπροσδιορισμού και αποδοτικότητα, πτυχές που οδηγούν σε έναν καλύτερο ρυθμό αποδοχής των νέων τεχνολογιών. Παρόλα αυτά, οι υλοποιήσεις σε SDR περιβάλλον, εισάγουν δυσκολίες οι οποίες συχνά σχετίζονται με τους υπολογιστικούς πόρους που απαιτούνται. Η εκτέλεση σε πραγματικό χρόνο συστημάτων με μεγάλο εύρος ζώνης όπως το IEEE 802.11a, μπορεί να είναι μια αρκετά δύσκολη πρόκληση για μια υλοποίηση σε λογισμικό. Προς αυτήν την κατεύθυνση, η παρούσα εργασία εκμεταλλεύεται όλες τις τεχνολογίες και τις διαθέσιμες τεχνικές βελτιστοποίησης, συμπεριλαμβανομένων τεχνικών παράλληλης εκτέλεσης και εντολών Single Instruction, Multiple Data (SIMD).

Στην εργασία αυτή παρουσιάζουμε μια υλοποίηση, εξ ολοκλήρου σε λογισμικό, ενός πομποδέκτη συμβατού με το πρωτόκολλο IEEE 802.11a, η οποία βασίστηκε στην SDR πλατφόρμα GNU Radio. Ξεπερνώντας προηγούμενες προσπάθειες, οι οποίες δεν κατάφεραν να παρουσιάσουν ένα ενοποιημένο και πλήρως λειτουργικό σύστημα, η υλοποίησή μας παρέχει ένα σύστημα, ικανό να εκτελείται σε πραγματικό χρόνο και συμβατό τόσο με το φυσικό όσο και με το επίπεδο Media Access Control (MAC) του IEEE 802.11a πρωτοκόλλου, το οποίο μπορεί να χρησιμοποιηθεί από το λειτουργικό σύστημα σαν ένας κανονικός ασύρματος προσαρμογέας δικτύου.

Η απόδοση του συστήματος αξιολογείται τόσο με βάση τις δυνατότητές του όσο και με τους υπολογιστικούς πόρους που απαιτεί η εκτέλεσή του. Τα αποτελέσματα υποδεικνύουν μία αρκετά αξιόπιστη υλοποίηση του φυσικού επιπέδου, συγκρίσιμη με τις καλύτερες εμπορικές συσκευές. Ο όγκος δεδομένων που μπορεί να διαχειριστεί το MAC επίπεδο, περιορίζεται μόνο από τις καθυστερήσεις που εισάγει το υλικό του SDR και οι υπολογιστικές απαιτήσεις μπορούν εύκολα να ικανοποιηθούν από έναν σύγχρονο επεξεργαστή. Επιπλέον, πολλές από τις τεχνικές βελτιστοποίησης καθώς και διάφοροι τεχνολογικοί ή θεωρητικοί περιορισμοί που παρουσιάζονται και αντιμετωπίζονται στην εργασία αυτή μπορούν να εφαρμοσθούν γενικότερα, και παρέχουν έτσι μία κατευθυντήρια γραμμή τόσο σε μελλοντικές υλοποιήσεις SDR όσο και στην ανάπτυξη νέων τεχνολογιών.

Acknowledgements

I would like to thank my supervisors, Prof. Apostolos Traganitis and Stefanos Papadakis for their guidance and help during this work. The collaboration with Stefanos was excellent and for me he was a friend rather an advisor. Many thanks to Andreas Miaoudakis for the help and numerous discussions about many OFDM issues. Also, special thanks to Antonis Makrogiannakis for the help on the SSDR driver.

Last, I would like to thank all those people that supported me all over these years. Magda, Lefteris and Moumia (just kidding, Giorgos X. and Andromachi), Giorgos P., Giannis, Pantelis, Aggeliki, Hlias and of course my family that always were there, when I needed them.

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Methodology . . . 4
- 1.3 Roadmap . . . 4
- 1.4 Related Work . . . 5

2 Software Defined Radios **7**
- 2.1 History . . . 9
- 2.2 Software Platforms . . . 10
- 2.3 Hardware . . . 11
- 2.4 The GNU Radio Platform . . . 11
  - 2.4.1 General Description . . . 11
  - 2.4.2 The Processing Block . . . 12
  - 2.4.3 Scheduler and Flowgraph . . . 14
  - 2.4.4 GNU Radio Companion . . . 14

3 IEEE 802.11a PHY Software Implementation **17**
- 3.1 IEEE 802.11a Standard . . . 17
  - 3.1.1 Frame Format . . . 18
- 3.2 General Software Implementation Information . . . 20
- 3.3 Transmit Chain . . . 22
  - 3.3.1 Data Scrambling . . . 24
  - 3.3.2 Convolutional Encoder . . . 26
  - 3.3.3 Bit Interleaving . . . 30
  - 3.3.4 Carriers Mapping . . . 33
  - 3.3.5 Inverse Fast Fourier Transform . . . 37
  - 3.3.6 Cyclic Prefix Insertion . . . 37
  - 3.3.7 Transmitting samples using an SDR Device . . . 37
- 3.4 Receive Chain . . . 41
  - 3.4.1 Energy Detection . . . 41
  - 3.4.2 Time and Frequency Synchronization . . . 47
  - 3.4.3 Frame Sampler and Fine Frequency Synchronization . . . 57
  - 3.4.4 Cyclic Prefix Removal . . . 59
  - 3.4.5 Fast Fourier Transform . . . 59
  - 3.4.6 Channel Estimation and Equalization . . . 59
  - 3.4.7 Phase Tracking . . . 65
  - 3.4.8 Constellation De-mapping . . . 69
  - 3.4.9 De-Interleaving . . . 71

3.4.10 Forward Error Correction . . . 72
3.4.11 Descrambling . . . 76
3.5 Sample Rate Conversion using FFT . . . 76

4 Connection with the IEEE 802.1 MAC Layer 79
4.1 General Description . . . 79
4.2 The mac80211 Driver . . . 79
4.3 The ssdr Driver . . . 81
4.4 MAC Functions implemented at the PHY . . . 82

5 Software Optimization Techniques 85
5.1 Vectorized programming . . . 85
5.1.1 Complex multiplication . . . 86
5.1.2 Type casting . . . 87
5.1.3 SIMD enabled FIR Filtering . . . 88
5.1.4 SIMD enabled Viterbi decoder . . . 91
5.2 Bit manipulation . . . 92
5.2.1 Modulo by power of 2 . . . 92
5.2.2 Reverse the bits in a byte . . . 93
5.2.3 Bit Counting . . . 93

6 Evaluation 95
6.1 Physical Layer Performance . . . 95
6.2 Latency and Throughput Measurements . . . 97
6.2.1 UDP Throughput . . . 99
6.2.2 TCP Throughput . . . 100
6.3 CPU Utilization . . . 101

7 Conclusions 105
7.1 Future Work . . . 105

List of Figures

2.1 Concept of SDR hardware 7
2.2 DSP implementation spectrum [1] 8
2.3 Photograph of 2-channel SPEAKeasy II Model Year 1 MBMMR 10
2.4 A simple GNU Radio flowgraph 12
2.5 The GNU Radio Companion tool 15

3.1 The SIGNAL field 19
3.2 The SERVICE field 20
3.3 PPDU frame format 20
3.4 A Problematic Flowgaph Scenario 22
3.5 Solution to the problematic flowgraph of Figue 3.4 22
3.6 Flowraph of the IEEE 802.11a Transmitter 22
3.7 OFDM training structure 24
3.8 Example of multiplicative: (a) Scrambler (b) Descrambler 25
3.9 IEEE 802.11a Data Scrambler 26
3.10 A simple Convolutional Encoder with K=3 and R=1/2 27
3.11 Convolutional encoder of IEEE 802.11 with K=7 and R=1/2 28
3.12 Punctured Coding $R = 2/3$ 28
3.13 Punctured Coding $R = 3/4$ 29
3.14 Software implementation of the IEEE 802.11a encoder using look-up table 30
3.15 A 3×4 block interleaver with $depth = 12$ 31
3.16 A simple convolutional interleaver 31
3.17 Subcarrier frequency allocation 33
3.18 Transfer function of transmit/receive hardware and the impact to the OFDM system design 34
3.19 BPSK, QPSK, 16-QAM, and 64-QAM constellation bit encoding 36
3.20 Inputs and Outputs for the IFFT 37
3.21 Samples buffering at SDR hardware 39
3.22 Improved TX chain 40
3.23 Overview of the IEEE 802.11a receive software implementation 42
3.24 Schmidl & Cox energy detection algorithm 43
3.25 Schmidl and Cox energy detection approach (a) Without normalization (b) Normalized 45
3.26 Proposed method for energy detection (a) Without normalization (b) Normalized 46
3.27 Comparison of (a) Proposed (b) Schmidl & Cox energy detection approach for different SNR values 47
3.28 Impact of the CFO at the cross-correlation using complex and real input signal 51

3.29 Loss of orthogonality due to CFO . . . 53
3.30 Constellation rotation due to CFO . . . 54
3.31 Coarse time and frequency synchronization hierarchical block . . . 55
3.32 Normalized output of the Fine Synchronization algorithm (a) Without phase offset (b) With phase offset . . . 56
3.33 CP removal process . . . 59
3.34 Effect of the channel on the OFDM subcarriers . . . 60
3.35 16-QAM constellation without equalization . . . 60
3.36 Phase offset introduced due to the offset from the ideal start of the OFDM symbol . . . 61
3.37 A bird's eye view of the IEEE 802.11a software receiver . . . 64
3.38 Discontinuity of the *atan2(y, x)* function . . . 67
3.39 BPSK constellation rotation due to phase noise . . . 68
3.40 Trellis Diagram of the convolutional encoder of Figure 3.10 . . . 73
3.41 Viterbi butterflies . . . 74
3.42 Parralel Viterbi decoding and traceback using two threads . . . 75

4.1 The mac80211 architecture . . . 80
4.2 The mac80211 with the ssdr architecture . . . 81
4.3 Structure of PSDU data: (a) retrieved by the ssdr driver (b) PHY to the ssdr driver . . . 82

5.1 (a) Scalar operation (b) SIMD operation . . . 86
5.2 Conditional branch elimination using SIMD instructions . . . 86
5.3 Throughput comparison of different FIR filter types . . . 89
5.4 Using SIMD for (a) minimum branch selection, and (b) calculating the minimum accumulated error for 8 different Viterbi states concurrently . . . 92

6.1 16-QAM constellation using (a) Noctar card (b) USRP B210 . . . 96
6.2 Noctar PHY statistics at 2.462 GHz for 24 Mbps 16-QAM modulation . . . 97
6.3 RTT CDF using BPSK modulation . . . 99
6.4 RTT CDF using 16-QAM modulation . . . 100
6.5 RTT CDF for different modulation schemes . . . 101
6.6 UDP throughput measurements . . . 102
6.7 UDP packet loss ratio . . . 103
6.8 Software components' CPU utilization distribution during full duplex communication . . . 103

List of Tables

3.1 Modulation-dependent parameters 18
3.2 TXVECTOR parameters 18
3.3 RXVECTOR parameters 19
3.4 Contents of the RATE field 24
3.5 Output of the convolutional encoder of Figure 3.10 for a specific input . . . 27
3.6 Example of using interleaving at a transmission over a channel with bursty errors 30
3.7 Modulation-dependent normalization factor K_{MOD} 33
3.8 Example of pilots insertion 35
3.9 Timing related parameters 38
3.10 Frequency domain representation of the short sequences 42
3.11 One period of IFFT of the short sequences 43
3.12 Phase noise characteristics of ADF4351 at 2.2 GHz 67

5.1 Performance comparison of scalar and SIMD instruction on an i7 2600K CPU 88
5.2 Performance comparison of various bitwise operations on an i7 2600K CPU 93

6.1 TX processing delay for different packet sizes and modulation schemes . . . 97
6.2 RX processing delay for different packet sizes and modulation schemes . . . 98
6.3 RTT in milliseconds for different packet sizes and modulation schemes . . . 98
6.4 TCP throughput in Kbits/s for different window size configurations 101
6.5 Userspace CPU utilization of the IEEE 802.11a software transceiver 101

Chapter 1

Introduction

1.1 Motivation

Over the last years the number of wireless devices increased drastically. As the number of these devices is expected to increase even more, it's obvious that wireless telecommunication systems will continue to play a key role in the future.

Not only the number of the wireless devices is rising, but also user demands in bandwidth, coverage, and network density are constantly increasing. At the same time low cost and energy consumption are highly desirable. For this reason, telecommunication research society is always on the move, providing higher capacity, greener and more sophisticated telecommunication standards to fulfill user demands.

Despite this effort, new technologies are adopted at a relatively slow pace. The main reason is the cost. From the side of the telecommunication providers, hardware upgrade is a high cost and time consuming process, while the end users may also be forced to upgrade their equipment in order to benefit from the new technology. In addition, as construction of new hardware is necessary, manufactures face large, costly development and research cycles.

However, along with the flood of wireless devices there is an increasing concern over the sharing of the frequency bands. As all these devices compete to gain access to the same medium, fair and efficient spectrum utilization is a challenging task. Nowadays frequency bands distribution seems to be problematic. Industrial, Scientific and Medical (ISM) spectrum bands are already overcrowded. The joint efforts for a smarter, dynamic and cooperative spectrum allocation, introduced the notion of Cognitive Radios (CRs).

Software Defined Radios (SDRs) can substantially support the lower cost and faster new technologies adaptation, along with the increased efficiency in spectrum utilization. Furthermore, Software Defined Radios (SDRs) enable the design and deployment of multi-standard, multi-band or both implementations providing great flexibility and cost efficiency.

In this work, we implement a software implementation of the Physical Layer (PHY) of IEEE 802.11a. Such an implementation can offer unmatched flexibility and advantages in experimentation, research, in-depth monitoring and investigation.

1.2 Methodology

SDRs are a great platform for experimentation, so our goal was to design a modular implementation that could be a super set of the IEEE 802.11a standard. Such a system can easily support additional non-standard bandwidths, data rates and modulation schemes, providing a great experimentation toolkit, while also being fully compatible with the IEEE 802.11a standard.

Another design issue was the hardware compatibility. An implementation that is independent of specific hardware, can support various SDR hardware with none or minor changes providing great flexibility. However, supporting different SDR hardware devices introduces many challenges too, often related with the different characteristics that each of the devices provide.

The implementation starts with the transmit part of the standard, mainly for debug and simplicity reasons. Transmitting frames is a quite straightforward task without complex information dependencies across the different processing tasks required. Furthermore, transmitted frames can be easily captured with normal wireless devices operating in monitor mode and analyze its contents with a network analyzing tool like Wireshark [2].

After successful transmission of IEEE 802.11a frames, the development continues with the receive part. The output of the transmit can be used as reference, avoiding capturing signals over the air. With this approach challenges introduced by the noise of the live signal or by unknown problems from the SDR hardware, can be avoided.

IEEE 802.11a standard has quite demanding timing restrictions. In order our implementation to support realtime execution, all processing blocks should meet these timing constraints. Therefore, every processing block should be able to produce or consume data in a rate greater than the data rate that the standard dictates. Using the *dd* tool of the GNU Core Utilities and named pipes a rough estimation of the processing data rate can be obtained.

1.3 Roadmap

The remaining part of the thesis proceeds as follows: Firstly some previous works are be discussed in the following section. The next chapter provides a bird's eye view of SDR hardware and software, presenting also some basic aspects of the SDR software platform that was used during this work. In Chapter 3 the software implementation of the whole PHY of 802.11a is discussed, among with the difficulties and the problems that emerged during the development phase. The connection of the PHY software with a fully functional Media Access Control (MAC) layer is presented at Chapter 4. In this chapter the reader can also find information about how the Operating System (OS) can use the PHY software implementation as a normal network device. Chapter 5 includes all the performance optimization techniques that were used in order this software to be able to operate in real-time. In Chapter 6, performance measurements of our implementation are presented, whereas in Chapter 7 the overall performance of the system is presented.

Finally, future work and conclusion sections are summarizing some enhancements that could be implemented and provide some considerations/ideas about SDR development in general, which were gained during the development of the IEEE 802.11a standard.

1.4 Related Work

As the benefits and flexibility of SDRs become more popular, it is expected more developers to get involved in this field. In conjunction with the popularity of the IEEE 802.11a standard and its Orthogonal Frequency Division Multiplexing (OFDM) variants 802.11g and 802.11p, it is reasonable that many other implementations have been done or are currently under development.

So far, most of the software implementations were based on Field Programmable Gate Arrays (FPGAs), Digital Signal Processing (DSP) processors or a combination of them. The constant and predictable processing delays of these architectures make them ideal for the demanding timing constraints of the standard. Such implementations lack of flexibility and maintainability while they demand large development cycles. However, each of these implementations [3, 4, 5, 6] provided vital information about the computational resources and the bottlenecks, opening the way for implementations running entirely on General Purpose Processors (GPPs).

As far as GPP implementations [7, 8, 9, 9, 10] is concerned, there are two different general approaches. At the first approach, the SDR application is a standalone program, created to serve only the purpose of the standard. These type of implementations have significant performance benefits and low latency over the other method, but the lack in terms of flexibility, reconfigurability and maintainability. On the other hand, there are software implementations that are based and extend existent SDR software stacks. This method provides great flexibility sacrificing some computational resources for the additional operations required by the underline software platform.

The majority of the IEEE 802.11 software implementations provide either the transmitter or the receiver. However, those works that provide both, fail to cover all the requirements of the standard. Issues like the acknowledgments, re-transmissions, peers association are some of the fundamental operations of the standard that are not implemented in those works. Furthermore, there is lack on a unified and flexible way of communication with an IEEE 802.11 compliant MAC layer. To our knowledge the only work that fulfills most of these requirements is the Microsoft Research Software Radio (SORA) [8] platform.

Our GPP approach provides a real-time and unified transceiver supported by a fully compliant MAC layer. Despite [8], our implementation relies entirely on software, compromising with the greater delays introduced.

Chapter 2

Software Defined Radios

SDRs are telecommunication systems that perform the signal processing trough modifiable software, instead of specific hardware components. The software can run on GPPs, DSP processors, System on Chip (SoC), FPGAs or any combination of them.

Using a generic RF front-end, that can tune in a wide range of radio frequencies, software can receive and transmit the desired signals. The generic RF front-end is responsible for applying some analog filtering, convert the analog signal in a digital form and vice versa. Furthermore, this generic hardware provides an interconnect fabric for the communication with the host. In most cases, the generic RF front-end accommodates an FPGA, that implements the communication protocol with the host computer and the control of the various hardware components. There are cases where the FPGA is performing also some basic signal processing for CPU offloading reasons.

A basic concept of such a generic RF front-end can be found in Figure 2.1. Baseband data samples derived from the host, are passed through the Digital to Analog Converter (DAC), which converts the digital signal to analog. After some filtering, the filtered analog signal reaches the up converter which is responsible to transform the signal from baseband to the desired target frequency. Then the resulting signal is transmitted through the appropriate antenna. Exactly the reverse procedure occurs in case of reception.

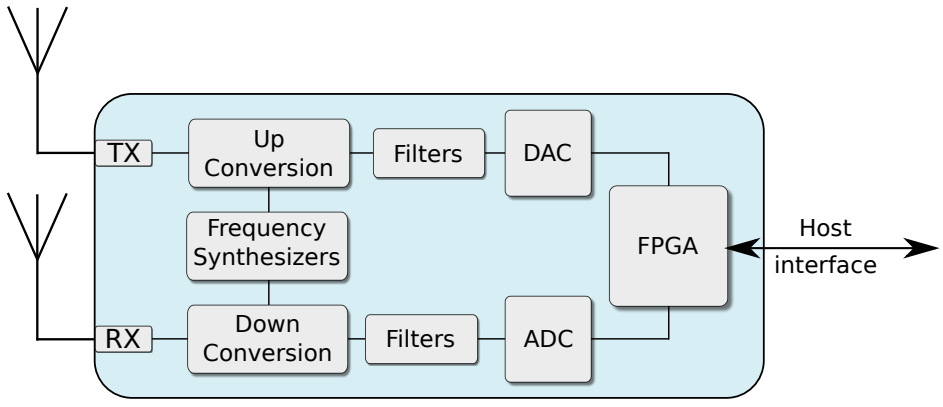


Figure 2.1: Concept of SDR hardware

It is obvious, that this approach provides great flexibility and has many advantages over the standard way that industry follows until today. Some of key aspects of SDRs are:

- *Multiple communication standards support*: One SDR device with enough CPU resources can provide multiple telecommunication protocols simultaneously. This allows the use of only one hardware device simplifying the structure of the wireless node.
- *Easy and cheap upgrades*: Only software patches should applied for upgrades or bug fixes. This a very strong point of SDRs, especially for telecommunication companies, where the market demands always the most recent technology. The cost for hardware upgrades at these situations is prohibiting.
- *Better utilization of the spectrum*: Some regions of the spectrum like the ISM bands are over-crowed where others, like the analog TV bands are not utilized. Also there are frequency regions whose utilization depends on locality or time. These regions can be easily and dynamically exploited by SDRs.
- *Easily develop, test and deploy new telecommunication standards*: SDRs can greatly reduce the time that is needed for creating and testing new telecommunication standards. Until now several simulations where done in special simulation environment platforms like Matlab, to ensure that the standards is ready. After that the appropriate hardware was constructed. SDRs allow us to reduce the development cycle.
- *Flexibility*: SDRs can be easily adapt to user needs. Different hardware can be used easily without changing the software and vice versa. For example IEEE 802.11a/g software implementation can be used with a hardware that supports the 2.4 GHz ISM band but also with a hardware that supports the 5 GHz ISM band without any change. On the other hand, with one hardware a user can try different software implementations of the same telecommunication protocol and decide what suits her best.

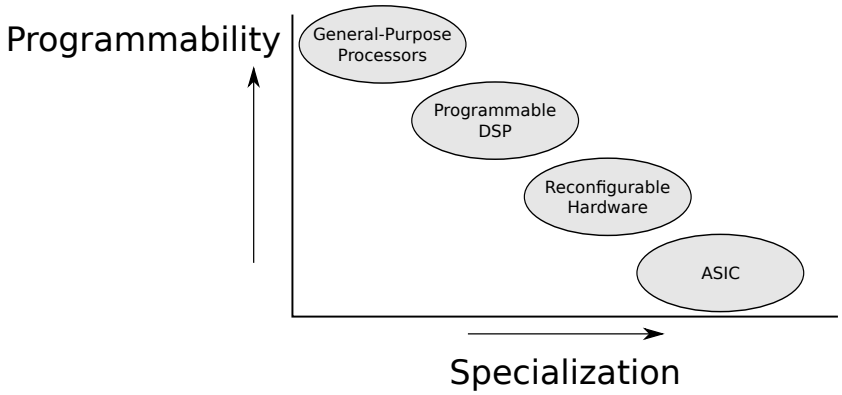


Figure 2.2: DSP implementation spectrum [1]

Of course there are drawbacks too. The biggest problem is the CPU resources that SDRs demands in order to perform the real-time signal processing. The computation cost especially for telecommunication standards with large bandwidths like IEEE 802.11n, can be prohibitive even for a high-end computer. Consequently, all the available technologies that can speed up the signal processing should be exploited. However the significance of this problem is expected to be reduced, as the number of cores and their capabilities are increasing. Modern CPUs are equipped with Single Instruction, Multiple Data (SIMD)

instructions that can drastically improve throughput of signal processing algorithms [11]. Towards this goal, even Graphics Processing Units (GPUs) are used [12] with their superior signal processing capabilities.

Another disadvantage of SDRs is related with the power consumption. SDR systems consume and will consume more energy [13] in contrast of classical solutions. The reason is the high CPU usage that outweighs the benefits of using only one hardware for receiving and transmitting.

2.1 History

Software Defined Radios is a technology that gained high publicity the last few years, despite the fact that the idea was known from early 90's. High demands in computational resources, was the main reason that this technology didn't emerged. Now, with the increasing number of the cores at CPUs that make feasible real time implementations of various telecommunications standards and the popularity that CRs are attracting, more and more people are getting involved with SDRs.

One of the first people that introduced the notion of SDRs as we know it today, was Joe Mitola [14] having also one of the first publications in this area [15]. In his paper he established the necessity of SDRs and described their fundamental parts providing a prototype.

The first attempt for a complete SDR device was the SPEAKeasy project [16] founded by the US Military. The project started at 1991 and demonstrated at 1994. Its main goal to provide interoperability between the different communication methods that are used in warfare, mobility responsiveness and adaptability, key aspects of military operations. Although this devices successfully demonstrated it could not easily maintained nor take advantage of newer processors, due to the three years hardware and software development cycle. Based on the Moore's Law, when SPEAKeasy was ready there were processors available with four times more transistors than the processors used at the beginning of the development phase. Of course SPEAKeasy could not take advantage of this processing power. The reason was the TMS320C40 processor that was used for the signal processing. This processor had its own assembly language making system upgrades practically unfeasible.

It was obvious than in order to take full advantage of SDRs capabilities, software should be completely independent from hardware. This was the main goal of the SpectrumWare project [17], the first complete SDR platform for general purpose CPUs developed in MIT and sponsored by Defense Advanced Research Projects Agency (DARPA). SpectrumWare main focus was the development of a software platform that could run on general purpose processors and conventional operating systems like UNIX, exploiting the fast increasing computational power, multitasking, single-cycle multiply-add capabilities and small development cycles. At this project several issues that even today are worrying SDR developers were taken into account. Meeting the time constraints, parallel execution and caching was some of the design parameters. SpectrumWare managed not only to successfully present a SDR platform, but also create a set of design guidelines for SDR platforms which are leading to low complexity and low cost hardware, adaptability in technological advances, reduced development times and easy upgrades - customization.

Taking into consideration all the limitations that were phased at SPEAKeasy first attempt, the development of SPEAKEasy Phase II[18] started at July 1995. Having as main objectives the easy and fast software re-programming, small size, easy in-field repair and using Commercial-Off-The-Shelf (COTS) software and hardware, development team

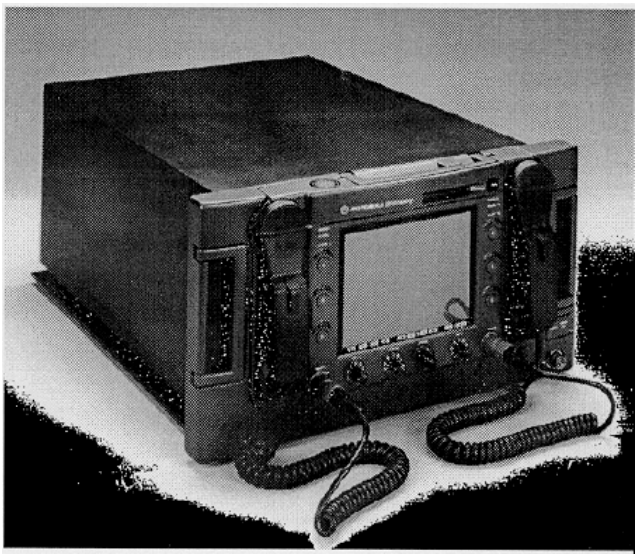


Figure 2.3: Photograph of 2-channel SPEAKeasy II Model Year 1 MBMMR

was able to successfully demonstrate a 2-channel working device only 15 months after the contract initiation. The device which is shown at Figure 2.3, had the size of an average PC many times smaller than its predecessor and could be reconfigured or upgraded easily.

2.2 Software Platforms

In the early days, SDR platforms were just programs performing specific tasks and were highly depended on the hardware too. If the SDR had to used for another telecommunication technology, software should be re-written from ground up. This of course cancels all the benefits that SDRs introduce. The nature of SDRs make them applicable in many different telecommunication technologies. Due to the different operation frequency and bandwidth requirements of these technologies, several different hardware devices may be used. This characteristic imposes many challenges that the software stack has to overcome.

In order a software platform to be efficient and attractive it should be able to deal with a great part of the signal processing without the need of new code. This can be achieved by offering out of the box a set of commonly used operations. Additions, multiplications, Finite Impulse Response (FIR) filters, Fast Fourier Transform (FFT) are some of them that are used in the majority of the telecommunication systems.

On the other hand, each application has its own needs. Software platform should be able to adopt at user needs easily by providing at the developers a convenient and efficient way to write their own modules-plugins that can be attached to the software platform. By combining the built-in features of the platform and easily integrated custom code, developers can create fast and efficiently their applications.

Another key feature of software platforms is the computational resources management and utilization. It is known that some common signal processing operations (e.g. large FIR filters, FFT) are CPU intensive. Also many telecommunication protocols require a high number of samples per second or have tight timing constraints. In order to conform with these needs, software stack should be able to perform signal processing with the lowest

possible computational overhead. To achieve that state of the art algorithms and methods should be used. Furthermore, by exploiting the advantages of parallel programming and multi-core systems computational delay can be minimized.

2.3 Hardware

Even if most of the signal processing is done by software, hardware is a vital component of SDR systems. Hardware is responsible for converting the analog waveforms to digital data, in order software to be able to process them and vice versa. This in general accomplished by an architecture like Figure 2.1 shows. Most of the hardware devices are hosted in small external cases with external power supply. Data transfer is accomplished with USB or Gigabit Ethernet technology. Lately, came up some implementations that managed to reduce even more the form factor of SDR hardware, that is able to fit on a single PCI Express card. Going even further, Lime Microsystems [19] announced the world's first single chip SDR hardware with Analog Devices following after a while with the impressive AD9361 chip [20]. These chips could be possibly placed in hand-held devices, creating a new era in SDR systems.

2.4 The GNU Radio Platform

GNU Radio [21] is open source platform part of the GNU project. It started at 2001 and at the time that these lines were written its latest version was 3.7.4. The core of the platform and the demanding signal processing tasks are performed using C/C++, where tasks like input and output connections are accomplished using Python [22]. Methods written in C/C++ are accessible from Python with the help of Simplified Wrapper and Interface Generator (SWIG) [23]. Moreover, it takes advantage of SIMD instructions that are available in most modern CPUs, using a library called Vector-Optimized Library of Kernels (VOLK). This library is a set of kernels implementing commonly used tasks (e.g. addition, complex and scalar multiplication) using SIMD instructions if the processor supports them. Also, GNU Radio is compatible and supports multiple hardware SDR devices like Universal Software Radio Peripheral (USRP), BladerRF, HackRF and others [24, 25, 26].

One of the most attractive features of this platform is that allows developers to add easily their own modules, extending the capabilities of the platform. Furthermore, the large, active and growing community makes GNU Radio and ideal choice of an SDR platform.

2.4.1 General Description

The approach of GNU Radio signal processing is based on multiple data streams. Each stream may be the output of a processing unit called *block*, either the input to another processing block. Blocks that have only output streams are called *sources* where those that have only input streams are called *sinks*. Each stream is characterized by a data type (e.g. complex, short, float) and a size. Stream of type complex and size 1, means that data in this particularly stream are complex numbers. On the other hand, stream of type complex and size 64 indicates that data of this stream are complex numbers in groups of 64 complex numbers.

Each processing block may have an arbitrary number of input and output streams of any type and size. In order to be able the program and the developer to identify each stream, each block has input and output ports. On every port one stream is assigned.

The restrictions that should be satisfied in every program is that every stream must be connected in an input and output port. In addition, input, output and the stream should have the same data type and size.

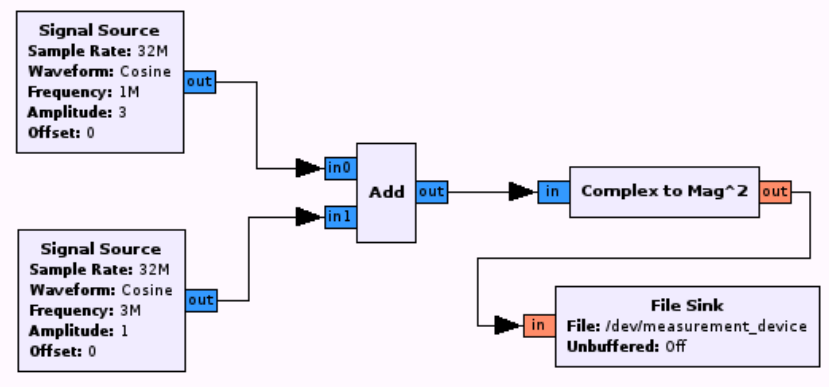


Figure 2.4: A simple GNU Radio flowgraph

The combinations of processing blocks and streams is called *flowgraph*. An example flowgraph of a simple program that computes the sum of two cosine signals and their energy is presented at Figure 2.4. A flowgraph is the graphical representation of the underline scenario that the program should execute. Each flowgraph should have at least one source block. This block will feed with data the rest of the signal processing blocks. Also, as every program produces output data so does the flowgraph of GNU Radio. So again at least one sink block is needed for the processed output data.

2.4.2 The Processing Block

GNU Radio block is the main processing unit of the platform. Combining several blocks in a flowgraph the developer can achieve the desired result. GNU Radio provides some basic signal processing blocks out of the box and leaves the developer free to implement custom blocks. Despite this, all possible blocks may derived from four fundamental block types. Blocks can be classified according to the ratio that consume - produce data into four classes:

1. **Synchronous blocks:** This type of blocks is the most commonly used and easiest to handle. For each available input item the synchronous block will produce equal amount of output items in a $1 : 1$ ratio. Moreover, the synchronous block guaranties that if multiple input ports exist, all of them may have the same amount of data. In other words, if input port 0 has 10 items so does input port 1, e.t.c.

2. **Decimation blocks:** These blocks are like synchronous blocks with a small difference. For every N input items, they produce 1 item. The output ratio is $N : 1$, where N is the decimation factor. Also, in order this block to be activated and be able to produce output, at least N input items should be available at all input ports.

3. **Interpolation blocks:** Interpolation blocks are the exact opposite of decimation blocks. For every input item they produce N output items in a ratio of $1 : N$.

4. **General blocks:** General blocks are the most powerful type of blocks but the most difficult to use. All previous block types are derived from general blocks.

These blocks may produce and consume output and input items in arbitrary ratio. Furthermore, this type of block may consume different number of items from each input port. On the other hand, due to GNU Radio scheduler restrictions the same number of output items should be written in all possible output ports of a block.

Special care should be taken with general blocks, because input port may have different number of available items. It is the responsibility of the programmer to handle it.

Another block type that GNU Radio uses but not mentioned above is the so called *Hierarchical block*. It is derived from the four fundamental block types because it is an indirect block type. It is just a convenience for the developers in order to group several connected blocks into one. This abstraction block during the flowgraph evaluation just expands the internally connected blocks without generating any additional code.

Until now, it was not clear how each block knows how many input items are available and how many outputs can write. This is described at the next paragraphs.

Each block should implement an abstract method. For synchronous, decimation and interpolation blocks this method is presented at Listing 2.1 whereas the corresponding method for general blocks at Listing 2.2. The implementation of these abstract methods, specifies also the processing tasks that the block should execute.

```
1 int
2 work(int noutput_items,
3      gr_vector_const_void_star &input_items,
4      gr_vector_void_star &output_items);
```

Listing 2.1: work() abstract method for synchronous blocks

```
1 int
2 general_work(int noutput_items,
3              gr_vector_int &ninput_items,
4              gr_vector_const_void_star &input_items,
5              gr_vector_void_star &output_items);
```

Listing 2.2: general_work() abstract method for general blocks

Both of the two methods provide the *noutput_items* parameter. This parameter informs the block for the maximum possible output items that the block can produce to every output port at the current method call. *input_items* is a vector of pointers with size equals to the number of input ports of the block. Correspondingly, *output_items* is a vector of pointers with size equals to the number of output ports. Each of these pointers leading to a memory location where the block can read its input items or write the produced output items. Of course, for every input and output port the memory regions are different.

Each invocation of *general_work()* or *work()* method should notify the platform for the number of items that the corresponding block consumed and produced. The produced items are indicated with the return value of these methods. Of course, the return value should be less or equal the *noutput_items* parameter.

On the other hand, the platform should have the knowledge of the items consumed too. In synchronous blocks where there is a fixed and known a priori ratio between consumed and produced items, there is no need to inform explicitly the platform for the processed input items. The platform calculates them from the return value and from the interpolation or decimation rate specified during the construction of the block. However, in general

blocks, the developer should also manually declare how many input items where consumed at the current method call invocation. This can be accomplished with two method. Either with *consume()* method (Listing 2.3) or *consume_ each()* method (Listing 2.4). The first informs the platform about how many items where consumed from all input ports of the block, whereas *consume_ each()* can be used to specify the items consumed from a specific input port.

```
1 void
2 consume(int which_input, int how_many_items)
```

Listing 2.3: consume() method

```
1 void
2 consume_each(int how_many_items)
```

Listing 2.4: consume_each() method

2.4.3 Scheduler and Flowgraph

At the previous section the platform mechanism of tracking consumed and produced items in every stream was described. But how and who is responsible for the invocations of the *general_ work()* and *work()* methods of the blocks that apart a flowgraph? This is the task of the GNU Radio scheduler that decides under specific circumstances when and which blocks should be dispatched in order to process the available data.

To accomplished that the scheduler should have a bird eye view of the blocks and the connections that are part of the program. This is performed with the use of a flowgraph. The flowgraph keeps track of all the blocks and the connections between them. With this information data dependencies can be found easily. For example as shown in Figure 2.4, the block that adds the two cosine signals has three dependencies. In order to perform the desired sum, samples from the two signal source blocks should have been placed at the corresponding streams. Moreover the output stream should have enough space to accommodate the resulting samples.

When a flowgraph starts executing, the scheduler identifies the involved blocks. For each block a separate thread is created and dispatched. Each thread checks the available input items on every input port. If there are enough items, *general_ work()* or *work()* method is called. Otherwise the thread stops its execution and blocks waiting for data. Moreover, after a successful *general_ work()* or *work()* call some output items where produced and propagated at the output ports. By checking the connections, scheduler informs the blocks that have as input the corresponding output ports, denoted as *readers*. If the readers' requirements are met, they will unblock, otherwise they will block again waiting for another notification. The same process is repeated while there are input data or until the end of the program.

2.4.4 GNU Radio Companion

GNU Radio is shipped together with a user friendly graphical interface, the GNU Radio Companion (GRC) (Figure 2.5). This tool gives the flexibility to create easily flowgraphs by drag and drop blocks into the workspace and apply the appropriate connections and parameters.

Each flowgraph is stored in an Extensible Markup Language (XML) style form. When a flowgaph should be executed the GRC converts the flowgraph into a Python executable by generating the appropriate code. After that, the program is executed as a normal Python application.

A great advantage of this tool is that also custom user defined blocks can be integrated to the graphical interface allowing the developers to easily use and distribute their blocks. Furthermore, during the flowgraph design phase, GRC performs checks in order to keep the flowgraph consistent. When the user connects blocks that incompatible due to data or stream size mismatch, the tool annotates the problematic connection indicating the error.

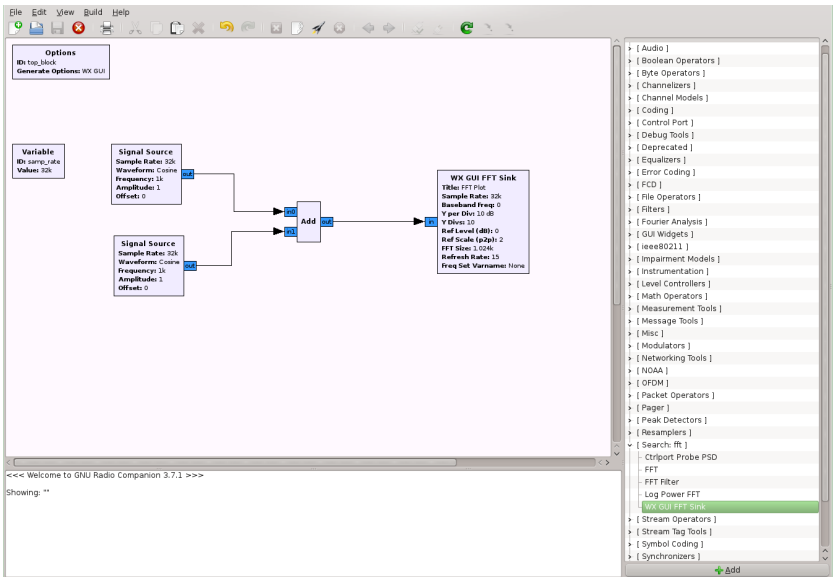


Figure 2.5: The GNU Radio Companion tool

Chapter 3

IEEE 802.11a Physical Layer Software Implementation

The purpose of this chapter is to describe and review the development process of a PHYsoftware implementation of the popular IEEE 802.11a [27] standard and its variants IEEE 802.11g and IEEE 802.11p.

Firstly, some fundamental aspects of the IEEE 802.11a PHY will be presented, followed by some general information related with the software development process. Then, all the necessary operations required for transmit data are described among with the corresponding software solutions in Section 3.3. After the transmit process, the more demanding and complicated receive phase follows. The chapter concludes providing a method, efficiently using the FFT in order to avoid Sample Rate Conversion (SRC) or to support easily additional channel bandwidths.

3.1 IEEE 802.11a Standard

The IEEE 802.11a standard is a set of specifications for MAC and Physical layer suitable for Wireless Local Area Network (WLAN). The PHY of IEEE 802.11a is based on OFDM and operates at the ISM unlicensed bands of 2.4 GHz and 5 GHz. Utilizing 20 MHz channel bandwidth it provides 6, 9, 12, 18, 24, 36, 48 and 54 Mega Bits per Second (Mbps) data rates. To achieve that 64 OFDM sub-carriers are used from which the 52 are non-zero, modulated with Binary or Quadrature Phase Shift keying (BPSK, QPSK), 16- or 64-Quadrature Amplitude Modulation (QAM). From the 52 non-zero OFDM sub-carriers the 48 are used for user data transferring. The 4 sub-carriers left are used as pilots, helping the receiver to perform channel estimation or tracking phase shifts. Furthermore, the system provides forward error correction coding with three different coding rates, 1/2, 2/3 and 3/4.

By reducing the channel bandwidth, the system can provide several different data rates. Using 10 MHz channel bandwidth, 3, 4.5, 6, 9, 12, 18, 24, 27 Mbps data rates can be achieved. Respectively, data rates of 1.5, 2.25, 3, 4.5, 6, 9, 12 and 13.5 Mbps can be reached with 5 MHz channel. These half- and quarter-clocked operations have the same number of OFDM sub-carriers but double and quadruple OFDM symbol timing from the normal 20 MHz channel bandwidth operation. At Table 3.1 all the possible combinations of data rates, modulations, e.t.c are presented.

802.11 PHY is responsible to transmit over a wireless channel data derived from the

Table 3.1: 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 (Mbps) (20 MHz channel spacing)**	**Data rate (Mbps) (10 MHz channel spacing)**	**Data rate (Mbps) (5 MHz channel spacing)**
BPSK	1/2	1	48	24	6	3	1.5
BPSK	2/3	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

MAC layer. Respectively, data that obtained over the air from an 802.11a receiver are propagated towards the MAC layer. This is accomplished by the Physical Layer Convergence Procedure (PLCP), which specifies the method that IEEE 802.11 PLCP Service Data Units (PSDUs) will be mapped in a suitable format, the so called PLCP Protocol Data Units (PPDUs), in order Stations (STAs) to be able to exchange user and management information.

The IEEE 802.11a MAC layer is designed to be as much as possible independent to the PHY layer. Yet, due to the fact that MAC layer keeps several state machines that affect or be affected by the PHY, a bidirectional information exchange between the two layers is necessary. This is accomplished with a set of parameters that are described in Tables 3.2 and 3.3.

Table 3.2: TXVECTOR parameters

Parameter	**Value & Description**
LENGTH	1-4095, the length in octets of the PSDU
DATARATE	6, 9, 12, 18, 24, 36, 48, and 54 Mbps for 20 MHz channel spacing, 3, 4.5, 6, 9, 12, 18, 24, and 27 Mbps for 10 MHz channel spacing, 1.5, 2.25, 3, 4.5, 6, 9, 12, and 13.5 Mbps for 5 MHz channel spacing respectively. This parameter indicates the OFDM data rate that is going to be used for the transmission of the corresponding PPDU
SERVICE	Initialization sequence for the data scrambler. (See Section 3.3.1)
TXPWR_LEVEL	1-8, the power level that PHY should transmit the corresponding PPDU

3.1.1 Frame Format

At Figure 3.3 the format of a PPDU frame is presented. A PLCP preamble with duration of 12 OFDM symbols is used in front of the frame for time and frequency synchronization. The PLCP header that follows contains the SIGNAL and the SERVICE fields. After that a variable number of data that are derived from the MAC construct the PSDU field. The end of the frame is indicated by 6 zero bits denoted as the TAIL field. Also, due to the fact that SERVICE, PSDU and TAIL segments should be an integer multiple of N_{DBPS},

Table 3.3: RXVECTOR parameters

Parameter	Value & Description
LENGTH	1-4095, the length in octets of the PSDU
DATARATE	6, 9, 12, 18, 24, 36, 48, and 54 Mbps for 20 MHz channel spacing, 3, 4.5, 6, 9, 12, 18, 24, and 27 Mbps for 10 MHz channel spacing, 1.5, 2.25, 3, 4.5, 6, 9, 12, and 13.5 Mbps for 5 MHz channel spacing respectively. This parameter indicates the OFDM data rate that was used for the reception of the corresponding PPDU
RSSI	0 - RSSI maximum. The Received Signal Strength Indication (RSSI) measured by the antenna that was used for the reception of the current PPDU

appropriate number of pad bits may be inserted after the TAIL segment, constructing the PAD field. SERVICE, PSDU, TAIL and PAD segments for simplicity are also mentioned as the DATA segment.

3.1.1.1 The PLCP Preamble

The PLCP preamble has a duration of 12 OFDM symbols which are the same for every IEEE 802.11a frame that is transmitted. The preamble is constructed by 10 repetition of the short training sequence denoted as t_1 - t_{10} at Figure 3.7. Then, the cyclic prefix of the long training sequence ($GI2$) and two repetitions of the long training sequence ($T1$ and $T2$) are following.

The purpose of the PLCP preamble is to enable the receive device to detect the presence of a frame. After that and based on the signal properties of the preamble, receiver perform Automatic Gain Control (AGC), time-frequency synchronization and channel estimation in order to be able to successfully decode the contents of the frame. The exact procedure is discussed at Section 3.4.

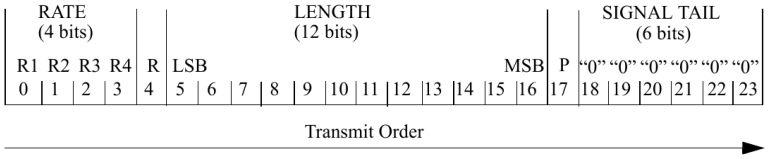


Figure 3.1: The SIGNAL field

3.1.1.2 The PLCP Header

The PLCP header contains vital information for the frame. The header has a constant size of 5 bytes and is composed by two major segments, the SIGNAL and the SERVICE as can be shown at Figure 3.3.

The SIGNAL field is 3 bytes long and has duration of one OFDM symbol. For every frame transmitted, the modulation scheme that is used for this symbol is Binary Phase-shift keying (BPSK) with coding rate R=1/2. Furthermore, the coded data of this field are not scrambled. SIGNAL segment is also contains some sub-field that are illustrated at Figure 3.1. RATE is a 4-bit field that contains the data rate that was used to transmit

the frame. The appropriate bit assignment for each rate can be derived from Table 3.4. If the data rate is known, the modulation and the coding rate can be found easily based on Table 3.1. The 12-bit long LENGTH sub-field contains the size of the PSDU in bytes that the MAC layer requested from the PHY to transmit. It is this field that specifies the maximum allowed PSDU size that can be transmitted to $2^{12} - 1 = 4095$ bytes. The Parity 1-bit sub-field is used for an early and coarse integrity check. It is the even parity of the 0-16 bits of the SIGNAL segment. The rest bits, reserved bit 4 and the SIGNAL TAIL bits are explicitly set to zero.

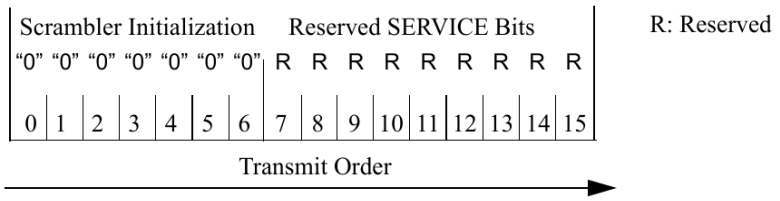


Figure 3.2: The SERVICE field

As can be seen from Figure 3.2, SERVICE is a 16-bit long field that helps the receiver to synchronize its descrambler. To accomplish this bits 0-6 are set to zero prior the scrambling process, while the rest of the bits are reserved for future use and should be ignored. The contents of this field should be scrambled and encoded with the modulation scheme that was specified at the RATE field of the SIGNAL segment.

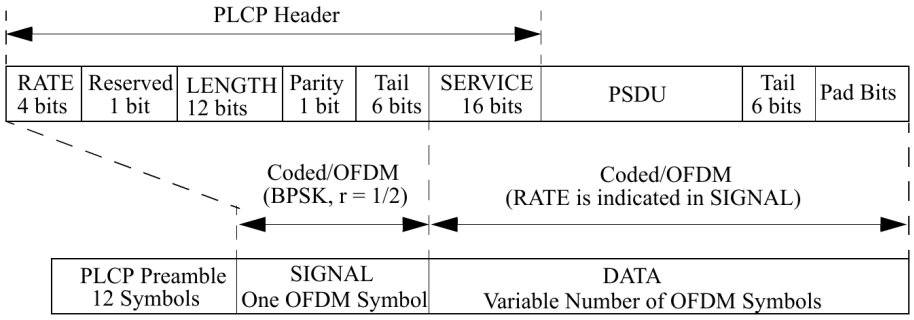


Figure 3.3: PPDU frame format

3.2 General Software Implementation Information

Before proceeding to examine the implementation details of the IEEE 802.11a transmitter and receiver, it will be necessary to present some general aspects of our implementation.

As a digital telecommunication system, 802.11a includes several bit operations. When have to deal with data operations most SDR developers handle bits as bytes. In other words, each single bit is represented as a byte with its Least Significant Bit (LSB) to correspond to the desired bit state. This approach simplifies the logic of each block that handles bit streams. The key problem with this method is that increases by a factor of 8 the data that should be transferred between blocks. This of course is not acceptable

by a real time system, so our approach handles bits in a per byte basis without wasting resources. However, our approach introduces challenges too. Bit operations become more complex and tricks should be applied when bits streams can not be handled as bytes. For example, how coded bits of an OFDM symbol that uses BPSK modulation scheme and $R = 2/3$ coding rate are handled? As can been seen from Table 3.1, each OFDM symbol with this specific modulation scheme is not an integer multiple of a byte.

Regarding the samples' data, the implementation uses complex numbers that are constructed by two interleaved single precision floating point numbers. As the commodity of Analog to Digital Converters (ADCs) and DACs have 16-bit resolution, a single precision floating point number has more that enough accuracy. Many argue that this redundant accuracy introduces unnecessary overhead and instead the short integer data type could be used that has 16-bit resolution. Although this statement is true, our approach uses floating point numbers to exploit the benefits of SIMD instructions without the need of casting from a type to another.

Using an SDR platform imposes limitations too. A well-known example of such limitations is the following. GNU Radio scheduler does not allow block output to be an input of the same block (feedback). This limitation is easily bypassed with buffering techniques that can be applied internally in the block. Furthermore, the scheduler restricts flowgraphs to be sequential, not allowing connections that create loops. An output of a block can not be the input of any block that is connected before. This of course makes many decision making algorithms, like feedback control loops, very hard to implement. To overcome this problem our approach uses asynchronous blocking calls with the help of thread-safe queues. By reading or writing a message at a queue blocks can communicate asynchronously. The blocking nature of these queues simplifies greatly the logic of processing blocks that use them.

However, there are difficulties introduced by the platform that are more severe. As discussed previously at Section 2.4, each block runs on its own thread. This parallel way of execution creates thousands of different scenarios of how and when blocks are executing. At each execution scenario blocks are producing different number of output items. If all blocks are synchronous blocks this is not a problem, as the 1:1 input-output ratio simplifies a lot the execution. The scheduler ensures that output item z_i is a function of inputs x_i and y_i. On the other hand, this is not true for general blocks and as expected a complicated implementation like IEEE 802.11a makes use of general blocks. As Figure 3.4 illustrates, RSSI measurement synchronous block computes constantly the RSSI of the received signal that should be reported at the MAC layer either before a transmission for the Carrier Sense Multiple Access (CSMA) mechanism, either during the reception of a packet. The Packet Detector is a general block that propagates data if and only if a packet is detected. The following block is the Packet Decoder that decodes the data and propagates them at the MAC layer including an RSSI value for the corresponding packet. The obvious problem with the flowgraph presented is that the RSSI value that is received by the Packet Decoder block, may not correspond to the actual RSSI value of the packet currently decoded. This happens because there is no more a 1:1 input-output ratio at the whole flowgraph due to the use of a general block.

A smarter way to avoid this problem is the following. The RSSI block output could be another input of the Packet Detector as illustrated in Figure 3.5. With this patch, each packet has the right RSSI value and also the CSMA mechanism get the right RSSI indication. But this solution comes with a great cost. Packet Detector block has now one more input and another output. This increases the the computational cost for this block, due to the scheduler additional operations that are required.

For the purposes of this work, real SDR devices where used and the communication

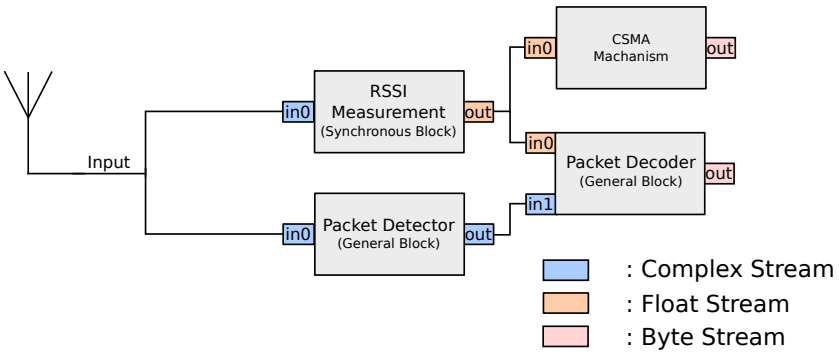


Figure 3.4: A Problematic Flowgaph Scenario

between computer hosts were performed using the air as the wireless medium and utilizing both the unlicensed ISM and Digital TV radio frequencies. As far the SDR hardware is concerned, the Noctar [28] PCIe card was used. This card was shipped with a very simple and basic driver that of course did not met the needs of such a complicated protocol like IEEE 802.11a. For these reason we have developed our own high performance driver for this SDR device.

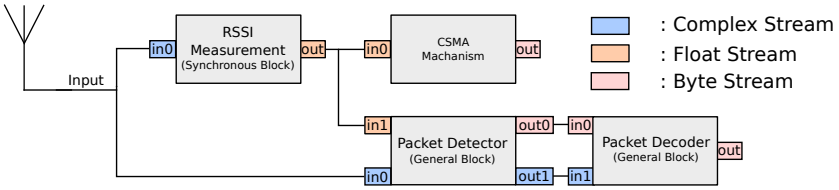


Figure 3.5: Solution to the problematic flowgraph of Figue 3.4

3.3 Transmit Chain

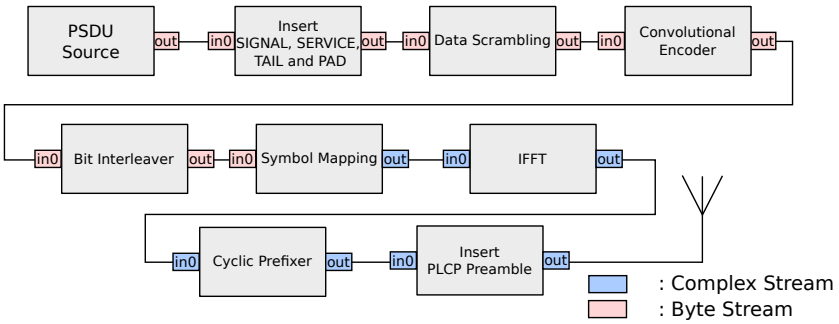


Figure 3.6: Flowraph of the IEEE 802.11a Transmitter

After discussing the key aspects of the IEEE 802.11a frame structure, this section moves on discussing the procedure of transmitting data. Figure 3.6 illustrates the flowgraph of

the transmitter. As this flowgraph reveals, transmit is a quite straightforward procedure. The procedure can be classified into several basic and sequential steps. These are:

1. After obtained the PSDU and the other vital information from the MAC layer, compute the PLCP header. The 4 bits of the RATE field are chosen based on the Table 3.4. Also based on the SERVICE field of the header initialize the sequence of the Scrambler.

2. Convolutionally encode the contents of the RATE, LENGTH, as well as the reserved, parity and tail bits of the PLCP header using coding rate R=1/2. After that perform bit interleaving. The resulting 48 bits should be BPSK mapped, constructing the so called SIGNAL symbol (Figures 3.3, 3.7) allowing the receiver to extract vital information (e.g. modulation, length) about the frame currently receiving. The contents of the SIGNAL OFDM symbol should not be scrambled.

3. Based on the modulation scheme, compute the coded bits per OFDM symbol (N_{CBPS}), as well as the data bits per OFDM symbol (N_{DBPS}) and the number of bits that each subcarrier carries (N_{BPSC}). Their values can be obtained from the Table 3.1.

4. Append to the SERVICE field, the PSDU and 6 zero TAIL bits. Because the resulting size should be an integer multiple of N_{DBPS} bits, an appropriate number of zero pad bits is appended. These all together, are denoted as the DATA symbols (Figure 3.3).

5. Perform scrambling of the DATA field. This is done by XORing the scrambling sequence with the contents of the DATA segment. After the scrambling procedure the 6 TAIL bits are explicitly set to zero again.

6. The scrambled data should be convolutionally encoded, based on the coding rate that the corresponding modulation indicates. Coding rates 2/3 and 3/4 are composed by performing appropriate puncturing at the output bits of the encoder with coding rate R=1/2.

7. Encoded bits are divided into groups of N_{CBPS}. In each group perform bit interleaving with a pattern constructed based on the modulation scheme.

8. Interleaved bits are divided in groups of N_{BPSC}. Each group is mapped into a complex number regarding the modulation scheme. Figure 3.19 shows how the conversion is done. These complex numbers are called constellation points. Note that the SIGNAL field data should be BPSK modulated regardless the modulation scheme of the DATA segment.

9. Constellation points shall be placed in the appropriate sub-carriers. As mentioned before, IEEE 802.11a uses 64 sub-carriers in total. 48 of them are used for data, 4 are pilot carriers and the rest are zeroed.

10. After the placement of the constellation points at the right sub-carriers, the Inverse Fast Fourier Transform (IFFT) is computed. The output is 64 complex number representing the signal samples in the time domain.

11. For each group of 64 time domain samples, a cyclic prefix of 16 samples is placed before. Cyclic prefix samples are the last 16 samples of the IFFT output.

12. Produce the PLCP preamble. This is composed by 10 repetitions of the short training sequence (denoted as t_{1-10} in Figure 3.7) followed by two repetitions of the long training sequence ($T1$ and $T2$) along with a cyclic prefix (denoted as GI2).

13. Append to the PLCP preamble the coded cyclic prefixed SIGNAL and DATA segment and transmit them using an SDR device.

This section has analyzed briefly the transmit procedure. The following sections will present some of the transmit tasks in more detail.

Table 3.4: Contents of the RATE field

R1-R4	**Rate (Mbps) (20 MHz channel spacing)**	**Rate (Mbps) (10 MHz channel spacing)**	**Rate (Mbps) (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

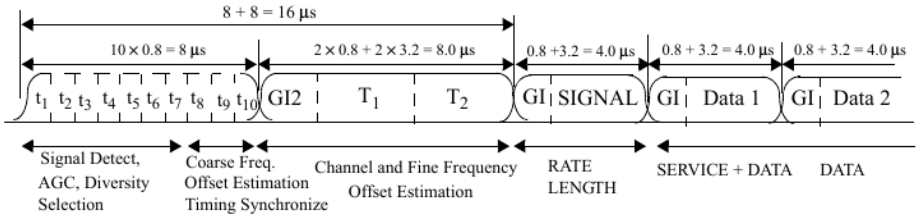


Figure 3.7: OFDM training structure

3.3.1 Data Scrambling

Bit streams produced from real life applications (e.g text, voice, video) have often long sequences of same bits. These can cause undesirable effects on OFDM telecommunication systems because the power is concentrated in a small fraction of the spectrum [29]. Scrambling is the procedure that its goal is to prevent the appearance of long sequences consisting of '1' or '0' only. To do that, the transmitter transforms input data to pseudo-random data based an a certain function. The receiver uses the opposite procedure to get the actual data. Scrambler accuracy depends on the ability of the scrambling function to produce a random output based on the input and avoid sequences of same bits.

There are two basic types of data scramblers:

1. Multiplicative or self-synchronizing scramblers

2. Additive of synchronous scramblers

Multiplicative scramblers produce scrambled output by multiplying the input stream with the scrambler transfer function in the z-domain. Their key characteristic is that they do not based on an initial seed word for frame synchronization, so they also referred as

self-synchronizing scramblers [30]. A graphical example of a multiplicative scrambler and descrambler is depicted at Figures 3.8a and 3.8b respectively.

On the other hand, additive scramblers perform modulo-two operation between the input data and a pseudo-random sequence. This sequence is generated using an Linear Feedback Shift Register (LFSR) and a characteristic polynomial. The initial state of the LFSR known also as *scrambler seed*, should be set for every new frame, that why this type of scramblers are called synchronous. In most cases this initial seed is provided into an unscrambled part of the frame header. A very convenient feature of additive scramblers is that scrambling and descrambling can be performed using exactly the same algorithm. IEEE 802.11a uses an additive scrambler illustrated in Figure 3.9.

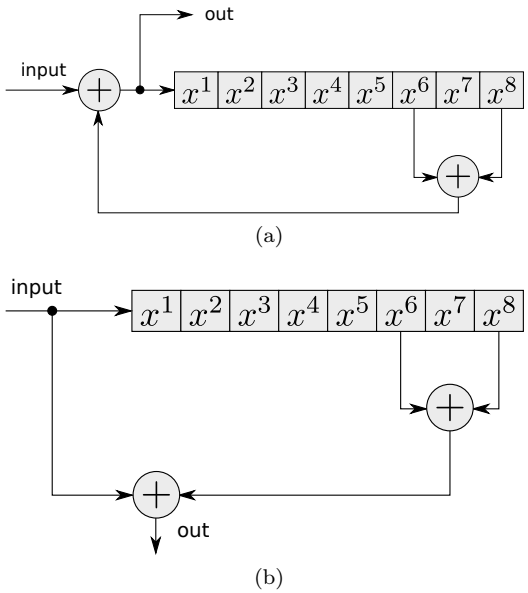


Figure 3.8: Example of multiplicative: (a) Scrambler (b) Descrambler

IEEE 802.11a performs scrambling by XORing DATA segment data bits with a frame synchronous scrambling sequence that is generated using the generator polynomial of Equation (3.1). This generator will produce a repeated 127-bit sequence. After performing the scrambling, TAIL bits should be explicitly set to zero again. This is done to reset the state of the Viterbi decoder (Section 3.4.10.2) at the receiver, in order to be ready for the next frame reception.

$$S(x) = x^7 + x^4 + 1 \tag{3.1}$$

The initial state of the scrambler is set randomly. When transmitting the 7 LSB of the SERVICE field are set to zero, in order the receiver to be able to estimate the initialization sequence of the scrambler.

3.3.1.1 Software implementation

Computing the XOR of the scrambling sequence and the data bits of the DATA field is a lightweight and easy procedure. On the other hand, due to the high data rates that IEEE

802.11 standard operates, the computation of the scrambling sequence is able to consume a notable amount of CPU resources. Based on the fact that we handle the bit stream in a per byte basis, arises also the problem that the 127-bit scrambling sequence is not an integer multiple of a byte. To overcome these problems our implementation is using a pre-calculated scrambling sequence.

$$LCM(127 \text{ bits}, 1 \text{ byte}) = 1016 \text{ bits} = 127 \text{ bytes} \tag{3.2}$$

As the scrambling sequence is repeated every 127 bits, we can construct a bigger sequence from multiple 127-bit sequences. The size of the resulting sequence can be derived from the Least Common Multiple (LCM) of the scrambling sequence size and the number of bits that each byte has (Equation 3.2). The new sequence continues to have the repetition property that the 127-bit sequence has, so only a simple modulo operation is needed for wrap around the sequence if the DATA field is greater than the sequence size. XOR operation is simplified as it is done in a per-byte basis without the need of tricky and hard maintainable bit-wise operations. With this technique, further performance improvements can be achieved utilizing SIMD instructions of the modern CPUs. For more information refer to Section 5.1.

According to the standard the 127-bit sequence should be produced having a pseudo-random initial sequence. Unfortunately, due to the fact that a pre-computed sequence is used, the initial state of the pseudo-random sequence is the same for every frame transmission. These is the only drawback of our approach, letting the performance of the transmitter relying on the performance of the scrambler of the standard.

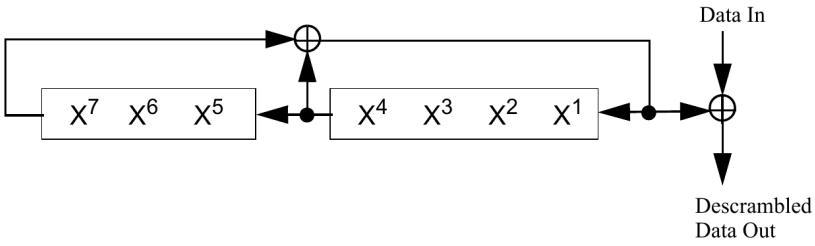


Figure 3.9: IEEE 802.11a Data Scrambler

3.3.2 Convolutional Encoder

Communication over wireless channels introducing many impairments that a telecommunication system should deal with. Noise, interference from other systems and multi-path phenomena are some of them. In order a system to be able to achieve reliable data transfer in such rough environments, redundant information is added at the transmitted data. These extra bits of information are used from the receiver to recover from any possible errors that may occur, a procedure known as Forward Error Correction (FEC) [31, 32]. The process that produces these redundant bits is known as Convolutional Encoding.

Main characteristic of convolutional encoders is that they have memory. Output bit is a function not only from the current but also from previous input bits. The number of memory stages (K) each holding one input bit, is known as *constraint length*. Coding rate (R) is the ratio of the number of input bits (n) to the number of output bits (m)

that are produced from the encoder. It is common in the bibliography to represent the properties of a convolutional encoder using the (m, n, K) notation. Furthermore, the encoder has m modulo-2 adders and m generator polynomials ($g_i = D$), one for each adder. Generator polynomials specify which memory stages the corresponding adder should take into consideration. For example, polynomial generator $g = 5_8 = 101_2$ indicates that the adder will use the bits of memory stages 0 and 2.

Figure 3.10 presents a simple convolutional encoder $(2, 1, 3)$ with $K = 3$ and coding rate $R = 1/2$. The corresponding polynomial generators are $g_0 = 5_8$ and $g_1 = 7_8$. Initially all memory stages are set to zero. A possible output for a specific input can be found at Table 3.5. The resulting coding bits are produced by interleaving the output bits returned from **A** and **B**. Note that LSBs are transmitted first.

According to the IEEE 802.11a standard, after scrambling the contents of SIGNAL and DATA fields (Figure 3.3) should be encoded. For the SIGNAL field coding rate $R = 1/2$ is used whereas for the rest of the data, the corresponding coding rate based on the data rate is used (Table 3.1). Note that the memory stages of the encoder after the encoding of the SIGNAL field are reset again to zero.

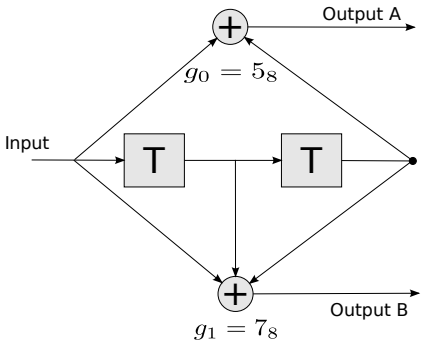


Figure 3.10: A simple Convolutional Encoder with K=3 and R=1/2

Table 3.5: Output of the convolutional encoder of Figure 3.10 for a specific input

Input sequence	**Output**
1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 Transmit Order →	A: 1 0 1 1 1 1 0 0 B: 1 1 1 1 0 0 0 1

IEEE 802.11a uses a widely used encoder with $K = 7$ and $R = 1/2$ (Figure 3.11). The corresponding polynomial generators are $g_0 = 133_8$ and $g_1 = 171_8$. Moreover, the standard supports and two additional rates $R = 2/3$ and $R = 3/4$ by performing puncturing at the outputs of the encoder. For more information about puncturing refer to Section 3.3.2.1.

3.3.2.1 Puncturing

In enviroments with low Signal to Noise Ratio (SNR), systems prefer to use more conservative coding rate that produce more redundant bits, whereas in better SNR environments the coding rate can be increased in order to achieve higher data rates.

Different coding rates can be achieved by performing puncturing at the outputs of the

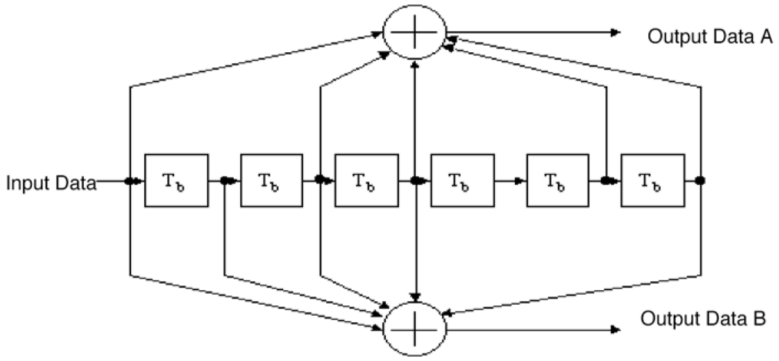


Figure 3.11: Convolutional encoder of IEEE 802.11 with K=7 and R=1/2

modulo-2 adders of the convolutional encoder. Puncturing omits certain output bits of an encoder to achieve greater coding rate.

For example assume the following scenario:

- A $(2, 1, 3)$ convolutional encoder is used with coding rate $R = 1/2$.
- For x input bits, $2 \times x$ output bits are produced from the encoder.
- Omit every fourth output bit of the encoder. By doing this the system transmits 3 coded bits for every 2 bits of input data. This is coding rate $R = 2/3$. It is obvious that these coding rate transfers more real data, achieving higher data rates.

Respectively the receiver inserts dummy bits at the corresponding places of the punctured bits, before the decoder process the received data.

IEEE 802.11a supports except the basic $R = 1/2$ rate, also the $R = 2/3$ and $R = 3/4$ rates by performing puncturing. The puncturing patterns can be obtained at Figures 3.12 and 3.13.

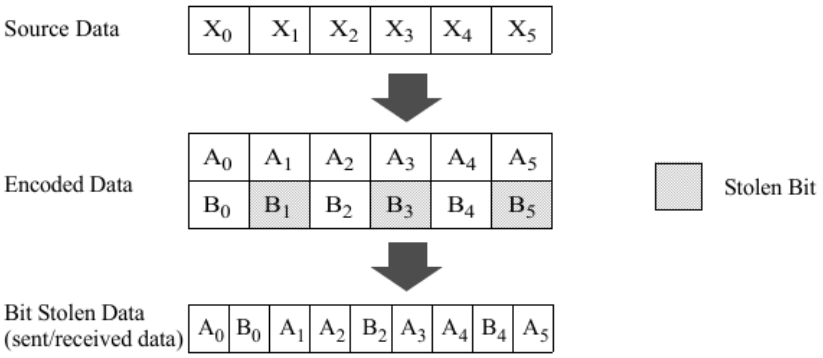


Figure 3.12: Punctured Coding $R = 2/3$

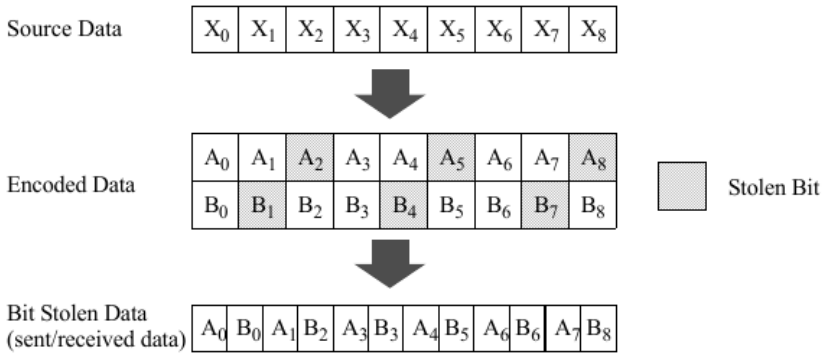


Figure 3.13: Punctured Coding $R = 3/4$

3.3.2.2 Software Implementation

Hardware implementations of convolutional encoders are straightforward and easy. This unfortunately is not true for software, where convolutional encoding can become a quite demanding process due to the great number of bitwise, modulo and division operations that required. Especially for the case of modulo and division, it is known that they require more cycles related to bitwise instructions. Furthermore, the fact that our approach handles the bit stream in a per byte basis also introduces additional complexity.

In order to overcome these problems our approach tries to minimize modulo and bit counting operations using pre-calculated values in lookup tables where possible. Also, in order to keep the number of processing blocks of our flow-graph low, our convolutional encoder performs puncturing, if needed, in parallel with the encoding of the input stream.

Our encoder uses a static lookup table that contains the number of the bits that are 1 for every possible value of a byte, modulo 2 (Listing 3.1). Obviously the size of the table is $2^8 = 256$ elements. Also the number of memory stages that the encoder of IEEE 802.11a uses ($K = 7$), can fit in a byte. By omitting one bit from this byte and by performing bit shifting we can emulate the memory stages of the encoder as a shift register. To compute the output of the g_0 and g_1 polynomial generators respectively, the bitwise AND, of the memory stages byte and the polynomial value is performed. The result is used as index for the lookup table, which eventually will give the appropriate output of the corresponding polynomial generator. The above procedure is summarized in Figure 3.14.

```
1  static const unsigned char lookup_table[256] =
2  { 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0,
3    1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1,
4    1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1,
5    0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0,
6    1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1,
7    0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0,
8    0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0,
9    1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1,
10   1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0,
11   1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0,
12   0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1,
13   0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1,
14   0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0,
15   1, 1, 0, 0, 1, 0, 1, 1, 0 };
```

Listing 3.1: Look-up table with the number of 1 bits modulo-2 for each possible byte

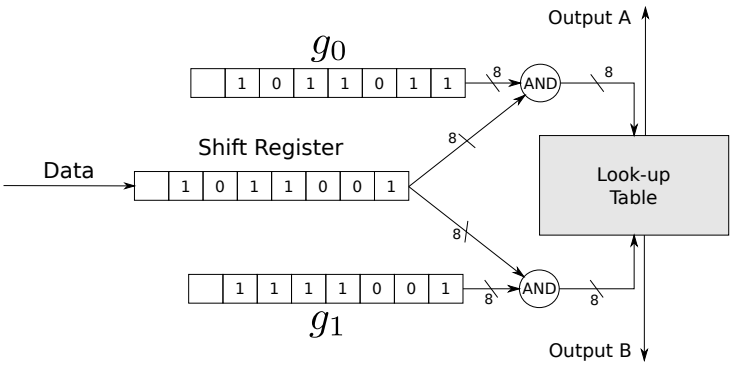


Figure 3.14: Software implementation of the IEEE 802.11a encoder using look-up table

3.3.3 Bit Interleaving

Bit interleaving is a common characteristic of digital telecommunication systems that helps to improve the performance of FEC algorithms. For this reason interleaving is performed after the convolutional encoding. Error correction algorithms like the Viterbi decoder [31], can correct a certain number of sequential false symbols. Due to the fact that errors caused by most wireless channels are bursty, interleaving techniques are used to randomize the transmitted coded bits in order the decoder to be able to recover from these type of errors. Table 3.6 presents an example that clearly shows the contribution of interleaving at digital telecommunication systems. This simple example reveals the necessity of interleaving. Without it FEC algorithms could not be able to correct the faulty received symbols.

The way that bits are interleaved is based on a set of parameters that depend on the channel characteristics on which the system operates in order to overcome the impairments that are introduced. One basic characteristic of a channel is whenever it is fast or slow fading. In fast fading channel the frequency response changes rapidly and one can assume that for every OFDM symbol transmitted, the channel impulse is different. On the other hand, the frequency response of a slow fading channel remains the same for multiple OFDM symbols. Another fundamental characteristic of a channel is, if it is flat or frequency selective fading. In flat fading channels the frequency response is the same for the whole signal bandwidth, whereas in frequency selective channels the frequency response of the channel is different over the bandwidth area of the signal. For an OFDM system, this type of channels impose different frequency response for each OFDM sub-carrier. These channel characteristics specify the type of the interleaver that will be used by the system.

Table 3.6: Example of using interleaving at a transmission over a channel with bursty errors

Without Interleaving	
Input Sequence	aaaaabbbbbccccddddeeeeeffffff
Sequence with Bursty Errors	aaaa_____bcccccdd_____eeffffff
Transmission With Interleaving	
Input Sequence	aaaaabbbbbccccddddeeeeeffffff
Interleaved Sequence	abcdefbcdefacdefabdefabcefabcd
Sequence with Bursty Errors	abcd_____efacdefab_____cefabcd
Received Sequence after Deinterleaving	aaa_ab_b_bc_cccd_d_d_ee_e_ff_f

There are two major interleaver designs widely used in modern digital systems, block and convolutional interleavers [33]. The first design operates on a number of bits at a time, called block. The size of the block is also known as interleaving *depth*. The processing at each block is completely independent from previous blocks. In most cases, input bits are placed in columns while output bits are read row by row or vice versa (Figure3.15). There are also block interleavers that do not follow this convention and choose the output bits from the block based on a permutation function.

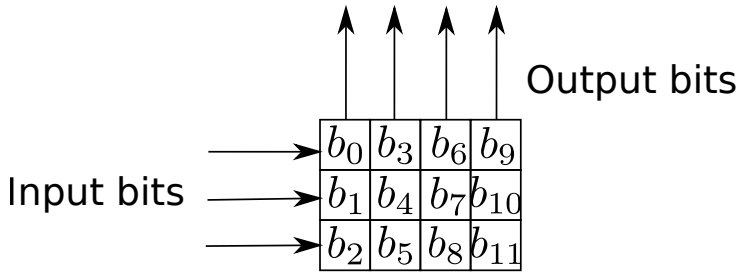


Figure 3.15: A 3×4 block interleaver with $depth = 12$

On the other hand, convolutional interleavers split the input bit stream into different paths, each with a different delay. At each cycle, the interleaver chooses at which path should shift an input bit. As output of the interleaver, the result of the corresponding path chosen is used. (Figure 3.16)

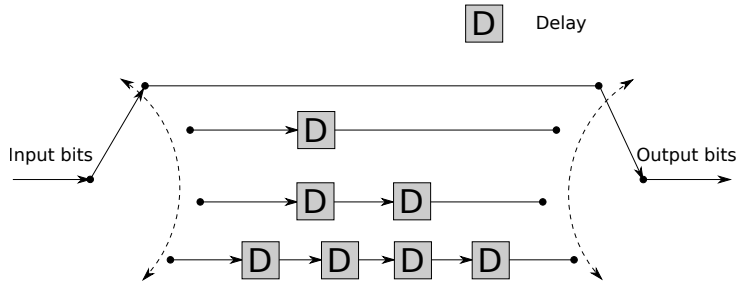


Figure 3.16: A simple convolutional interleaver

WLAN systems like IEEE 802.11a, assume channels that they operate are slow fading based on the fact that each frame has a small duration. Furthermore, due to the large bandwidth that such systems utilize, channels also are considered frequency selective [34]. Such channels characteristics and the bursty nature of their transmission make ideal the use of block interleavers.

3.3.3.1 IEEE 802.11a Interleaver

As mentioned before, IEEE 802.11a uses a block interleaver. The interleave depth is equal to the number of coded bits (N_{CBPS}), that each OFDM symbol carries. So according to the Table 3.1, BPSK, Quadrature Phase-shift keying (QPSK), 16-QAM and 64-QAM have interleaving depths of 48, 96, 192 and 288 bits respectively. The fact that the interleaving depth is the same with the OFDM symbol size is not a coincidence. In the previous section, the characteristics of the wireless channels where discussed. IEEE 802.11 assumes that

the channel is slow fading, thus for each OFDM symbol of a frame the channel response should be the same. Based on this, setting an interleaving depth greater than the coded data each OFDM symbol carries, would not give any performance advantage. On the other hand, large interleave depth values increasing latency.

IEEE 802.11a interleaver performs two-step permutation. The first permutation ensures that adjacent coded bits are mapped into nonadjacent sub-carriers of the OFDM symbol. This increases the performance of FEC, because IEEE 802.11a operates in frequency selective channels. This means that if the data of a sub-carrier are corrupted due to the channel impairments, with high probability the data of the adjacent sub-carriers will also be corrupted. The second permutation ensures that adjacent coded bits are mapped alternately onto less and more significant bits of the constellation. This prevents the system from long runs of low reliability bits. Below is presented how the first and the second permutation is accomplished. The index of the coded bit before the first permutation is denoted as k, whereas the index after the first and before the second permutation is denoted as i. Finally, j is the index after the second permutation.

The first permutation can be derived by the equation

$$i = (N_{CBPS}/16)(k \bmod 16) + floor(k/16) \qquad k = 0, 1, \ldots, N_{CBPS} - 1 \tag{3.3}$$

The function $floor(x)$ returns the largest integer not exceeding x.

The second permutation is defined by the equation

$$j = s \times floor(i/s) + (i + N_{CBPS} - floor(16 \times i/N_{CBPS})) \bmod s \qquad i = 0, 1, \ldots, N_{CBPS} - 1 \tag{3.4}$$

where s can be obtained from

$$s = max(N_{BPSC}/2, 1) \tag{3.5}$$

3.3.3.2 Software Implementation

First and second permutation pattern is constant for each modulation scheme. Our software implementation relies on this and uses pre-calculated lookup tables to reduce computational overhead. Furthermore, the memory usage of these tables have been taken into consideration. Observing that interleave depth is the same with the N_{CBPS} which depending on the modulatin scheme can be equals to 48, 96, 192, 288, appropriate data types for the lookup tables where used for minimize the memory usage. So for BPSK, QPSK and 16-QAM *unsigned char* data type was used and *unsigned short* for 64-QAM. So the total memory usage for the first permutation mem_{first_permut} is

$$\begin{aligned} mem_{first_permut} &= 48 \times \textbf{sizeof}(\text{unsigned char}) \\ &\quad + 96 \times \textbf{sizeof}(\text{unsigned char}) \\ &\quad + 192 \times \textbf{sizeof}(\text{unsigned char}) \\ &\quad + 288 \times \textbf{sizeof}(\text{unsigned short}) \\ &= 912 \ \textit{bytes} \end{aligned} \tag{3.6}$$

The second permutation lookup tables have exactly the same size, so the total amount

of memory used for the lookup tables is

$$\begin{aligned} mem_{total} &= 2 \times mem_{first_permut} \\ &= 1824 \text{ bytes} \end{aligned} \tag{3.7}$$

which is quite small and can fit easily inside modern CPUs caches.

3.3.4 Carriers Mapping

After scrambling, coding and interleaving the bit-stream should be appropriate mapped into the right sub-carriers of multiple consecutive OFDM symbols. IEEE 802.11 as mentioned before, uses 64 sub-carriers at each OFDM symbol. 48 of them are carrying actual data, 4 are used as pilots and the rest are set to zero. Each data sub-carriers shall be modulated by using BPSK, QPSK, 16-QAM or 64-QAM depending on the requested transmission rate of the current frame (Table 3.1), whereas pilot carriers are always BPSK modulated.

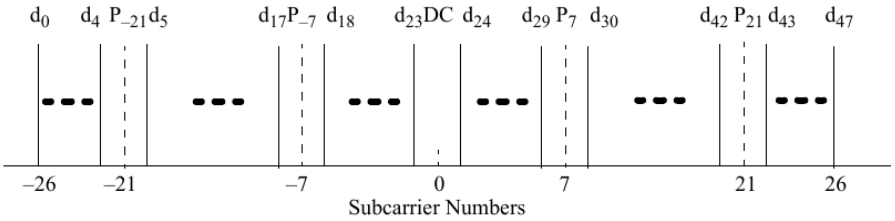


Figure 3.17: Subcarrier frequency allocation

Carriers at both sides and the carrier at the central frequency are set explicitly to zero. This is done due to some restrictions of hardware implementations. Low pass filters required from DAC and ADC for the conversion of the transmitted and received signals tend to have such a transfer function that attenuate the energy of the sub-carriers near the Nyquist frequency $f_s/2$ as Figure 3.18 presents. Furthermore, the center sub-carrier is not used because this frequency region suffers from heavy noise due to Direct Current (DC) offsets of the DACs/ADCs and carrier feed-through [35].

Table 3.7: Modulation-dependent normalization factor K_{MOD}

Modulation	K_{MOD}
BPSK	1
QPSK	$1/\sqrt{2}$
16-QAM	$1/\sqrt{10}$
64-AQM	$1/\sqrt{42}$

To map input data into OFDM symbols the input bit-stream is divided into groups, each containing N_{CBPS} bits. Each of these groups is converted into 48 complex numbers, the so called *constellation points*. Each complex number corresponds to N_{CBPC} bits and the transformation from bits to complex number and vice versa is based on the constellation vector of each modulation scheme. The bits to constellation points mapping for the available modulation schemes is presented in Figure 3.19. Depending the modulation

scheme, each constellation point $(I + jQ)$ is multiplied by a normalization factor K_{MOD} that can be derived from Table 3.7 using Equation 3.8.

$$d = (I + jQ) \times K_{MOD} \tag{3.8}$$

The purpose of this normalization is to achieve the same mean power for all the available constellations. The resulting complex constellation point d is placed in the appropriate frequency bin relative to the center frequency, using the mapping function $M(k)$, where k is the $k-th$ complex number of the 48 of the current OFDM symbol as Figure 3.17 shows.

$$M(k) = \begin{cases} k - 26 & 0 \leq k \leq 4 \\ k - 25 & 5 \leq k \leq 17 \\ k - 24 & 18 \leq k \leq 23 \\ k - 23 & 24 \leq k \leq 29 \\ k - 22 & 30 \leq k \leq 42 \\ k - 21 & 43 \leq k \leq 47 \end{cases} \tag{3.9}$$

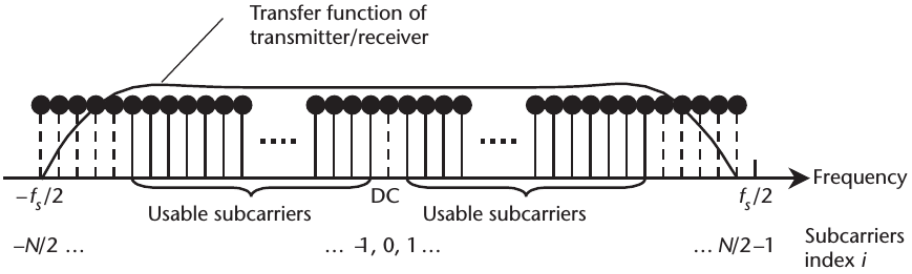


Figure 3.18: Transfer function of transmit/receive hardware and the impact to the OFDM system design

3.3.4.1 Pilot Carriers Insertion

Pilot carriers are extremely useful in OFDM systems, as they can used for frequency synchronization, channel estimation and phase tracking at the receiver. For these reasons, the values of pilots carriers should be predefined for each OFDM symbol but not the same across all symbols of a frame in order to avoid spectral lines. Also pilot carriers' values should be known for both transmitter and receiver.

The modulation used for each pilot carriers is BPSK. The value of that a pilot carrier has can be derived from Equation 3.10 multiplied by the polarity value.

$$P_i = \begin{cases} 1 & i = -21 \\ 1 & i = -7 \\ 1 & i = 7 \\ -1 & i = 21 \end{cases} \tag{3.10}$$

```
static const char polarity[127] =
{ 1, 1, 1, 1, -1, -1, -1, 1, - 1, -1,
  -1, -1, 1, 1, -1, 1, -1, -1, 1, 1,
  -1, 1, 1, -1, 1, 1, 1, 1, 1, -1,
  1, 1, 1, -1, 1, 1, -1, -1, 1, 1, 1,
  -1, 1, -1, -1, -1, 1, -1, 1, -1, -1,
  1, -1, -1, 1, 1, 1, 1, 1, - 1, -1, 1,
  1, -1, -1, 1, -1, 1, -1, 1, 1, -1,
  -1, -1, 1, 1, -1, -1, -1, -1, 1, -1,
  -1, 1, -1, 1, 1, 1, 1, -1, 1, -1, 1,
  -1, 1, -1, -1, -1, -1, -1, 1, -1, 1,
  1, -1, 1, -1, 1, 1, 1, -1, -1, 1,
  -1, -1, -1, 1, 1, 1, -1, -1, -1, -1,
  -1, -1, -1
}
```

Listing 3.2: Pilot carriers polarity look-up table

Table 3.8: Example of pilots insertion

i	OFDM Symbol	p_i	Pilot -21	Pilot -7	Pilot 7	Pilot 21
0	SIGNAL	1	$1.0 + 0.0j$	$1.0 + 0.0j$	$1.0 + 0.0j$	$-1.0 + 0.0j$
1	DATA	1	$1.0 + 0.0j$	$1.0 + 0.0j$	$1.0 + 0.0j$	$-1.0 + 0.0j$
2	DATA	1	$1.0 + 0.0j$	$1.0 + 0.0j$	$1.0 + 0.0j$	$-1.0 + 0.0j$
3	DATA	1	$1.0 + 0.0j$	$1.0 + 0.0j$	$1.0 + 0.0j$	$-1.0 + 0.0j$
4	DATA	-1	$-1.0 + 0.0j$	$-1.0 + 0.0j$	$-1.0 + 0.0j$	$1.0 + 0.0j$

Polarity is a pseudo-random sequence with 127 elements, each having a value 1 either -1. Listing 3.2 provides a lookup table with the polarity values. The sequence can be produced by using the scrambler of Section 3.3.1 with an initial seed of all 1's. Substituting all 1's to -1's and all 0's with 1's of the scrambler output bit stream, the polarity sequence is derived. The first element of the sequence is used for the polarity of the pilots that belong to the SIGNAL OFDM symbol, whereas the rest of the elements are used for the pilots of the remaining DATA OFDM symbols. An example of the polarity use is presented in Table 3.8. If the DATA symbols are more than the sequence elements the sequence is traversed again from the beginning.

3.3.4.2 Software Implementation

Once again the software implementation is based on lookup tables. Constellations points are placed in lookup tables such a way, that if the numerical value of a group consisted from N_{CBPC} bits is used as index, then the corresponding complex constellation point can be retrieved.

IEEE 802.11a changes dynamically the modulation scheme so all the lookup tables for all possible modulations should be loaded. This is not a problem as the total number of constellation points is relative small. The total amount of memory required for the lookup tables is

$$\begin{aligned} memory_{total} = (&sizeof(constellation_{BPSK}) \\ &+ sizeof(constellation_{QPSK}) \\ &+ sizeof(constellation_{16-QAM}) \\ &+ sizeof(constellation_{64-QAM})) * sizeof(complex) = 688 \text{ bytes} \end{aligned} \tag{3.11}$$

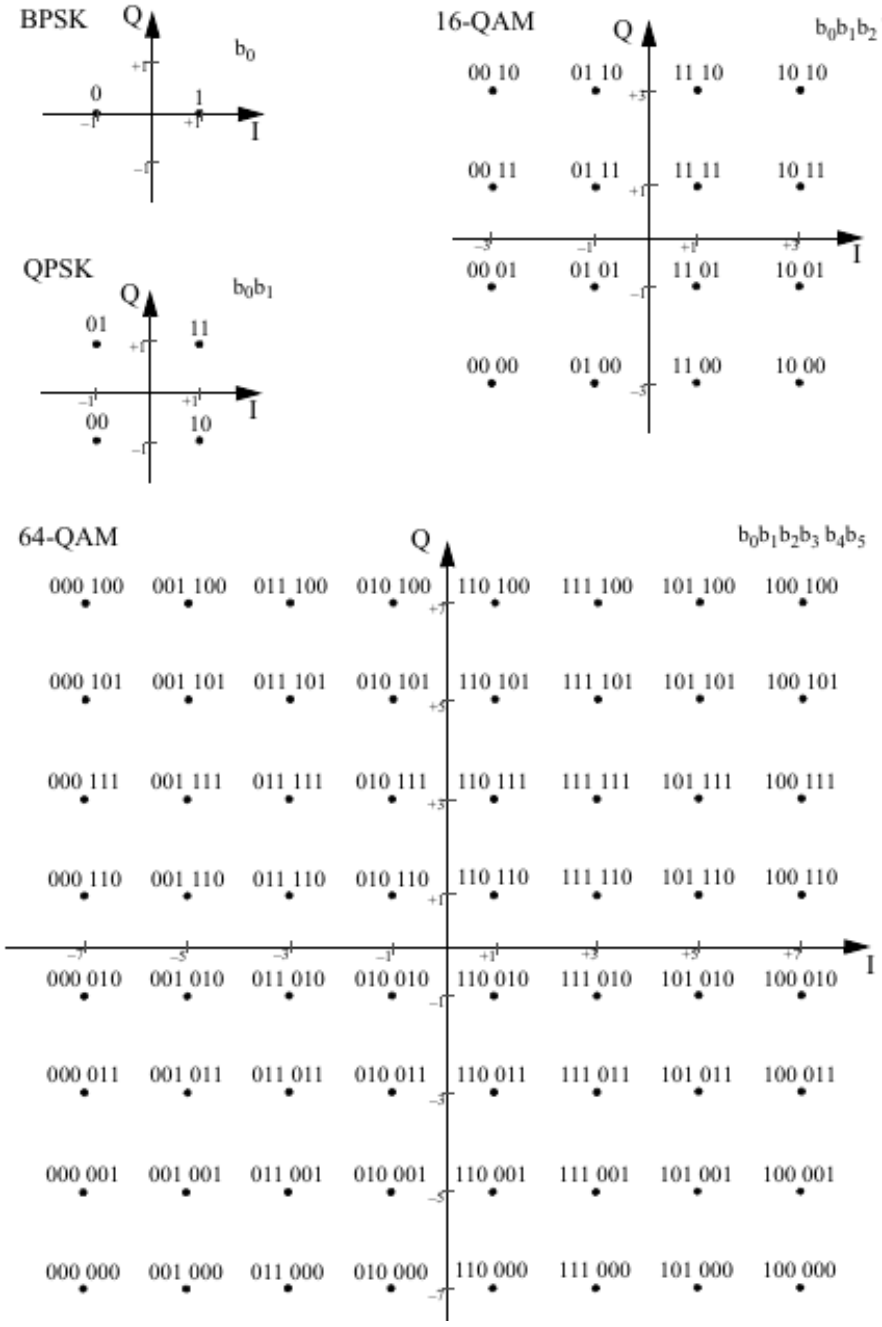


Figure 3.19: BPSK, QPSK, 16-QAM, and 64-QAM constellation bit encoding

3.3.5 Inverse Fast Fourier Transform

After mapping the constellation points into the right frequency bins, the time domain signal should be created. This is accomplished by computing the 64 point IFFT of the corresponding sub-carriers. Each sub-carrier should be assigned to the appropriate IFFT bin according to Figure 3.20. The transformation produces 64 complex samples in the time domain, ready to be transmitted over the air from the SDR device.

For the IFFT calculation GNU Radio uses the well known and highly optimized FFTW [36] library. This library takes advantage of parallel computing and SIMD instructions available in modern CPUs, achieving an $O(n \log n)$ complexity for every FFT size. This makes the FFTW library ideal for the demanding timing constraints of the protocol. As Table 3.9 presents, each OFDM symbol has a duration of 4 μs (8 μs, 16 μs for 10 MHz and 5 MHz channel bandwidth respectively) so the IFFT algorithm should perform the computational tasks in less time in order the system to be real-time.

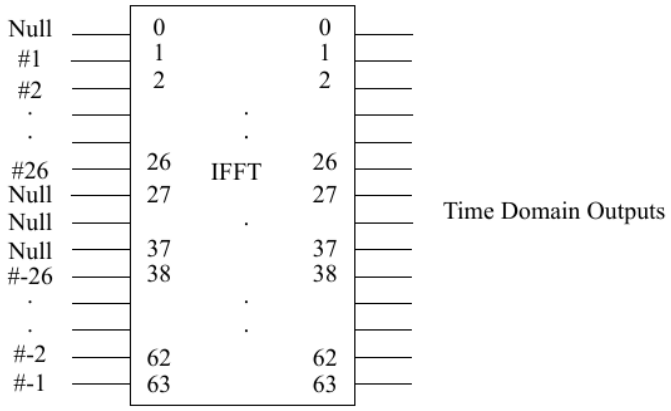


Figure 3.20: Inputs and Outputs for the IFFT

3.3.6 Cyclic Prefix Insertion

Prior transmission of the time domain samples produced by the IFFT, a Cyclic Prefix (CP) should be inserted in front of the IFFT output sequence as a Guard Interval (GI). This is accomplished by copying a portion of the last output IFFT samples. The number of the samples that are copied is specified by each communication protocol. Large GI shields the system from multipath phenomena and synchronization accuracy issues in lack of effective throughput.

IEEE 802.11a uses a GI which has duration of $T_{GI} = T_{FFT}/4$ which corresponds to 16 time domain samples (Table 3.9). As far as the software implementation is concerned, adding a CP is quite easy and it is performed by copying the last 16 samples of the IFFT block output in front of them. The resulting symbol has a duration of $(T_{GI} + T_{FFT})$ which corresponds to 80 samples (Table 3.9) and are ready to be transmitted over the air.

3.3.7 Transmitting samples using an SDR Device

The commodity of SDR hardware use ADCs and DACs that have up to 16 bits resolution. For this reason, interfaces that communicate with the hardware use two interleaved short

Table 3.9: Timing related parameters

Parameter	**Value (20 MHz channel)**	**Value (10 MHz channel)**	**Value (5 MHz channel)**
N_{SD}: Number of data subcarriers	48	48	48
N_{SP}: Number of pilot subcarriers	4	4	4
N_{SZ}: Number of zero subcarriers	12	12	12
N_{ST} : Number of subcarriers, total	64 (N_{SD}+N_{SP}+N_{SZ})	64 (N_{SD}+N_{SP}+N_{SZ})	64 (N_{SD}+N_{SP}+N_{SZ})
Δ_F: Sub-carrier Spacing	0.3125 MHz (20 MHz/64)	0.15625 MHz (10 MHz/64)	0.078125 MHz (5 MHz/64)
T_{FFT}: IFFT/FFT period	3.2 μs	6.4 μs	12.8 μs
T_{GI}: Guard Interval (GI) duration	0.8 μs (T_{FFT}/4)	1.6 μs (T_{FFT}/4)	3.2 μs (T_{FFT}/4)
T_{SYM}: Symbol duration	4 μs ($T_{GI} + T_{FFT}$)	8 μs ($T_{GI} + T_{FFT}$)	16 μs ($T_{GI} + T_{FFT}$)

integers for the representation of the real and imaginary part of each complex sample. This also reduces the amount of redundant data that should be transferred between the host computer and the SDR device. Samples derived from the cyclic prefix block should be type-cast into an appropriate data type. Even if this seems to be a quite straight forward procedure, type casting may have a notable amount of performance degradation.

Each sample from derived from the cyclic prefix block should be processed one by one. For each complex number both real and imaginary part should be converted with a float to signed short integer type cast. At the data rate of 20 Mega Samples per Second (MSPS) which is required by the standard, even the cast itself may have a performance impact. To overcome these problems our approach exploits again the available SIMD instructions to speedup the type casting in both terms of latency and throughput. For more information refer to Section 5.1.

Unfortunately, type casting is not the only problem that arises when dealing with hardware. SDR devices perform buffering, just like network interfaces. The host computer feeds with data the device which consumes them in a constant rate, known also as Sampling Rate (SR). When the buffers are filled up, the hardware blocks the host data transfers. With this technique the data rate between the host and the device eventually has a mean value that is proportional to SR, preventing from overloading the host computer. In order to ensure proper and contiguous signal transmission, buffers should have sufficient data that correspond to the number of samples per second that are required regarding the SR configuration.

Noctar device, like most of the SDR devices do, store host data into a set of buffers. When a buffer is filled by the host, its data are becoming available and can be accessed from the DAC in order to be transmitted, while the host can continue to feed with data the rest of the available buffers of the device. When all the data of the buffer are processed by the DAC, the buffer becomes again available for the host computer. The procedure can be implemented efficiently with the use of a circular set of buffers as Figure 3.21 illustrates.

There is a severe problem with this approach though. Assuming a IEEE 802.11a frame of size S samples that fills two and a half buffers and by taking into consideration the bursty nature of the WLAN transmissions, continuous transmission of the frame is not feasible. The last part of the frame that is stored in the half filled buffer will never be transmitted until the buffer is full. This of course may happen later enough in time (e.g. after a second), so the receiver does not get the whole frame leading to a packet with erroneous data. For this reason, the design of buffering was altered. In our driver implementation instead of using a circular set of buffers, a large continuous in physical address space memory region was allocated. To use this memory region as a circular buffer, just two pointers heading to the start and the end of the memory addresses that enclose data which the host wrote to memory are enough. If the device reads data and transmit them over the air, the start pointer is increased. If the host computer writes data to memory, the end pointer is advanced. With this approach there is no need to wait for a buffer to be filled, so frames with arbitrary sizes can be send without problems. However, FPGAs Direct Memory Access (DMA) implementations may have some restrictions. Most of them have to do with the minimum or the maximum size of the DMA transaction or the memory alignment requirements. To overcome these problems the frame memory region can be spitted into segments and issue multiple DMA transactions. In order the next frame to start from an aligned memory address, appropriate zero padding is applied. Note that zero data does not affect the system at all, as they translated as a signal with zero amplitude.

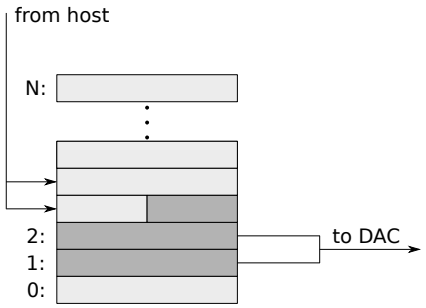


Figure 3.21: Samples buffering at SDR hardware

However, is this trick enough for a successful frame transmission? The answer is again no. While now the SDR device buffering handles an arbitrary size of frame properly, there two more levels of buffering. The data transfer between the host and the *Noctar* card is accomplished with the use of a character device. In order to feed the card with data, conventional methods are used for writing into files. The first level of buffering that is totally abstracted from the developer is done when the *fwrite()* function is used. This function may perform some internal buffering, so some of the last data of the frame may be issued to the card later on time. This problem can be easily bypassed by using the *write()* function, which is a system call that does not performs any buffering and calls the appropriate method of the character device driver directly. The second and more tricky

buffering level is performed by the GNU Radio platform. The scheduler may decide not to call the *work()* method of any block that belongs to the transmit chain. This may happen because the scheduler may wait for more input or output items to become available. If this occur while the frame is about to end, then the continuous transmission may not be feasible. To overcome this problem our implementation utilizes the *set_ max_ noutput_ items (int m)* method that GNU Radio platform provides. This method limits the number of output items that may produced at each *work()* or *general_ work()* invocation to m. Using appropriate value of m at each block of the transmit chain in order each block to produce exactly one OFDM symbol at every *work()* or *general_ work()* run, has a very convenient side effect. The corresponding block will be activated when available input items are enough for exactly one OFDM symbol. This will prevent each block from waiting for more available input items, so when the last OFDM symbol of a frame is processed, it will propagated towards the last processing block, reaching eventually the SDR device. A drawback of this solution is the increased CPU utilization, caused by the increased number of scheduler invocations, one for each OFDM symbol for every processing block that belongs to the transmit chain.

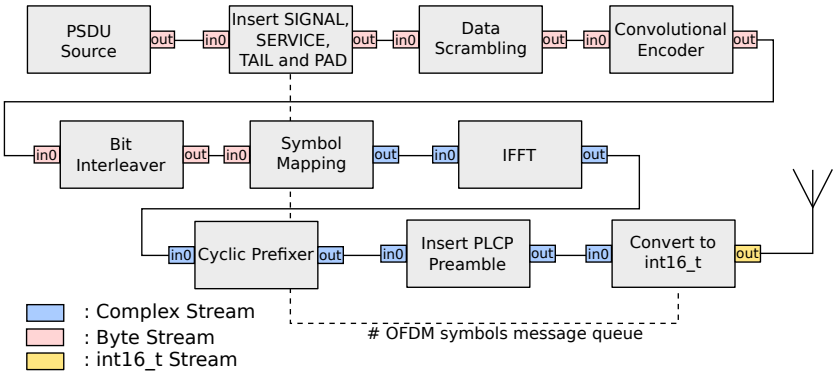


Figure 3.22: Improved TX chain

On the other hand, writing small chunks of data into the character device may also lead to discontinuous transmission. If samples of exactly one OFDM symbol are issued for transmission to the SDR device, the host may not be able to transfer the samples for the next OFDM symbol at the right time. This is very common in non-realtime operating systems with arbitrary preemption timings of each thread. For this reason the last block that is responsible to transfer the samples to the SDR device, performs buffering of OFDM symbols. However, in order to achieve continuous transmission of the whole frame, the number of the OFDM symbols that apart the transmitted frame should be known. This is accomplished with the help of an asynchronous message queue as Figure 3.22 presents. When the SIGNAL, SERVICE, TAIL and PAD field are inserted at the corresponding processing block, the number of the OFDM symbols that apart the frame becomes available. A message with this number is inserted at the message queue. The last block of the chain blocks until it receives the number of the OFDM symbols. When it does, it starts processing and buffering the received OFDM symbols based on the Algorithm 1.

```
buffer[MAX_FRAME_SIZE] = {0} ;
symbols_count = 0;
buffered_symbols = 0;
frame_symblols = 0;
/* A user defined threshold.  This threshold is set based on the
   desired latency, hardware buffers e.t.c                          */
threshold = x;
while program is running do
    if symbols_count == 0 then
        /* The end of the frame reached.  Write any buffered OFDM
           symbols at the device                                    */
        write (device, buffer, buffered_symbols × sizeof(OFDM symbol)) ;
        buffered_symbols = 0 ;
        /* Block waiting the number of OFDM symbols of the new frame
           frame                                                    */
        frame_symblols = msg_queue.delete_head ();
    else
        if buffered_symbols == threshold then
            write (device, buffer, buffered_symbols × sizeof(OFDM symbol)) ;
            buffered_symbols = 0 ;
        else
            memcpy (&buffer[buffered_symbols × sizeof(OFDM symbol)], input,
                sizeof (OFDM symbol)) ;
            buffered_symbols ++ ;
        end
    end
end
```

Algorithm 1: Buffering of OFDM symbols

3.4 Receive Chain

At the previous section the software implementation of IEEE 802.11a transmission presented. That was a quite straightforward procedure, with the blocks that apart the flowgraph to work in a quite sequential way. Output data from a block where propagated to the next block for processing, until data reach the SDR device.

On the other hand receiving 802.11a frames is not such simple. Several issues should be overcome having always as limitation the computational cost and delay or the SDR platform itself. For example, how the system can identify that is receiving a valid 802.11 frame? When the receiver should start decoding the samples that are currently received?

Furthermore, wireless channels are a quite dynamic and diverse medium introducing many impairments and challenges. For successful data reception, the system should be able to deal with these issues and be real-time at the same time.

At the following sections a real-time software implementation of an 802.11a receiver is presented.

3.4.1 Energy Detection

The SDR device, has been successfully set up and tuned at the appropriate frequency and started to produce samples in a constant data rate. It is obvious that processing all these samples should be a great overhead. Instead the system could process only those samples

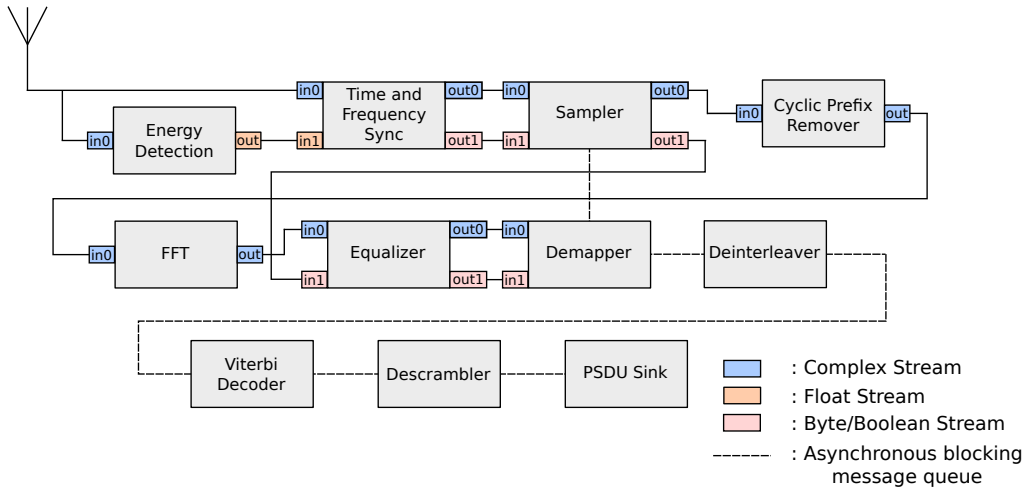


Figure 3.23: Overview of the IEEE 802.11a receive software implementation

that are part of a possible IEEE 802.11a frame. The question now is, how the system can identify a possible frame.

Every transmitted IEEE 802.11a frame has at the beginning some known training symbols (Figure 3.7). The short training sequence is constructed of 10 repetitions of a known sequence (denoted as $t_1 - t_{10}$) with period of $T_{FFT}/4\ \mu s = 16$ samples resulting in a total duration of 160 samples ($2 \times T_{SYM}$). These samples can be derived by computing the IFFT of the contents of Table 3.10. The resulting samples are presented in Table 3.11. Energy detection algorithms take advantage of the short training sequence trying to identify if a possible IEEE 802.11a frame is currently received.

Table 3.10: Frequency domain representation of the short sequences

#	**Re**	**Im**	#	**Re**	**Im**	#	**Re**	**Im**	#	**Re**	**Im**
-32	0	0	-16	1.472	1.472	0	0	0	16	1.472	1.472
-31	0	0	-15	0	0	1	0	0	17	0	0
-30	0	0	-14	0	0	2	0	0	18	0	0
-29	0	0	-13	0	0	3	0	0	19	0	0
-28	0	0	-12	-1.472	-1.472	4	-1.472	-1.472	20	1.472	1.472
-27	0	0	-11	0	0	5	0	0	21	0	0
-26	0	0	-10	0	0	6	0	0	22	0	0
-25	0	0	-9	0	0	7	0	0	23	0	0
-24	1.472	1.472	-8	-1.472	-1.472	8	-1.472	-1.472	24	1.472	1.472
-23	0	0	-7	0	0	9	0	0	25	0	0
-22	0	0	-6	0	0	10	0	0	26	0	0
-21	0	0	-5	0	0	11	0	0	27	0	0
-20	-1.472	-1.472	-4	1.472	1.472	12	1.472	1.472	28	0	0
-19	0	0	-3	0	0	13	0	0	29	0	0
-18	0	0	-2	0	0	14	0	0	30	0	0
-17	0	0	-1	0	0	15	0	0	31	0	0

In a first attempt, the well known Schmidl & Cox algorithm [37] was used. This algorithm computes the auto-correlation of the received signal. To accomplish that the received signal is multiplied by the complex conjugate of a delayed version of the original signal. The resulting samples are passed to a moving sum filter with an appropriate length.

Table 3.11: One period of IFFT of the short sequences

#	**Re**	**Im**	#	**Re**	**Im**	#	**Re**	**Im**	#	**Re**	**Im**
0	0.046	0.046	1	-0.132	0.002	2	-0.013	-0.079	3	0.143	-0.013
4	0.092	0.000	5	0.143	-0.013	6	-0.013	-0.079	7	-0.132	0.002
8	0.046	0.046	9	0.002	-0.132	10	-0.079	-0.013	11	-0.013	0.143
12	0.000	0.092	13	-0.013	0.143	14	-0.079	-0.013	15	0.002	-0.132
16	0.046	0.046	17	-0.132	0.002	18	-0.013	-0.079	19	0.143	-0.013
20	0.092	0.000	21	0.143	-0.013	22	-0.013	-0.079	23	-0.132	0.002
24	0.046	0.046	25	0.002	-0.132	26	-0.079	-0.013	27	-0.013	0.143
28	0.000	0.092	29	-0.013	0.143	30	-0.079	-0.013	31	0.002	-0.132
32	0.046	0.046	33	-0.132	0.002	34	-0.013	-0.079	35	0.143	-0.013
36	0.092	0.000	37	0.143	-0.013	38	-0.013	-0.079	39	-0.132	0.002
40	0.046	0.046	41	0.002	-0.132	42	-0.079	-0.013	43	-0.013	0.143
44	0.000	0.092	45	-0.013	0.143	46	-0.079	-0.013	47	0.002	-0.132
48	0.046	0.046	49	-0.132	0.002	50	-0.013	-0.079	51	0.143	-0.013
52	0.092	0.000	53	0.143	-0.013	54	-0.013	-0.079	55	-0.132	0.002
56	0.046	0.046	57	0.002	-0.132	58	-0.079	-0.013	59	-0.013	0.143
60	0.000	0.092	61	-0.013	0.143	62	-0.079	-0.013	63	0.002	-0.132

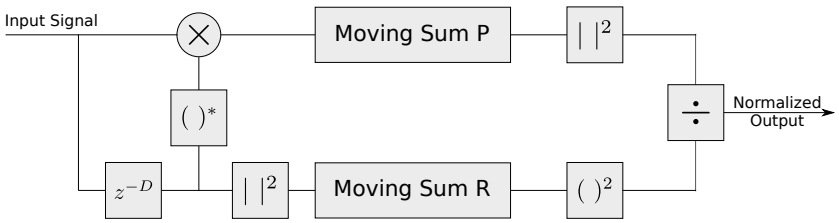


Figure 3.24: Schmidl & Cox energy detection algorithm

The energy of the auto-correlation varies for different signal amplitudes, as Figure 3.25a shows. This of course is not acceptable because the detector can not rely on a predefined threshold, above which can indicate a possible IEEE 802.11a frame start. For this reason normalization is performed by dividing the energy of the auto-correlation with the energy of the corresponding windowed input signal samples (Figure 3.25b). The above algorithm is mathematically described by Equation 3.12.

$$M(k) = \frac{|P(k)|^2}{R^2(k)} \tag{3.12}$$

where $P(k)$ and $R(k)$ are:

$$P(k) = \sum_{i=0}^{D-1} y(k+i)y^*(k+i+D), \tag{3.13}$$

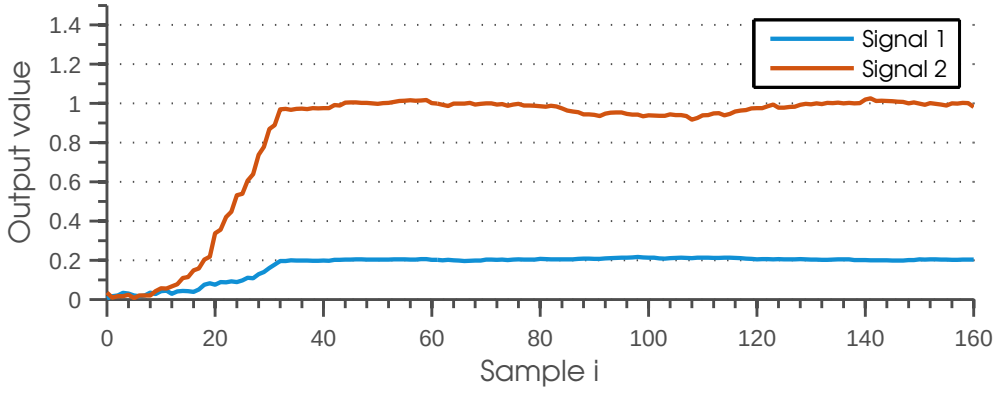
$$R(k) = \sum_{i=0}^{D-1} y(k+i+D)y^*(k+i+D) = \sum_{i=0}^{D-1} |y(k+i+D)|^2 \tag{3.14}$$

with $y(k)$ being the input signal and $D = 16$

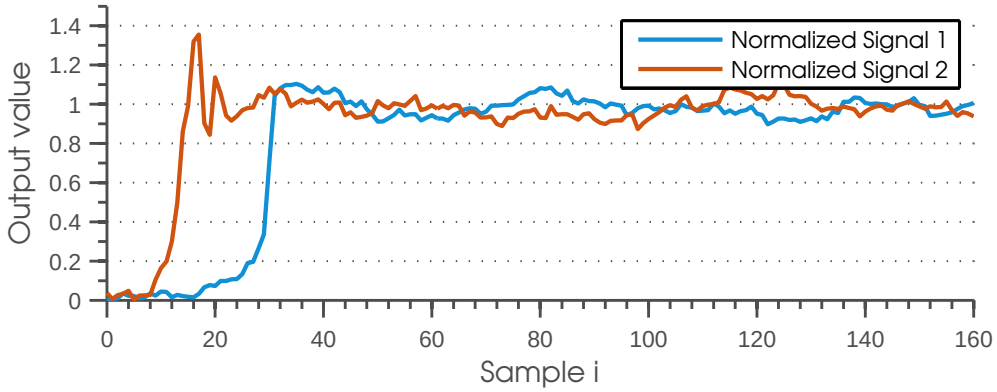
The software implementation of Equation 3.12 is presented at the flowgraph of Figure 3.24. The two moving sum processing blocks of the figure implement the corresponding Equations 3.13 and 3.14 respectively.

Despite the simple nature of this algorithm it has a main disadvantages. The first is that performs several different operations on the complex input stream. The other drawback is that the output produced by this method is a continuous plateau, during the short training sequence reception. In the case of energy detection, this is not a severe problem. However, timing synchronization which will be discussed in the next section, requires a very accurate estimation of the start of the frame. The plateau that is produced, makes this estimation very difficult and computationally demanding often with low accuracy.

Focusing on the disadvantages of the above algorithm, our approach tries to cope with them and provides a more accurate and efficient algorithm. Instead of using a delayed copy of the actual input signal, our approach uses the complex conjugate of a pre-calculated part of the short training sequence in reverse order, as the coefficients of an FIR filter. In fact the FIR filter has as much taps as the the number of samples that correspond to a period of the short training sequence (16 samples). This filter actually computes the cross-correlation of a known sequence and the received signal. Filtering the input signal with this FIR filter and computing the energy of the output samples, 10 distinct spikes are produced, as many as the periods that apart the short training sequence (Figure 3.26a). As with the previous approach, different input signal energies, produce different output levels, so again normalization technique should be applied. Rewriting Equation 3.12 the new proposed method is described by Equation 3.15.



(a)



(b)

Figure 3.25: Schmidl and Cox energy detection approach (a) Without normalization (b) Normalized

$$M(k) = \frac{|P(k)|^2}{R^2(k)} \tag{3.15}$$

where $P(k)$ and $R(k)$ are:

$$P(k) = \sum_{i=0}^{D-1} y(k+i)S(i), \tag{3.16}$$

$$R(k) = \sum_{i=0}^{D-1} |y(k+i)|^2 \tag{3.17}$$

$$\tag{3.18}$$

with $y(k)$ being the input signal, $D = 16$ and $S(i)$ the time domain of the short training

sequence derived from Table 3.11.

After normalization the output of the proposed algorithm, even for received frames with different signal energies has exactly the same form, as Figure 3.26b illustrates. This feature, simplifies the logic of energy detection and can enable joint energy detection - timing synchronization implementations.

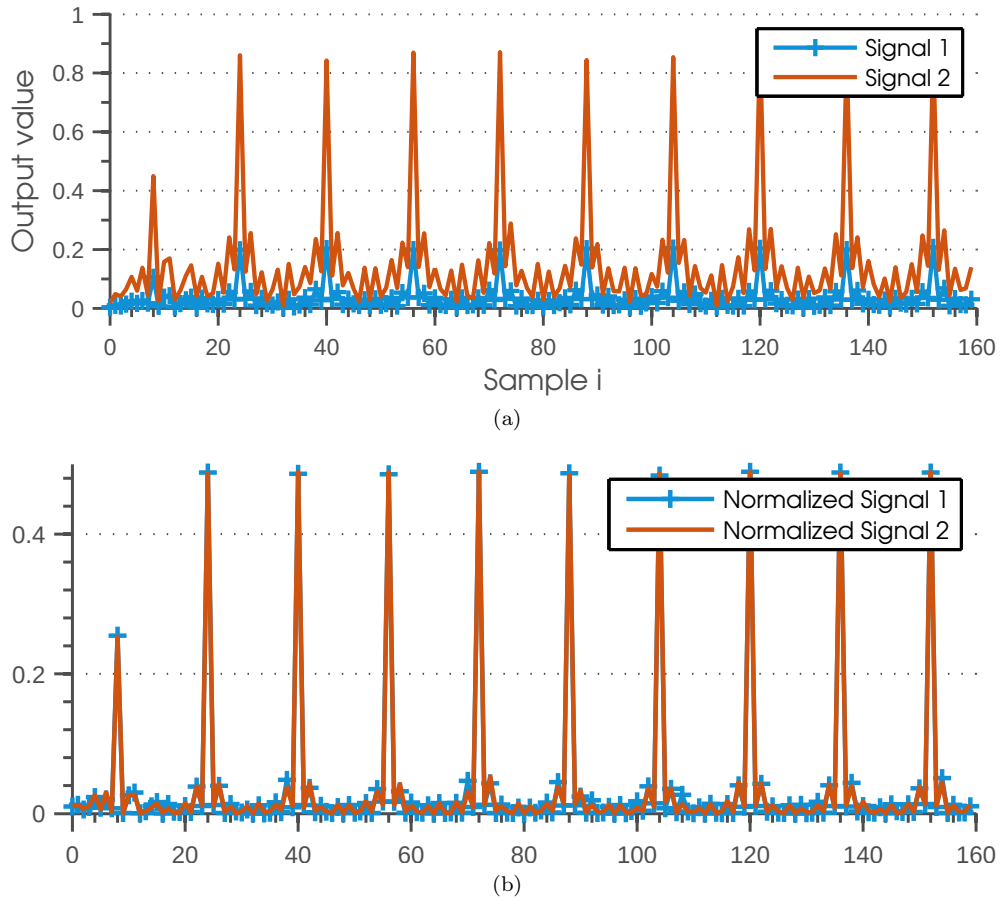


Figure 3.26: Proposed method for energy detection (a) Without normalization (b) Normalized

Another key feature of our approach is the tolerance that provides in noisy environments. Even for small SNR values, the proposed algorithm outperforms Schmidl & Cox approach. In Figure 3.27, a comparison of the the two algorithms is presented with input signals of different SNR values. The resistance of our algorithm in the various noise sources is based on the fact that our implementation does performs cross-correlation instead of auto-correlation, using the input signal and a part of the IEEE 802.11a standard training sequence. On the other hand, Schmidl & Cox approach uses a delayed stream of the input signal to compute the auto-correlation assuming that any possible interference is uncorrelated, which is not always true.

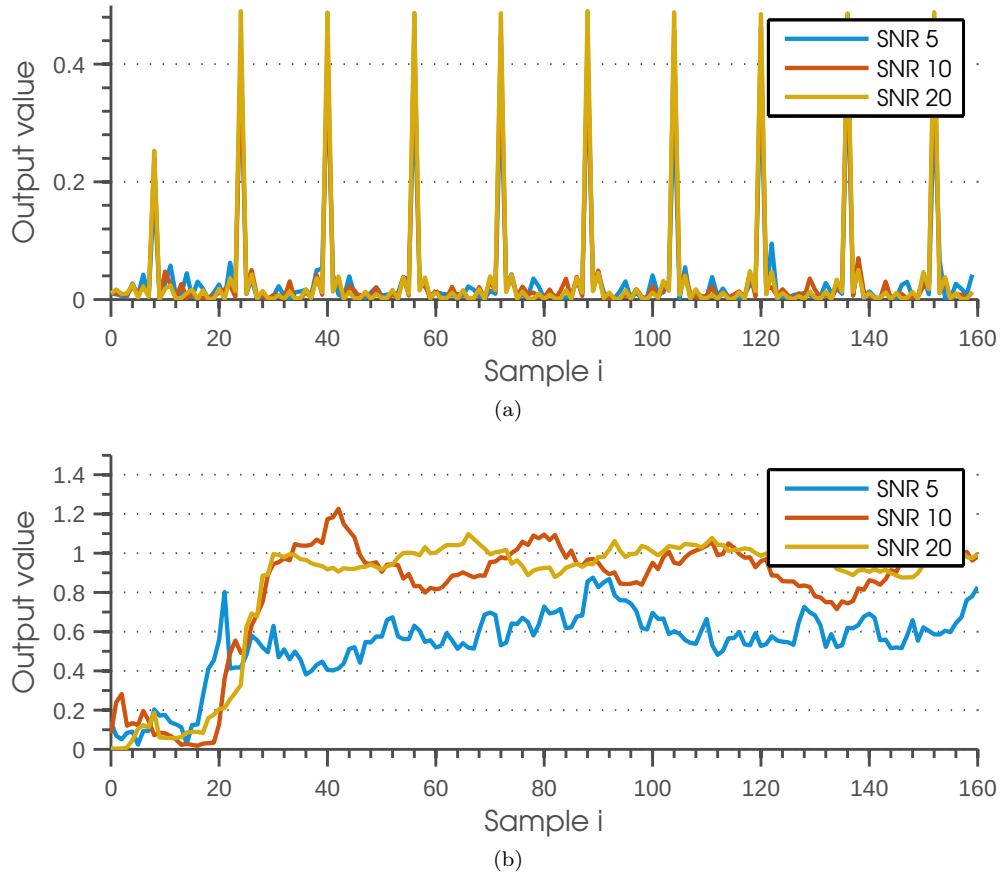


Figure 3.27: Comparison of (a) Proposed (b) Schmidl & Cox energy detection approach for different SNR values

3.4.2 Time and Frequency Synchronization

Time synchronization refers to the procedure of finding the start of an OFDM symbol. This is essential for proper frame decoding. In burst transmission protocols like IEEE 802.11a, the correct starting point should be found in a very short period of time. For better accuracy the timing synchronization is performed in two steps. Coarse timing synchronization makes use of the short training sequence, estimating a rough starting point. Then, fine timing synchronization process follows, utilizing the long training sequences of the preamble for better accuracy. At this point the start of the SIGNAL symbol, which is the first OFDM symbol of the frame that contains unknown to the receiver data, should be already found, otherwise the frame can not be decoded.

3.4.2.1 Coarse Timing Synchronization

Coarse timing synchronization relies on the short training sequence of the preamble, trying to find a rough estimation of the start of an OFDM symbol. In the case of our implementation, the desired OFDM symbol is the start of the long training sequence (denoted as $T1$ in Figure 3.7). This specific symbol was selected for two main reasons:

1. The GI of the $T1$ training symbol (denoted as GI2) has double the duration of the GI of the other OFDM symbols. This allows a greater estimation error margin, making it ideal for the coarse timing synchronization process.

2. Knowing approximately the start of the long training sequence, provides the flexibility to start the fine time and frequency synchronization process after the estimated point, saving vital computational resources.

Most timing synchronization techniques, use the output of a Schmidl & Cox variant approach (Equation 3.12) in order to find the start of the desired symbol. To accomplish that the maximum output value should be found within a window. The values of the window should be above a certain threshold. The window size S is directly related with the size of the short training sequence and its period. It is expected that during the reception of a short training sequence, $M(d)$ will be a plateau with amplitude above a user defined threshold and close to 1. The length of this plateau is exactly the same with the length of the 10 repetitions of the short training sequences ($10 \times 16 = 160$ samples). Observing Scmhidl & Cox approach output for different SNR values in Figure 3.27b it is obvious that peaks are in different indexes for every scenario. This makes the timing synchronization harder. To avoid this problem most approaches using an appropriate window S, that is one or two short training sequence periods long ($S = 2 \times T_{FFT}/4$ or $S = T_{FFT}/4$ samples), so the first time that the window is full with values above the threshold, the maximum value is computed. The index of the peak, is the desired estimation (Equation 3.19).

$$\hat{n} = arg \max_{0<k<S}(M(d) > threshold) \tag{3.19}$$

The main drawback of this method is the large deviation of the estimation index $\hat{n}$ from the ideal point n_{ideal}, for different scenarios. This greatly affects both frequency and fine time synchronization accuracy (Sections 3.4.2.3 and 3.4.3).

To overcome this problem the output of the proposed algorithm described by Equation 3.15 and a different timing synchronization algorithm were used. Based on the fact that the peaks have a certain distance between them, equal to the period of the short training sequence and that this sequence will eventually produce 10 peaks, a much more simple time synchronization algorithm can be applied. The exact procedure is described

in Algorithm 2.

```
peaks_count = 0;
while there are samples do
    x = normalized cross correlation;
    if x > threshold then
        peaks_count++;
        if x > max then
            max = x;
            max_index = index of x;
        end
        if peaks_count == 10 then
            return [index of x, max_index];
        end
        skip 16 samples;
    end
    /* False previous peak                                    */
    else if peaks_count then
        peaks_count = 0;
        max = 0;
    end
end
```

Algorithm 2: Coarse timing synchronization algorithm

The advantages of the proposed algorithm are the less computing resources that are needed and higher accuracy for both timing and Carrier Frequency Offset (CFO) estimation.

Unfortunately even with the proposed algorithm, the system can not perform calculations in real-time. The bottleneck is the FIR filter of Equation 3.15 that consumes too many resources and cannot produce output samples in a rate greater than 20 MSPS required by the standard. To solve this problem several optimization techniques where used, utilizing SIMD instructions and are described in Section 5.1. Even with these optimizations the output rate of the FIR filter was below the required limit. The extremely high demands on computing resources relies on the complex multiplications that are performed. More precisely, for one output sample of the FIR filter of Equation 3.15, 16 complex multiplications and 16 complex additions are needed. GNU Radio uses two interleaved float numbers for the representation of a complex, so for every output of the filter the number of the operations required is:

$$
\begin{aligned}
&\text{total operations} = \\
&\quad 16 \times (4 \text{ float multiplications} + 1 \text{ float addition} + 1 \text{ float substraction}) \\
&\quad + 16 \times (2 \text{ float additions}) \\
&\quad = 64 \text{ float multiplications} + 48 \text{ float additions} + 16 \text{ float substractions}
\end{aligned}
$$

To drastically reduce the number of float operations we followed a different approach that eliminates the complex operations. As Table 3.10 shows, the real and the imaginary part of the frequency domain of the short training sequence are equal. That gives a lot of information about the correlation of the real and imaginary parts at the time domain. Indeed, the time domain of the short sequence at Table 3.11 shows that the real and the imaginary part of the time domain are the same with a phase difference that is equal with the half of the short training sequence period ($(T_{FFT}/4)/2 = 8$ samples).

Based on the above observation, our new approach performs the cross-correlation of the real part of the input signal. Furthermore, as coefficients of the FIR filter, the real part of the first 16 samples of the short training sequence are used. Equation 3.15 can now be written as:

$$M(k) = \frac{|P_{real}(k)|^2}{R^2_{real}(k)} \tag{3.20}$$

where $P_{real}(k)$ and $R_{real}(k)$ are:

$$P_{real}(k) = \sum_{i=0}^{D-1} \mathrm{Re}\Big(y(k+i)\Big)\,\mathrm{Re}\Big(S(i)\Big), \tag{3.21}$$

$$R_{real}(k) = \sum_{i=0}^{D-1} |\,\mathrm{Re}\Big(y(k+i)\Big)|^2 \tag{3.22}$$

with $y(k)$ being the input signal, $D = 16$ and $S(i)$ the time domain of the short training sequence derived from Table 3.11.

With this approach the computations that are needed are greatly reduced and can be performed in real-time. The total number of operations for each output sample of the FIR filter is now:

$$\begin{aligned}\text{total operations} &= 16 \times (1 \text{ float multiplication} + 1 \text{ float addition}) \\ &= 16 \text{ float multiplications} + 16 \text{ float additions}\end{aligned}$$

But this great speedup comes with a serious cost. The transmit and the receive device may have a slightly different center frequency. This is known as CFO and techniques to compensate its effects will be presented at Sections 3.4.2.2 and 3.4.3. The contribution of the CFO to the received signal $r(t)$ can be expressed as

$$r(t) = s(t) \times e^{-j2\pi\Delta f t} \tag{3.23}$$

where $s(t)$ is the actual signal and Δf the frequency difference between the transmitter and receiver. Coarse timing synchronization is performed before any CFO correction happens so our algorithm should be able to deal with this impairment and still have better accuracy and performance from other synchronization techniques.

The effect of the CFO at the proposed synchronization method is presented at Figure 3.28. It is clear that the approach that uses only the real part of the input signal and real FIR coefficients is affected by the CFO whereas the previous approach that uses complex input signal and coefficients seems that remains unaffected.

Hopefully all is not lost. A closer look at Figure 3.28 reveals that there are spikes with various amplitudes not only every $T_{FFT}/2 = 16$ samples, but also every $(T_{FFT}/4)/2 = 8$ samples. Moreover these extra spikes seems to be complementary in amplitude with the spikes that are appeared every 16 samples. Based on this phenomenon, the detection algorithm of Algorithm 2 can be improved to take into consideration the above effect

without increasing its complexity.

```
peaks_count = 0;
while there are samples do
    x = normalized cross correlation[i];
    /* Take the value of the extra peak after (T_FFT/4)/2 = 8 samples */
    y = normalized cross correlation[i + 8];
    if x + y > threshold then
        peaks_count++;
        if x > max then
            max = x;
            max_index = index of x;
        end
        if peaks_count == 10 then
            return [index of x, max_index];
        end
        skip 16 samples;
    end
    /* False previous peak                                          */
    else if peaks_count then
        peaks_count = 0;
        max = 0;
    end
end
```

Algorithm 3: Improved coarse timing synchronization algorithm

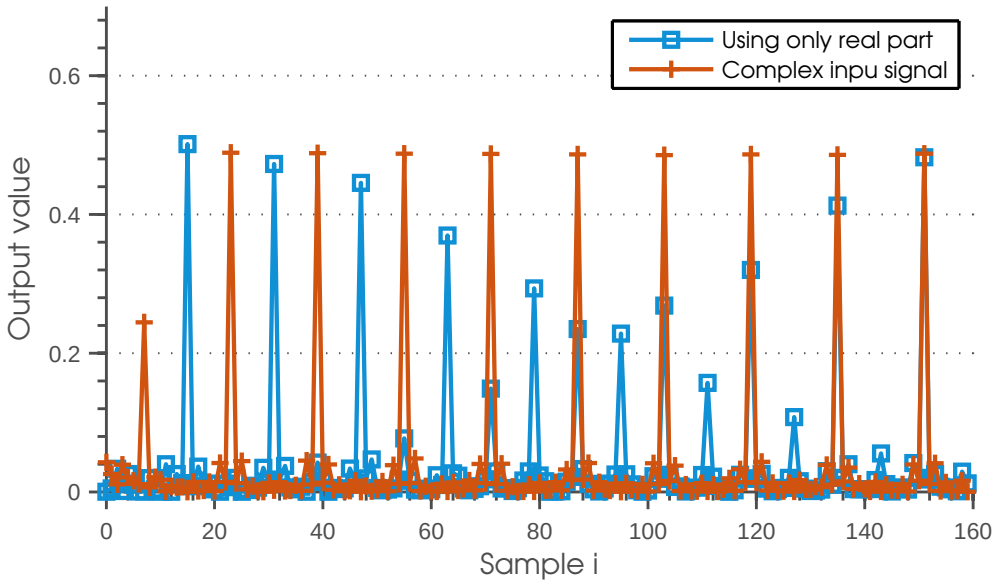


Figure 3.28: Impact of the CFO at the cross-correlation using complex and real input signal

Of course the above phenomenon it is no coincidence. The received signal with CFO is derived by multiplying the complex original signal with a complex cosine with frequency

Δf as described in Equation 3.23. From the definition of complex multiplication

$$(a, bj) \times (c, dj) = (ac - bd, (ad + cb)j) \tag{3.24}$$

it is clear that the imaginary parts of both original signal and complex cosine contribute to the real part of the received signal. Assuming a CFO of Δf Hz and a sample period of T_s, the received sampled signal $r(i)$ during the short training sequence $s(i)$ will be

$$r(i) = s(i) \times e^{-j2\pi \times \Delta f \times i \times T_s} \tag{3.25}$$

Now assume two time points t_{16} and t_{16+8} which correspond to the 16-th and $16 + T_{FFT}/8 = 24$-th sample of the short training sequence accordingly. The effect of the CFO at these samples would be

$$\begin{aligned} r(16) &= s(16) \times e^{-j2\pi \times \Delta f \times 16 \times T_s} \\ &= s(16) \times \big(\cos(2\pi \times \Delta f \times 16 \times T_s) - j \sin(2\pi \times \Delta f \times 16 \times T_s)\big) \end{aligned} \tag{3.26}$$

$$\begin{aligned} r(24) &= s(24) \times e^{-j2\pi \times \Delta f \times 24 \times T_s} \\ &= s(24) \times \big(\cos(2\pi \times \Delta f \times 24 \times T_s) - j \sin(2\pi \times \Delta f \times 24 \times T_s)\big) \end{aligned} \tag{3.27}$$

Our proposed algorithm utilized only the real part, so based on (3.26) and (3.27)

$$\begin{aligned} Re\{r(16)\} = {} & Re\{s(16)\} \times \cos(2\pi \times \Delta f \times 16 \times T_s) \\ & + Im\{r(16)\} \times \sin(2\pi \times \Delta f \times 16 \times T_s) \end{aligned} \tag{3.28}$$

$$\begin{aligned} Re\{r(24)\} = {} & Re\{s(24)\} \times \cos(2\pi \times \Delta f \times 24 \times T_s) \\ & + Im\{r(24)\} \times \sin(2\pi \times \Delta f \times 24 \times T_s) \end{aligned} \tag{3.29}$$

The above equations reveal that the imaginary part of both the CFO carrier and the original signal contribute to the real part of the received. Furthermore, based on the symmetry property of the short training sequence described in 3.31, Equation 3.29 can re-written as

$$\begin{aligned} Re\{r(24)\} = {} & Im\{s(16)\} \times \cos(2\pi \times \Delta f \times 24 \times T_s) \\ & + Re\{r(16)\} \times \sin(2\pi \times \Delta f \times 24 \times T_s) \end{aligned} \tag{3.30}$$

This indicates the strong relationship that is created from the symmetry property, so for this reason our cross-correlation based algorithm produce peaks also at every $t_{i\times 16+T_{FFT}/8}$ sample of the short training sequence.

$$\begin{cases} Re\{\text{short sequence}(i)\} = Im\{\text{short sequence}(i \pm T_{FFT}/8)\} \\ Im\{\text{short sequence}(i)\} = Re\{\text{short sequence}(i \pm T_{FFT}/8)\} \end{cases} \tag{3.31}$$

3.4.2.2 Coarse Frequency Synchronization

Transmitting devices using a local oscillator to up-convert a baseband signal to a target RF frequency f_{target}. Respectively receiving devices use their own oscillator with frequency f_{target} for the reception of the RF signal before they down-convert it to baseband.

Unfortunately in real systems f_{target} is not the same at both receive and transmit sides. Oscillators may have a small offset Δf from the desired f_{target} frequency. Moreover, this offset may not be stable in time. This phenomenon is well investigated by many studies and several advances have been made in order to equip telecommunication systems with more accurate and stable oscillators [38, 39].

The Δf frequency offset is known also as Carrier Frequency Offset (CFO). Frequency synchronization is a process of estimating Δf CFO and compensate it. The estimation is performed at the start of a frame and used for the entire frame, so frequency synchronization process assumes that the CFO remains constant over time. As mentioned before, oscillators may produce also fluctuations of the f_{target} frequency during time. How the systems deals with these kind of frequency variations is described at Section 3.4.7.

OFDM systems are quite sensitive in CFO errors. This happens for two main reasons. Firstly, each sub-carrier is not sampled at its peak, leading to amplitude attenuation and smaller SNR values. In addition, a Δf CFO, shifts all OFDM sub-carriers on a certain distance from the ideal sub-carrier frequency and can be computes as:

$$shift = \sum_{i=1}^{n} (T_s \times \Delta f) \tag{3.32}$$

where T_s denotes the period of the OFDM symbol and n the number of the OFDM symbols processed. This shift has as a result that the sub-carriers of the OFDM symbol are no longer orthogonal, meaning that part of the energy of the adjacent sub-carriers interfere each other. This impairment is known as Inter-Carrier Interference (ICI). The above statements and the impact of the CFO on an OFDM symbol are visually described at Figure 3.29.

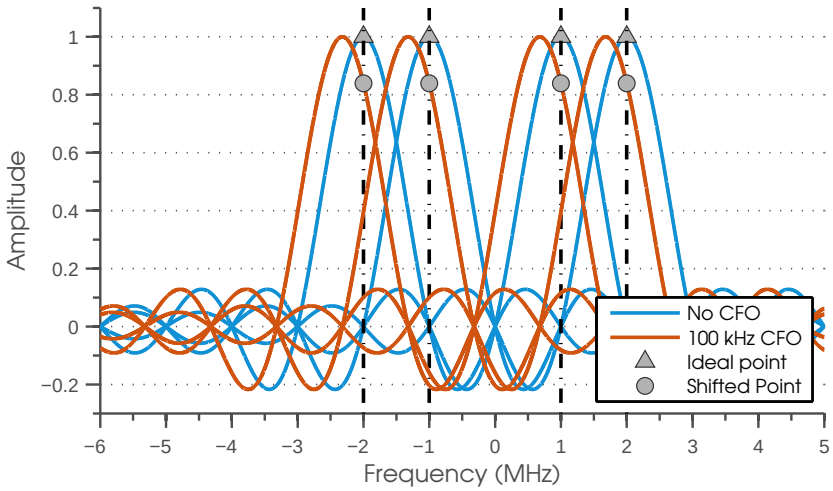


Figure 3.29: Loss of orthogonality due to CFO

Each IEEE 802.11a frame consists from several OFDM symbols. Based on Equation 3.32 it is easy to understand that the effects of the CFO are additive during the decoding of the consecutive OFDM symbols of the frame. This will cause the constellation to be rotated $\Delta\phi$ for each OFDM symbol of the frame. Eventually the constellation points will cross the decision boundaries of the modulation scheme, making the data non-

decodable. For example in Figure 3.30 the system is imposed at a CFO of 5 kHz, using 16-QAM as modulation scheme. It is obvious that after 4 OFDM symbols the constellation points pass the decision boundaries making the frame undecodable. Several studies [40, 41] have been performed to measure the performance degradation that CFO introduces at different modulation schemes.

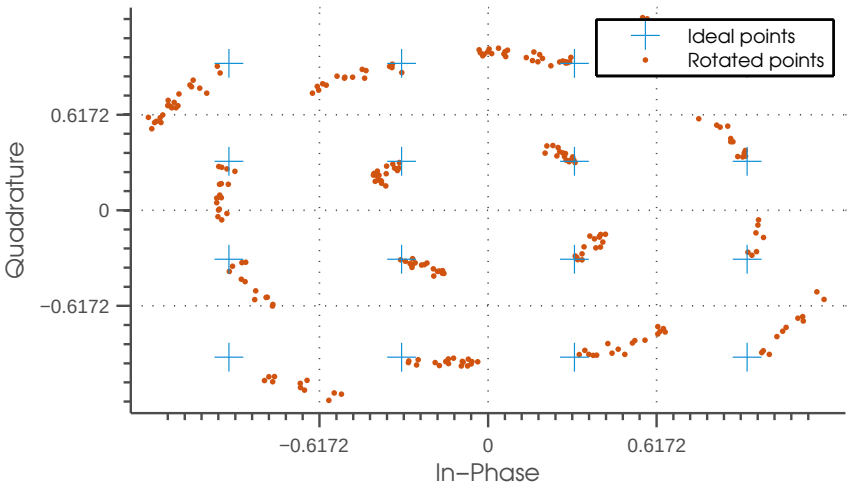


Figure 3.30: Constellation rotation due to CFO

From the above it is clear that CFO highly degrades the performance and the system should be able to deal with it. The CFO compensation in IEEE 802.11a is performed in two phases. The first phase performs a coarse estimation of the CFO and corrects it. The second and final phase (Section 3.4.3) performs a more fine grained estimation.

In order to correct the frequency offset Δf, first a good estimation $\hat{\Delta} f$ should be retrieved. To accomplish that, several techniques and algorithms have been proposed over the years and can be divided into two main categories:

- Time domain approaches that using training symbols
- Frequency domain approaches that analyze the received signal in frequency

Our implementation uses a time domain approach for estimating the CFO. The training information that is needed for this approach is provided by the IEEE 802.11a short training sequence that used for the energy detection and timing synchronization. Several time domain approached have been proposed. Most of them [42, 37, 41, 43] uses two consecutive and identical training symbols to estimate the $\hat{\Delta} f$. Our approach uses the estimation of two consecutive short training sequences t_i and t_{i+1}.

The main reason for this choice is the CPU resources that can be gained by taking advantage the output of the energy detector and coarse time synchronization. Algorithm 3 returns also the index of the maximum peak that observed. This maximum peak is the best estimation for the end of one from the 10 repetitions of the short training sequences $(t_1 - t_{10})$. Knowing the end of a short training sequence t_i means that also the start of the t_i and the next short training symbol t_{i+1} can be found (extreme cases which the estimation corresponds to t_1 or t_{10} are not described for simplicity). The CFO estimation

can be found by applying:

$$\hat{\Delta}f = \angle\left(\frac{\sum_{j=0}^{D-1} \hat{t}_i(j) \times \hat{t}^*_{i+1}(j))}{2 \times \pi \times T_s}\right. \tag{3.33}$$

where D is the period of the short training sequence and T_s the sampling period.

After obtaining the $\hat{\Delta}f$ the effects of the CFO can be compensated and an estimation of the original signal $s(i)$ can be retrieved based on Equation 3.34. Ideally if $\hat{\Delta}f = f$, then the original transmitted signal $s(i)$ is obtained.

$$\begin{aligned}\hat{s}(i) &= r(i) \times e^{j2\pi\times\hat{\Delta}f\times i\times T_s} \\ &\overset{(3.23)}{=} si(i) \times \times e^{-j2\pi\times f\times i\times T_s} \times e^{j2\pi\times\hat{\Delta}f\times i\times T_s} \\ &= s(i) \times e^{-j2\pi\times(f-\hat{\Delta}f)\times i\times T_s}\end{aligned} \tag{3.34}$$

In order to correct the CFO our system utilizes a software Numerically Controlled Oscillator (NCO). This processing block produces a complex cosine wave with frequency $\hat{\Delta}f$ at the same sample rate $\frac{1}{T_s}$ of the receiver. This wave is multiplied by the received input signal in the time domain resulting to the estimation of the originally transmitted signal. For each new frame, the frequency of the NCO is updated with the new value of $\hat{\Delta}f$. The overview of the frequency synchronization process is depicted in Figure 3.31.

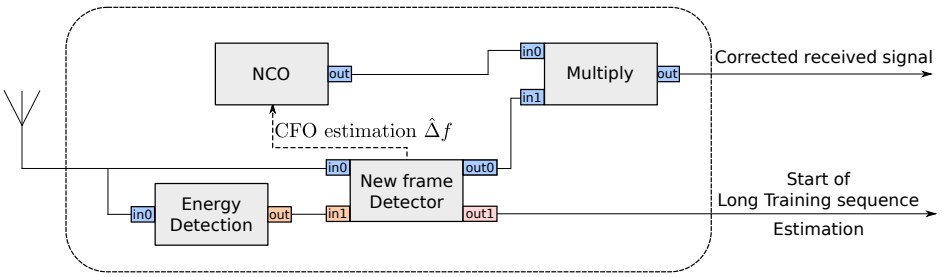


Figure 3.31: Coarse time and frequency synchronization hierarchical block

3.4.2.3 Fine Timing Synchronization

Fine timing synchronization is performed after an indication of the coarse timing synchronization and after the coarse CFO correction process. This procedure has as goal to find the start of the SIGNAL OFDM symbol, using the long training sequence of the preamble. Likewise the energy detection, the same algorithm was used with the only difference that the coefficients of the FIR filter were obtained by the time domain of the long training sequence. As the number of taps is concerned, normally a 64 taps filter should be used as the long training sequence is constructed by two identical OFDM symbols (denoted as T_1 and T_2 in Figure 3.7). This would demand a considerable amount of processing power. Hopefully, studies [44] have shown that only 28 taps can be used for the cross-correlation without any performance degradation.

Although coarse frequency synchronization process should have correct the CFO, this in not always true. Noise, multipath effects and carrier frequency drifts affects the estimation $\hat{\Delta}f$. The estimation error may have significant impact on the fine synchronization process. The reason lies on the structure of the long training sequence itself. While the short

training sequence was periodic with period $T_{FFT}/4$, the same is not true for the long training sequence. Unlike the short training sequence the long training sequence has now symmetry between the real and imaginary parts of the time domain. This means that the trick applied in coarse timing synchronization can not compensate the effect of the CFO on the fine synchronization process.

Even if the coarse frequency synchronization block made a perfect estimation of $\hat{\Delta}f$, there is another factor that until now remained hidden. This factor is the phase difference of the local oscillator of the receiver and the NCO. When the coarse frequency synchronization block reports an estimation $\hat{\Delta}f$, the frequency of the NCO is set to $\hat{\Delta}f$. Of course the produced cosine signal has a phase $\phi = 0$, but is not sure that the local oscillator may have the same phase. Applying an appropriate phase offset at the NCO is extremely difficult. It would require the knowledge of the phase not only of the local oscillator, but also of the transmitter oscillator.

The effect of phase offset on the fine synchronization output is presented in Figure 3.32. Depending the phase difference between the local oscillators and the NCO significant amplitude fluctuations can be obtained from the cross-correlation block. This make timing synchronization unfeasible because the process can not rely on a certain decision threshold.

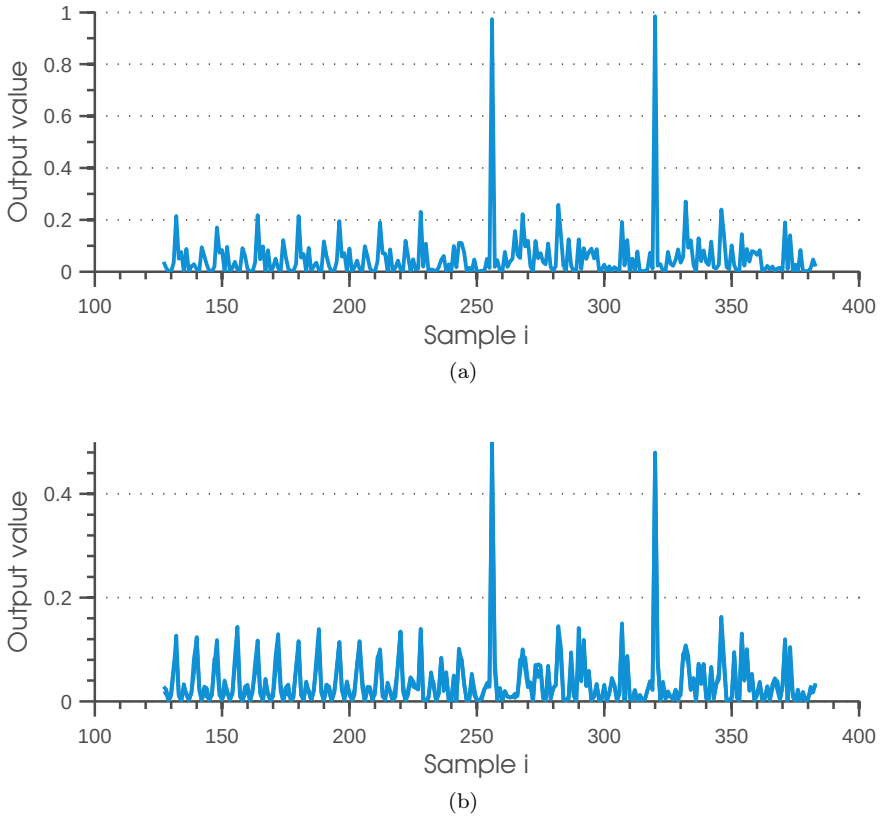


Figure 3.32: Normalized output of the Fine Synchronization algorithm (a) Without phase offset (b) With phase offset

A possible solution to this problem would be to compute the cross-correlation using the complex input samples. Unfortunately, as discussed before this is not applicable due to

the real time requirements. Performing FIR filtering in a complex input stream consumes too many resources that a single CPU thread can not serve in real time. For these reason, our approach tries to make use of FIR filters that operate on float input streams.

Normally, the output of the algorithm for the fine timing synchronization process using cross-correlation would be computed using

$$threshold(i) = |\sum_{j=0}^{D-1} y(i-D+j) \times coeffs(j)|^2 \tag{3.35}$$

where $y(i)$ is the complex input signal with the effects of the coarsely estimated CFO compensated and $coeffs(i)$ the filter coefficients derived from the real part of the long training sequence. Due to the fact that $coeffs(i)$ are real numbers, Equation 3.35 can be rewritten as:

$$\begin{aligned} threshold(i) &= |\sum_{j=0}^{D-1} Re\{y(i-D+j)\} \times coeffs(j) + Im\{y(i-D+j)\} \times coeffs(j)|^2 \\ &= |\sum_{j=0}^{D-1} Re\{y(i-D+j)\} \times coeffs(j)|^2 \\ &+ |\sum_{j=0}^{D-1} Im\{y(i-D+j)\} \times coeffs(j)|^2 \end{aligned} \tag{3.36}$$

From Equation 3.36 the following conclusion can be made. The desired threshold can be found by adding the powers of the two separate FIR filters. As discussed previously in Section 3.4.2.1, an FIR filter with float numbers as input and coefficients can operate in real time. Furthermore with this approach we eliminate the threshold fluctuations due to phase offset between the NCO of the coarse frequency offset correction and the local oscillators.

3.4.3 Frame Sampler and Fine Frequency Synchronization

Fine timing synchronization has as task to find the start of the first OFDM symbol of a frame, the SIGNAL symbol. After the point indicated by the synchronization process, samples that are received from the device should be grouped in OFDM symbols before they processed by the following blocks. Furthermore, each frame has a specific number of OFDM symbols. Frame sampler is responsible to split the input stream, beginning from the right sample , into OFDM symbols and propagate the appropriate number of symbols at the next blocks. Also, another responsibility of the sampler is to propagate the estimation of the long training sequence that is vital for the channel estimation and equalization process that is going to be presented at the next section.

At the time the start of SIGNAL symbol is indicated, samples of the long training sequence have been lost. For this reason the sampler block has a circular buffer that stores the D last incoming samples.

$$\begin{aligned} D &= \mathbf{sizeof}(T_1) + \mathbf{sizeof}(T_2) + \mathbf{sizeof}(GI) \\ &= 64 + 64 + 16 = 144 \text{ samples} \end{aligned} \tag{3.37}$$

The reason of storing also a part of the long training sequence guard interval $GI2$ is

the following. The start estimation of the SIGNAL symbol, may have an error. If the indication lies inside the GI of the SIGNAL symbol, there is no problem and the frame can be still decoded. But this offset from the ideal start point should be known to the equalizer in order to handle it accordingly. This is accomplished by passing the same amount of samples of the $GI2$ before the start of long training sequence T_1. Buffering D samples in a circular buffer makes it extremely easy. For better accuracy the mean of T_1 and T_2 is calculated using Listing 3.3.

```
1  /**
2   * Computes the estimate of the long training sequence.
3   * This is performed
4   * by averaging the two possible long training sequences
5   *
6   * @param out The output buffer
7   *
8   * @param buffer buffer with a part of GI2 and the two
9   * consecutive long training sequences with their CFO
10  * corrected.
11  *
12  * @param fft_length the size of the fft
13  */
14 static inline void
15 compute_long_sequence_estimate(gr_complex *out,
16                                gr_complex *circ_buffer,
17                                const size_t start,
18                                const size_t fft_length)
19 {
20   for(i = 0; i < fft_length; i++) {
21     out[i] = ( circ_buffer[(i + start) % fft_length]
22             + circ_buffer[(i + fft_length + start) % fft_length])
23             / 2;
24   }
25 }
```

Listing 3.3: Long training sequence estimation

After computing the long training sequence estimation $\hat{r}$, a dummy cyclic prefix is appended to it in order to have the size of an OFDM symbol. At this point, $\hat{r}$ and the symbols that apart the frame are propagated to the next blocks. One obvious question that needs to be asked is, how many symbols should be propagated? The sampler has no information about the length of the frame. This information exist in the coded data of the SIGNAL field. To deal with this problem our approach makes use of blocking asynchronous message queues. After sending the estimation $\hat{r}$ and the SIGNAL symbol, sampler blocks and waits to receive the frame length from the block that is responsible for decoding. After retrieving the symbols number of the current frame, the appropriate number of symbols are propagated until the end of the frame is reached, where the whole process is repeated. At the case that the decoder can not successfully decode the SIGNAL symbol, frame size is set to 0, causing the sampler immediately to enter the initial state, waiting the next new frame indication from the time synchronization process.

Another problem with this approach is that there are executions that deadlock may occur. This happens, when the SIGNAL symbol due to buffering has not yet sent, by the time the sampler thread blocks waiting for the frame size at the message queue. To ensure that this execution may never happen, after creating the possible SIGNAL symbol, our implementation explicitly informs the GNU Radio scheduler to propagate this symbol. This is accomplished by returning from the *general_work()* method immediately, indicating also the samples consumed and output OFDM symbols produced.

This approach ensures that any sample that may pass from the sampler is part of a possibly valid frame. In the case of a corrupted frame only the minimum amount of data are propagated at the next blocks, the $\hat{r}$ and the SIGNAL symbol. This simplifies the logic of the next processing blocks and reduces drastically unnecessary checks.

Furthermore, in order to reduce the number of the processing blocks, this block performs also the fine frequency synchronization. In order to retrieve the fine CFO estimation $\hat{\Delta}_{fine}f$ an estimation of the two consecutive long training sequences T_1 and T_2 is needed. Due to the fact that this block already has buffered them both, the $\hat{\Delta}_{fine}f$ can be derived using Equation 3.38.

$$\hat{\Delta}_{fine}f = \angle(\frac{\sum_{j=0}^{D-1} \hat{T}_1(j) \times \hat{T}_2^*(j))}{2 \times \pi \times T_s} \tag{3.38}$$

where $D = 64$ is the period of the long training sequence and T_s the sampling period.

After the estimation process, another NCO is used to compensate the frequency offset, as was performed in Section 3.4.2.2. Then the corrected samples become available to the next processing blocks.

3.4.4 Cyclic Prefix Removal

The next processing block of the receiver is the Cyclic Prefix Removal block, which is a quite straightforward processing block. It takes an OFDM symbol from the sampler that has duration of $T_{GI} + T_{FFT} = 80$ samples, discards the CP and returns 64 samples.

As discussed previously, the timing synchronization process may indicate the start of the frame in any possible sample that is part of the GI. For this reason the CP removal block should discard the last 16 samples of each symbol derived from the sampler block, as Figure 3.33 illustrates.

Concerning the long training sequence estimation $\hat{r}$ that the sampler block sends, the dummy samples that there were added will be discarded from this block. The reason that were added was to avoid unnecessary communication or checks and keep the CP removal process simple, as much as possible.

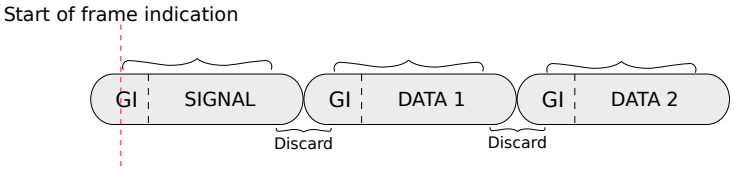


Figure 3.33: CP removal process

3.4.5 Fast Fourier Transform

With the GI removed, OFDM symbols that are in the time domain should be transformed again in the frequency domain. This is accomplished by using the FFT algorithm. It is exactly the inverse procedure that discussed in Section 3.3.5. In this case too, the FFTW [36] library was used.

3.4.6 Channel Estimation and Equalization

Wireless channels are a quite challenging and unpredictable medium of communication. The wireless channel can introduce such impairments on the system, that can affect each OFDM subcarrier differently. The effect of the channel across the bandwidth of interest is characterized by the frequency response H. Figure 3.34 provides the effects of a channel S

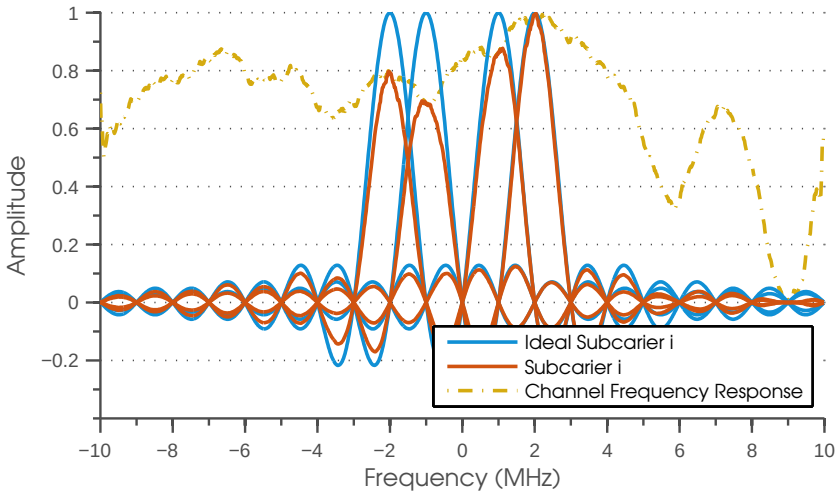


Figure 3.34: Effect of the channel on the OFDM subcarriers

across different subcarriers of an OFDM symbol. S was created using the *stdchan* function of MATLAB tool, provided in the Communications System Toolbox™using the 802.11a channel model. As can been seen from this figure, H is different across the 20 MHz bandwidth of the protocol. Each subcarrier i at frequency f_i is affected differently depending on the value of the frequency response of the channel at the corresponding frequency. This causes each subcarrier to have different amplitude from the expected. Furthermore, transmit or receive devices may have antennas and/or amplifiers with different gains causing the subcarriers to scale accordingly from the ideal amplitude. As a result, constellation points drift from the expected ranges making unfeasible the demodulation of the frame, as it is clearly shown in Figure 3.35.

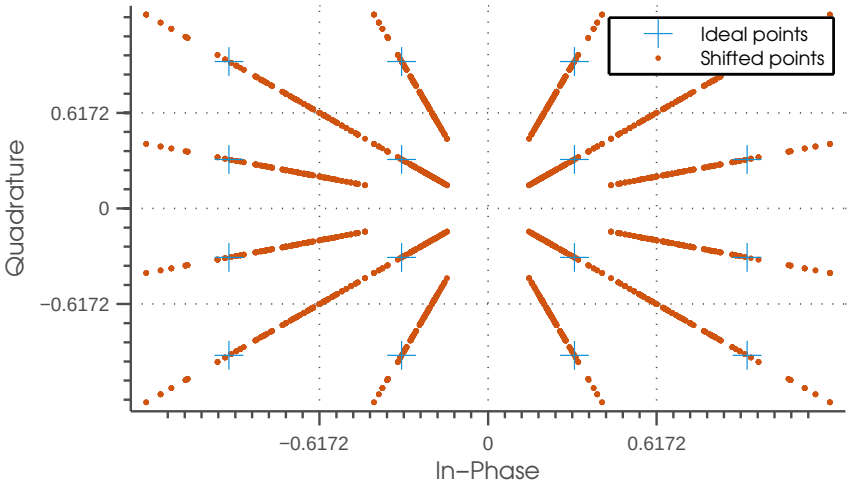


Figure 3.35: 16-QAM constellation without equalization

Channel estimation is the process of estimating the frequency response $\hat{H}$ of the medium, in which the signal is being transmitted. After the estimation $\hat{H}$ is retrieved, the equalizer tries to correct the implications of the channel to the signal. This ensures that the amplitude of each subcarrier will scale accordingly to a reference level. After that the decoder can decode the resulting constellation points.

Equalization also corrects the implications that are introduced if the start of each OFDM has an offset a from the ideal symbol start, provided this offset is inside the CP of the corresponding symbol. Figure 3.36 depicts the phase shift on the received signal that this offset introduces (red area on the plot). This phase shift due to the property 3.39 of the Fourier transformation will cause each constellation point to rotate and change its amplitude.

$$y(x-a) \xrightarrow{\mathcal{FFT}} e^{-j2\pi fa} \times X(f) \tag{3.39}$$

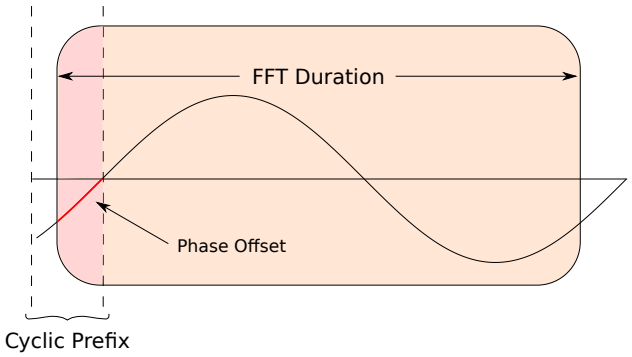


Figure 3.36: Phase offset introduced due to the offset from the ideal start of the OFDM symbol

Channel estimation can be achieved with several different methods. The main approaches that can be divided into two categories.

1. Time domain approach

2. Frequency domain approach

Both approaches in order to estimate the channel frequency response use information that is known in both transmitter and receiver. This known information depending the telecommunication system, is either a training sequence or pilots subcarriers. Each of them has its advantages and disadvantages. It's the responsibility of the developer to choose an approach that meets not only the performance requirements of the telecommunication system but the computational requirements that needed in order the implementation to be able to execute in real-time.

3.4.6.1 Time domain approaches

Time domain approaches are performed before the samples are passed at the FFT block. Instead of trying to find the frequency response of the channel $\hat{H}$, their goal is to estimate the channel impulse response $\hat{h}$.

The received signal $y(t)$ in the time domain can be expressed as

$$y(t) = h(t) \star x(t) + w(t) \tag{3.40}$$

where

$$h(t) = \begin{bmatrix} h(0) \\ h(1) \\ \vdots \\ h(N-1) \end{bmatrix}$$

is the channel impulse response, $x(t)$ the transmitted signal and $w(t)$ the noise. The convolution of $h(t)$ with $x(t)$ can be also written as matrix multiplication. Assuming

$$x(t) = G = \begin{bmatrix} x(0) & x(1) & \cdots & x(N-1) \\ x(0) & x(1) & \cdots & x(N-1) \\ \vdots & \vdots & \ddots & \vdots \\ x(0) & x(1) & \cdots & x(N-1) \end{bmatrix}$$

Equation 3.40 can be re-written as

$$y(t) = G \times h(t) + w(t) \tag{3.41}$$

so by ignoring the uncorrelated noise $w(t)$ the impulse response estimation can be derived by calculating

$$\hat{h}(t) = G^{-1} \times y(t) \tag{3.42}$$

Of course in order to be able to perform the estimation, G should be known at the receiver. G is constructed by the time domain of a known training sequence or pilots carriers.

Then the frequency response of the channel can be computed easily using the Fourier transformation of $\hat{h}$.

$$\hat{h}(t) \xrightarrow{\mathcal{FFT}} \hat{H}(f) \tag{3.43}$$

3.4.6.2 Frequency domain approaches

On the other hand, frequency domain approaches trying to estimate the frequency response of the channel after the samples have been processed by the FFT block. The frequency domain of Equation 3.40 is

$$Y = H \times X + W \tag{3.44}$$

where for the $k-$th OFDM symbol

$$Y_k(f) = \begin{bmatrix} Y_k(0) \\ Y_k(1) \\ \vdots \\ Y_k(N-1) \end{bmatrix}$$

is the samples derived from the FFT block,

$$X_k(f) = \begin{bmatrix} X_k(0) & 0 & \cdots & 0 \\ 0 & X_k(1) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & X_k(N-1) \end{bmatrix}$$

is a diagonal vector, with the main diagonal to hold the frequency domain of the samples that actually transmitted.

$$H_k(f) = \begin{bmatrix} H_k(0) \\ H_k(1) \\ \vdots \\ H_k(N-1) \end{bmatrix}$$

is the channel frequency response at frequency k and

$$W_k(f) = \begin{bmatrix} W_k(0) \\ W_k(1) \\ \vdots \\ W_k(N-1) \end{bmatrix}$$

is white Gaussian noise that can be assume that is independent from the channel H. If the noise W is ignored the channel estimation can be computed using Equation 3.45

$$\hat{H} = Y \times X^{-1} \tag{3.45}$$

Due to the fact that vector X is diagonal the inverse vector X^{-1} is

$$X_k^{-1} = \begin{bmatrix} \frac{1}{X_k(0)} & 0 & \cdots & 0 \\ 0 & \frac{1}{X_k(1)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \frac{1}{X_k(N-1)} \end{bmatrix}$$

so 3.45 can be re-written as

$$\hat{H}_k(i) = \frac{Y_k(i)}{X_k(i,i)}, \qquad 0 < i < N \tag{3.46}$$

Again, vector X_k must be known at the receiver. For this reason the frequency domain of a training sequence or that of pilots subcarriers is used.

3.4.6.3 Software Implementation

The approach that was used in our system, was a frequency domain approach exploiting the available by the IEEE 802.11a standard training symbols. More specifically, the frequency domain approximation $\hat{R}$ of the long training sequence was used. For better performance the mean of both long training sequences of the PLCP preamble T_1 and T_2 were used. Due to the fact that FFT is a linear transformation, the mean can be computed before the FFT. This is performed at the sampler as discussed previously. This trick greatly

simplify the flowgraph. It can be seen from Figure 3.37 that after the sampler block, CP removal and FFT are implemented using synchronous blocks. So only the channel estimation block needs an indication for the start of each frame. This is accomplished with a boolean stream derived from the sampler block. When the start of the frame is indicated, channel estimator receives $\hat{R}$ and calculates the channel frequency response estimation $\hat{H}$. To accomplish that, Equation 3.46 is used.

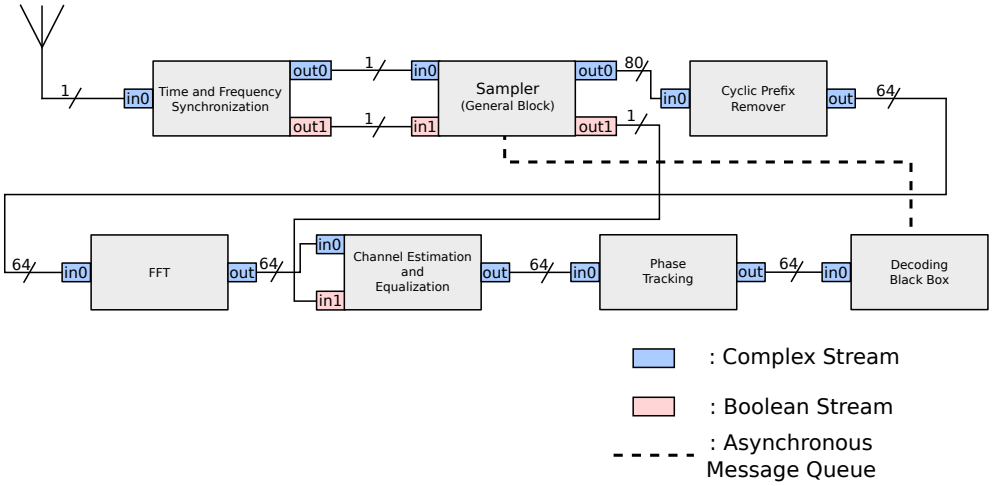


Figure 3.37: A bird's eye view of the IEEE 802.11a software receiver

After retrieving the $\hat{H}$ it is time for the equalization. This is performed for every OFDM symbol k of the frame using Equation 3.47.

$$equalized_k(i) = Y_k(i) \times \hat{H}_k(i), \qquad \begin{matrix} 0<k<\#(\text{frame symbols}) \\ 0<i<\#(\text{FFT bins}) \end{matrix} \tag{3.47}$$

The reasons that frequency approach was used were based on both the computational overhead and the flowgraph complexity. As discussed previously, channel estimation in time domain is a very demanding process due to the convolution operation involved. Furthermore, convolution would be required if the equalization was performed in the time domain. This would require a tremendous amount of processing power making unfeasible our implementation to be executed in real-time. Using a frequency domain approach, both channel estimation and equalization can be performed simultaneously, reducing the number of the required processing blocks. Moreover, Equation 3.47 reveals that the equalization is accomplished using one-dimensional vector multiplication. This makes the frequency domain approach ideal for exploiting the SIMD instructions. Complex multiplication can greatly benefit these instructions. For more information refer to section 5.1.

IEEE 802.11a operates on channels that assumed to be slow fading. The frequency response of these type of channels remains constant during the transmission of a frame. For this reason, channel estimation is performed only at the beginning of each frame. So there is no need the estimation process to performed multiple times while a frame is received.

On the other hand, realistic wireless channels are a very dynamic medium, so the channel frequency response may introduces small variations during the reception of a frame. The following section presents how the system can handle these variations, utilizing the four pilot subcarriers that each OFDM symbol provides.

3.4.7 Phase Tracking

In the previous sections, the two stage (coarse and fine) CFO estimation and compensation algorithms where discussed. One question that needs to be asked, however, is whether this estimation is accurate enough in order the whole OFDM frame to be able to be decoded.

There is always a small error in the CFO estimation. This error may vary depending the noise of the channel or multi-path effects. Furthermore, carrier frequency of both transmit and receive device may do not be constant in time due to fluctuation in local oscillators' frequency. This phenomenon is also known as phase noise [45].

This CFO estimation error in conjunction with the phase noise may be enough to rotate the constellation further enough, that the frame becomes impossible to decode. This happens commonly in frames that use high order constellations or in large frames with many OFDM symbols duration.

To overcome this problem, receivers exploit the pilot carriers. As was mentioned previously in Section 3.3.4.1, each OFDM symbol has 4 pilots BPSK modulated carriers. Furthermore, the value of each pilot carrier is known and can be easily calculated. The basic idea behind phase tracking, is to identify how much the known pilot carriers have been rotated. After computing the rotation angle $\hat{\theta}$, the constellation can be de-rotated by an angle of $-\hat{\theta}$ degrees.

3.4.7.1 Software implementation

Phase angle $\hat{\theta}$ estimation and correction is performed in a per OFDM symbol basis. Consequently, phase tracking can be a demanding task, so designing an efficient phase estimator and correction algorithm is a critical aspect of the system.

Regarding $\hat{\theta}$ estimation, one problem of the pilots carriers is that they have not the same value for every transmitted frame. The value of each pilot depends on the polarity that is assigned at every OFDM symbol based on the lookup table of Listing 3.2. So normally the phase estimator should keep track the polarity value. The phase estimation for the i-th OFDM symbol can be derived from Equation 3.48.

$$\begin{aligned}\hat{\theta}_i = \frac{1}{4} \times \big(& arg(polarity[i] \times P_{-21}) + arg(polarity[i] \times P_{-7}) \\ & + arg(polarity[i] \times P_7) + arg(-polarity[i] \times P_{21}) \big)\end{aligned} \tag{3.48}$$

where $arg(x)$ function returns the angle ϕ of the complex number x and P_j is received value of the pilot carrier j.

A closer look of pilots and polarity values reveals that polarity tracking may be redundant and can be avoided. Due to the fact that polarity values are 1 either -1, it could be possible to completely ignore the contribution of the polarity if the square of each pilot carrier is taken into consideration. If x is a complex number with angle θ then,

$$\left.\begin{aligned} x &= re^{j\theta} \\ arg(x) &= \theta \end{aligned}\right\} \quad \Rightarrow \quad \begin{aligned} x^2 &= re^{j2\theta} \\ arg(x^2) &= 2\theta \end{aligned} \tag{3.49}$$

Based on 3.49, Equation 3.48 can be rewritten as:

$$\hat{\theta}_i = \frac{\frac{1}{2} \times arg(P^2_{-21}) + \frac{1}{2} \times arg(P^2_{-7}) + \frac{1}{2} \times arg(P^2_{7}) + \frac{1}{2} \times arg(P^2_{21})}{4}$$

$$= \frac{arg(P^2_{-21}) + arg(P^2_{-7}) + arg(P^2_{7}) + arg(P^2_{21})}{8} \tag{3.50}$$

After obtaining the $\hat{\theta}_i$ the correction of the i-th OFDM symbol is accomplished using Algorithm 4.

```
for OFDM symbol i do
    θ̂_i = compute_phase();
    rotator = e^{-jθ̂};
    foreach sample j of the OFDM symbol do
        corrected[i, j] = rotator * input[i,j];
    end
end
```

Algorithm 4: Phase correction algorithm

Although this is the most comprehensive method to correct the phase without the need of tracking the polarity, it does suffer from a severe flaw. For the $\hat{\theta}$ estimation *std::arg(x)* function is used. As the C++11 standard [46] indicates, *std::arg(x)* has the same outcome with function *atan2(x.imag(), x.real())* whose behavior can be derived from Equation 3.51. The *arctan(x)* function has principal value in the range of $\left(-\frac{\pi}{2}, \frac{\pi}{2}\right]$, so the principal value of *std::arg(x)* will be in the range of $(-\pi, \pi]$.

$$atan2(y,x) = \begin{cases} arctan(\frac{y}{x}) & x > 0 \\ arctan(\frac{y}{x}) + \pi & y \geq 0, x < 0 \\ arctan(\frac{y}{x}) - \pi & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ undefined & y = 0, x = 0 \end{cases} \tag{3.51}$$

However, *atan2(y, x)* introduces a discontinuity at angle π, as Figure 3.38 illustrates. This means that when the constellation rotation due to phase offset exceeds π, the phase tracking mechanism fails to correct the constellation. The situation is even worse, when the proposed method without the polarity tracking is used. Based on the fact that the square of each pilot is used and 3.49, the phase offset limit is reduced at $\pi/2$.

Yet, one question that should be answered is the following. Is possible the constellation rotation angle to cross the above boundaries during the reception of a OFDM frame and if yes, what causes it? Firstly, the longer frame should be found in terms of number of OFDM symbols. Based on Table 3.1, SERVICE bits, the maximum value of LENGTH field and padding bits that may applied, the maximum number of OFDM symbols can be derived from Equation 3.52. Note that the maximum possible duration of a IEEE 802.11a

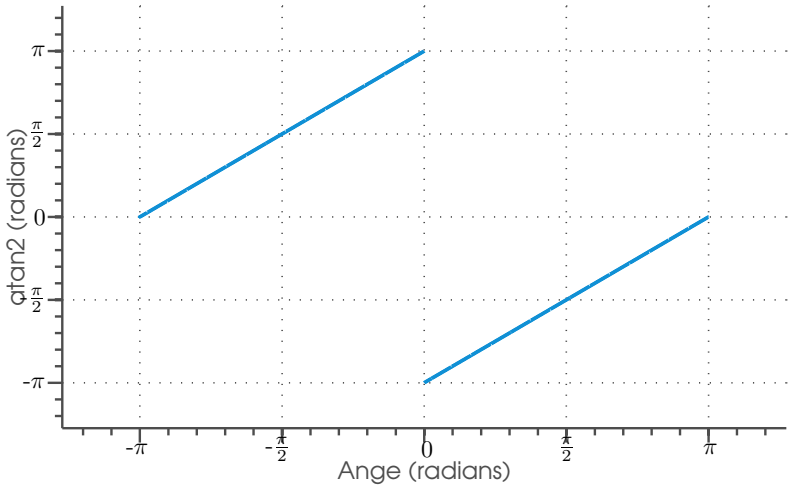


Figure 3.38: Discontinuity of the *atan2(y, x)* function

frame is obtained using the most conservative modulation scheme (BPSK, $R = 1/2$).

$$max_{\#symbols} = \left\lceil \frac{(16 + 4095 \times 8 + 6)}{24} \right\rceil + \text{SIGNAL symbol}$$
$$= 1367 \tag{3.52}$$

After obtaining the maximum possible frame length, the impact of the phase noise is analyzed. To achieve this the following method was used. In order the simulation to be realistic as much as possible, the phase noise characteristics of the synthesizer ADF4351 [47] at the frequency of 2.2 GHz where used (Table 3.12). This is the transmit and receive synthesizer of the Noctar [28] that was used during this work. After creating $max_{\#symbols}$ OFDM symbols at baseband, we up-convert them to the target frequency $f_{up} = 2.2GHz$ applying the phase noise characteristics of ADF4351 synthesizer. Then, the up-converted signal was down-converted using $f_{down} = -f_{up}$ and the same phase noise characteristics. Figure 3.39 depicts the resulting constellation. This figure clearly shows that the rotation of the constellation due to the phase noise of the local oscillators is negligible ($\theta \approx 0.03$ radians). So phase noise can not impose such a constellation rotation for the phase tracking mechanism to fail.

Table 3.12: Phase noise characteristics of ADF4351 at 2.2 GHz

Phase noise (dBc/Hz)	**Offset from 2.2 GHz carrier**
−89	10 kHz
−114	100 kHz
−134	1 MHz
−148	5 MHz

On the other hand, CFO as discussed in the previous sections, also rotates the constellation points. Despite the fact that IEEE 802.11a performs two phase frequency synchronization, there is always the chance of error in the $\hat{\Delta}f$ estimation. In this case however,

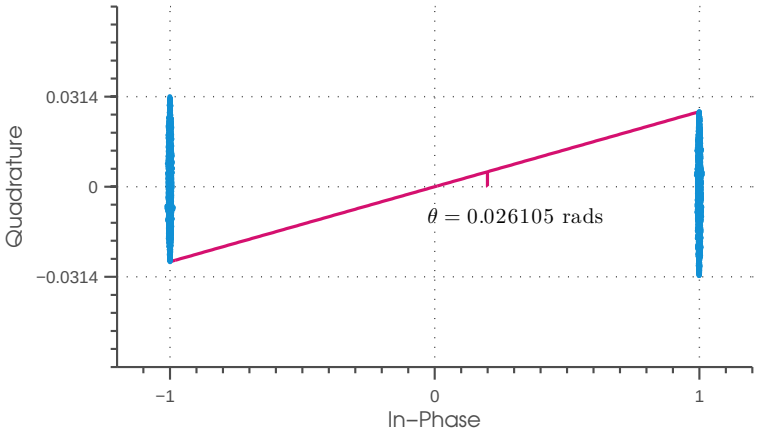


Figure 3.39: BPSK constellation rotation due to phase noise

the rotation angle of each symbol is predictable. Assuming that the CFO remains constant in time, the rotation of each OFDM symbol will be also constant. Having found the maximum number of symbols, the maximum possible rotation of the i-th OFDM symbol from the previous $(i-1)$-th can be computed using 3.53. Based on that the maximum CFO estimation error that the phase tracking mechanism can afford is derived from 3.54.

$$rotation_i^A < \frac{\pi}{max_{\#symbols}} \simeq 0.002 \text{ rad} \tag{3.53}$$

$$\begin{aligned} 2 \times \pi \times \hat{\Delta} f \times T &= rotation_i^A \\ \hat{\Delta} f &= \frac{rotation_i^A}{2 \times \pi \times T} \\ \hat{\Delta} f &< 91.4 \text{ Hz} \end{aligned} \tag{3.54}$$

where $T = 4\mu s$ is the duration of the OFDM symbol (Table 3.9).

Respectively, if the approach without polarity tracking is used, the maximum allowed rotation is $\frac{\pi}{2}$. So the new maximum rotation per symbols and the maximum affordable CFO are decreased by half.

$$rotation_i^B < \frac{\frac{\pi}{2}}{max_{\#symbols}} \simeq 0.001 \text{ rad} \tag{3.55}$$

$$\begin{aligned} 2 \times \pi \times \hat{\Delta} f \times T &= rotation_i^B \\ \hat{\Delta} f &= \frac{rotation_i^B}{2 \times \pi \times T} \\ \hat{\Delta} f &< 45.7 \text{ Hz} \end{aligned} \tag{3.56}$$

These findings indicates that even small CFO estimation errors may affect the phase tracking mechanism. Such errors are quite common, especially in noisy and multipath environments. A solution to this problem would be the use of a two stage phase offset

correction algorithm. Based on the fact that the phase ϕ caused by the CFO estimation error is additive for each consecutive OFDM symbol of a frame, Algorithm 4 can be enhanced to overcome the discontinuity problem of the *atan2(y, x)* function. Algorithm 5 provides an approach, which corrects the current symbol i with the phase of the previous $i-1$ symbol, before it estimates the rotation of the symbol i. With this method, the rotation does not reach the discontinuity boundary of the $atan2(y, x)$ function.

```
for Frame n do
    rotator = 0;
    for OFDM symbol i do
        /* Rotate the constellation with the angle of the previous i-1
           symbol                                                    */
        foreach sample j of the OFDM symbol i do
            corrected[i, j] = rotator * input[i,j];
        end
        /* Perform the phase estimation at the corrected constellation
           */
        θ̂_i = compute_phase(corrected);
        rotator = e^{-jθ̂};
        foreach sample j of the OFDM symbol i do
            corrected[i, j] = rotator * input[i,j];
        end
    end
end
```

Algorithm 5: Improved Phase correction algorithm

On the other hand, the two stage phase tracking algorithm uses twice the number of complex multiplication operations. In systems with large numbers of OFDM sub-carriers this can be a problem and the polarity tracking approach may be more suitable. Regardless the approach chosen, the high throughput of SIMD instructions can be again exploited for the phase correction process, using the vectorized complex multiplication procedure that is presented in Section 5.1. Our implementation makes use of the two stage phase tracking mechanism.

3.4.8 Constellation De-mapping

So far all the processing blocks that have been described are related with the received signal processing and how the system can overcome the impairments of the wireless medium. This section describes how the information that the received signal carries, is transformed into bits of information. The procedure of converting constellation points that are scattered across the z-plane into bits, is known as constellation de-mapping or demodulating. Processing blocks that are executed after the constellation de-mapping are handling only bit streams.

The responsibility of the constellation de-mapping block is to decide for every constellation point received at which bit set corresponds, according to the modulation scheme that is used. The bit sets that each constellation point corresponds for the four different modulation schemes can be derived from Figure 3.19. In an ideal system, received constellation points of a frame should be at the same position that were transmitted. Having discussed so many impairments, it is clear that this will never happen. In real environments constellation points are scattered around the ideal point. How far can be from the ideal point and still the frame to be able to be decoded depends on the modulation

scheme that the frame is modulated. Low order constellations like BPSK may carry a small amount of information, but they are tolerant in large offsets from the ideal points. The same is not true for high order constellations like QAM-64. Even small offsets may lead to unrecoverable errors. For this reason IEEE 802.11a provides different modulation schemes that are chosen based on an adaptive algorithm that takes as feedback the SNR and the number of failed or successful transmissions.

After the phase tracking block, samples of every $k-$th OFDM symbol are received by the constellation de-mapping block in groups of N_{ST} complex samples, denoted as

$$Y_k = \begin{bmatrix} Y_k(0) \\ Y_k(1) \\ \vdots \\ Y_k(N_{ST}-1) \end{bmatrix}$$

From them only the N_{SD} contain data, the other are either zero or pilots carriers. For this reason, the constellation de-mapping block uses a lookup table in order easily and fast retrieve the index of the right data subcarrier. For simplicity these carriers will be mentioned as

$$D_k = \begin{bmatrix} D_k(0) \\ D_k(1) \\ \vdots \\ D_k(N_{SD}-1) \end{bmatrix}$$

In order the de-mapping block to decide which constellation point actually transmitted, a simple approach would be the following. For each constellation point $D_k(i)$ received, compute the Euclidean distance (Equation 3.57) d_i for every reference constellation point r_j. The reference point r_j with the the minimum Euclidean distance is chosen as the best estimate for the actually transmitted constellation point. Of course this is a very demanding process requiring $O(n^2)$ computations. Assuming 64-QAM is used $N_{SD} \times 64 = 3072$ calls of the Euclidean distance procedure are required for every OFDM symbol received. Hopefully this is not the optimal solution.

$$d(p,r) = d(r,p) = \sqrt{\Big(Re(p) - Re(q)\Big)^2 + \Big(Im(p) - Im(q)\Big)^2} \tag{3.57}$$

Again, a two dimensional lookup table is utilized for fast and efficient de-mapping, with the first dimension corresponding to the real part and the second to the imaginary. To achieve that, the received constellation points that are floating point numbers lying on the range of $[-1, 1]$ are converted to appropriate integer numbers that can be used as index on a lookup table. Firstly, the effects of the normalization performed in the mapping process (Equation 3.8) is compensated by multiplying the constellation points with the inverse of the K_{MOD} factor, $1/K_{MOD}$. After that, our approach ensures that both the real and imaginary part of the constellation points are not exceeding the range of the corresponding modulation scheme. This is performed to protect our system from a possible memory access violation, due to wrong array indexing. To restrict the values of the received constellation points to the appropriate range, Equations 3.58 and 3.59 are

used.

$$Re\{d\} = min(constellation_{max}, max(constellation_{min}, Re\{D_k(i)\} \times \frac{1}{K_{MOD}})) \tag{3.58}$$

$$Im\{d\} = min(constellation_{max}, max(constellation_{min}, Im\{D_k(i)\} \times \frac{1}{K_{MOD}})) \tag{3.59}$$

Finally to retrieve the right lookup table index, Equation 3.60 is used

$$index = roundf(\frac{x + constellation_{max}}{2.0}) \tag{3.60}$$

where $x = Re\{d\}$ for the first lookup table dimension and $x = Im\{d\}$ for the second one.

With this technique the de-mapping is performed without calculating the Euclidean distance. Moreover, all the necessary operations (min, max, multiplication, roundf) are available for the majority of the SIMD extensions, thus the de-mapping process can gain significant performance improvements by utilizing them.

3.4.9 De-Interleaving

Having described at Section 3.3.3 how the block interleaver of IEEE 802.11a performs the permutations of the bits of each OFDM symbol, it is time to describe how the receiver place the interleaved bits again at the proper order. This is accomplished based on the permutations defined by Equations 3.61 and 3.62.

Assuming that j is the index of the received bit derived from the demapper block, i is the index after the first and before the second permutation. Also, k is the index after the second permutation before the block of N_{CBPS} bits are delivered to the convolutional decoder. The first permutation of the de-interleaver, which is the inverse operation of Equation 3.4, is defined by

$$i = s \times floor(j/s) + (j + N_{CBPS} - floor(16 \times j/N_{CBPS})) \bmod s \qquad j = 0, 1, \ldots, N_{CBPS} - 1 \tag{3.61}$$

where s is derived from Equation 3.5.

The second permutation can be determined from

$$k = 16 \times i - (N_{CBPS} - 1) \times floor(16 \times i/N_{CBPS}) \qquad i = 0, 1, \ldots, N_{CBPS} - 1 \tag{3.62}$$

This is the inverse permutation of Equation 3.3.

Likewise interleaving, de-interleaving process is based also in lookup tables. Pre-calculated permutations for every modulation scheme are stored in lookup tables. As mentioned at the previous section, de-mapper block propagates data in a per OFDM symbol basis. This is very convenient because the block de-interleaver threats the N_{CBPS} bits of each symbol independently. Furthermore, in order the de-interleaver to make use of the proper permutation lookup table, it takes advantage of the additional information that is sent by the de-mapper block for every first OFDM symbol of each frame received. With the knowledge of the modulation scheme and the duration of the frame, the appropriate permutation tables can be used allowing the successful bit de-interleaving of each OFDM symbol.

Each group of N_{CBPS} processed, is delivered to the convolutional decoder block using an asynchronous message queue. Again, in order the next block to be able to decode the data properly, the modulation scheme and the length of the frame are encapsulated at the data of every first group of N_{CBPS} bits of each received frame.

From a first look, like in the case of interleaver, seems that except the lookup tables there is no other way to boost more the de-interleave process. However, it is known that the second permutation of the interleaver shuffles the bits of each constellation point separately. The de-interleaver at the first permutation reverts this shuffling using Equation 3.61. Based on the fact that this permutation is independent across the different constellation points, the process of the first permutation at the de-interleaver can be bypassed. To achieve this the constellation point to bit set mapping was altered accordingly. Using this optimization, each group of N_{CBPS} bits derived from the de-mapper has already its bits de-interleaved for the first permutation. Only the second de-interleave permutation should be applied, greatly reducing the overhead of this processing block.

3.4.10 Forward Error Correction

As was pointed out in the Section 3.3.2, the convolutional encoder adds redundant bits in the original bit stream in order the receiver to be able to recover from possible errors introduced by the wireless channel impairments. The purpose of this section is to present how the receiver can take advantage of these redundant bits and retrieve the original data without errors, a process also known as Forward Error Correction (FEC).

For the decoding of the convolutionally encoded data the well known Viterbi algorithm [31, 48] was used with the help of Trellis Diagrams.

3.4.10.1 Trellis Diagrams

Trellis diagrams are very useful in the decoding process of a convolutionally encoded bit stream. These diagrams graphically represent the encoding process. As can be seen from Figure 3.40, Trellis diagram is a graph, with each node to represent a possible state of the $K-1$ memory bits of the encoder. The 2^{K-1} states are placed vertically. Moreover, there are two type of transitions between states, also referred as branches. Either a 0-type transition can be performed if a 0 bit is received at the encoder (dashed lines), either a 1-type (solid line) in the presence of 1 at the input of the encoder. Among with each branch the encoder produces a 2-bit output based on its generator polynomials.

For each time step, the trellis diagram accommodates a column of 2^{K-1} states. The index of the time step is often mentioned as *trellis depth*. Every state of depth i is connected with 0- and 1-type branches with the states of depth $i+1$ forming a directed acyclic graph. Every path of the trellis diagram represents a possible message. For each message the resulting output of the encoder can be easily retrieved based on the 2-bit output of the corresponding 0 or 1 branch.

3.4.10.2 The Viterbi decoder

The Viterbi algorithm is one of the most widely used FEC decoders. It is a dynamic programming algorithm finding applications in telecommunications, speech recognition, Optical Character Recognition (OCR) and many others. For the purposes of FEC, its goal is to obtain the best estimation of a sequence actually transmitted based on the received sequence. This is accomplished by constructing the trellis diagram of the convolutional code of the system. After that, the decoder receives the coded bits trying to find the

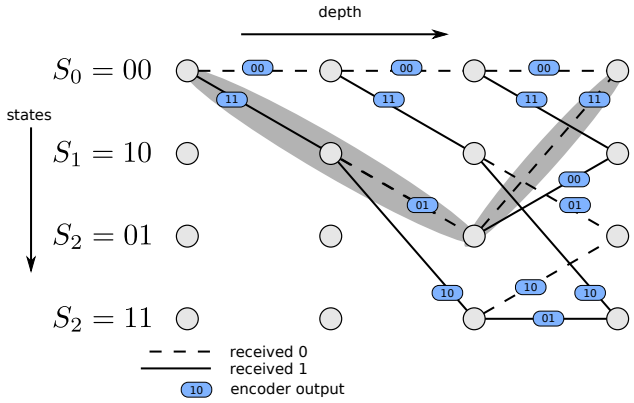


Figure 3.40: Trellis Diagram of the convolutional encoder of Figure 3.10. The highlighted path indicates that the input message 100 is encoded into 110111

path in the trellis diagram that corresponds to the best estimation of the original message transmitted, prior encoding.

To achieve this, the decoder utilizes two metrics for the best path estimation.

1. **Branch error metric** indicates the similarity of the received bits with the corresponding output of a branch of each state. The metric D for comparing similarity that our approach uses is the hamming distance [49].

2. **Accumulated error metric** is the minimum accumulated branch metrics that reach the corresponding state from the initial state at depth 0.

Our Viterbi decoder software implementation uses a custom variant of trellis diagram that has minimum memory requirements. For each depth level of the diagram, only the accumulated error and a pointer to the state from which the current state was reached is stored. Furthermore for performance reasons each state has the pre-calculated information both of the destination state and the corresponding output, either if zero or one is received. With this approach the traversal of the trellis diagram is accomplished very fast and efficiently. The structure of each of the 64 trellis states that IEEE 802.11a has, can be derived from Listing 3.4.

```
1  struct trellis_diagram_state
2  {
3    /* State number*/
4    unsigned char current_state;
5
6    /* Output if a zero is received at the current state */
7    unsigned char out_if_zero_received;
8
9    /* Output if a one is received at the current state */
10   unsigned char out_if_one_received;
11
12   /* One of the two possible previous states that reached this state */
13   struct trellis_diagram_state *branch[2];
14
15   /* Next state if a zero is received */
16   struct trellis_diagram_state *zero_received;
17
18     /* Next state if a one is received */
19   struct trellis_diagram_state *one_received;
20
21     /* The accumulated error metric of each state at a certain depth */
22   unsigned short int acc_error_metric[TRACEBACK_DEPTH + 1];
```

```
23
24    /* From the two branches that reach this state, which survived?? */
25    struct trellis_diagram_state *survivor_branch[TRACEBACK_DEPTH + 1];
26  };
```

Listing 3.4: Trellis diagram structure

Before proceeding to describe how the Viterbi algorithm creates and makes use of the trellis diagram, lets make some assumptions to simplify the algorithm description. For now, the coding rate that is used is $R = 1/2$. This means that for 2 coded bits, one bit of information will be extracted. In addition, the initial state is the zero state. How our approach handles the punctured coding rates will discussed later.

Starting from an initial 0 depth, the incoming coded bit stream is handled in a per 2-bit basis. For every 2 coded bits received, the trellis diagram is expanded by one step, leading to an increased by one depth.

As Figures 3.40 and 3.41 reveal, each state at depth i may be reached from only two other states of depth $i - 1$. The algorithm chooses to keep track only one of them, and eliminate the other. This is accomplished by adding the previous accumulated error metric of the corresponding state to the new branch metrics, comparing the results and selecting the smaller accumulated error metric. The branch, commonly called *survivor branch*, that contributed to this smallest accumulated error is saved, whereas the other is eliminated. The entire process is also known as *Viterbi Butterflies* due to its the graphical representation, depicted in Figure 3.41.

At the end of the frame processing, the accumulated error metric of each state indicates how similar is the output sequence created by the trellis with the received one. Choosing the state with the minimum accumulated error and traversing back the survivor transitions until depth 0 is reached, the best estimation of the original data prior encoding is retrieved. This process is also known as *traceback*.

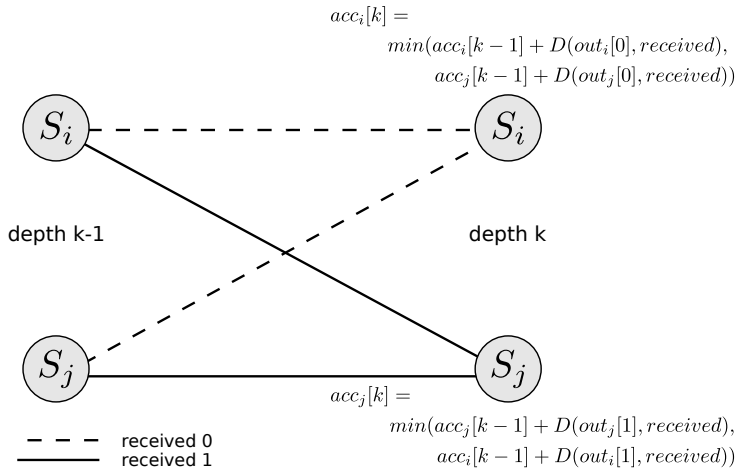


Figure 3.41: Viterbi butterflies

Theoretically, the traceback process can be triggered at the end of the bit stream of each frame. However, this would require a lot of memory resources. Although nowadays memory consumption is not such a big issue, following processing blocks will have to wait the Viterbi decoder to finish the traceback process, before they can acquire access to the decoded data. In order to exploit parallelism and be able following blocks to process some portion of the frame's data, traceback is performed more regularly during the processing of

a frame. Once a certain amount of processed coded bits is reached, traceback is performed making immediately the decoded data available to the next processing block. After that, the trellis diagram is reset. The process is repeated until the end of the frame is reached, where a traceback is trigger explicitly to decode possible left bits. Following a similar to the previously described blocks approach, the first traceback of every frame piggybacks the frame length among with the decoded data in order the descrambler block to be informed for the arrival of a new frame and its length.

For higher throughput, our approach uses two threads, one for the Butterflies computation and the other for the traceback, as Figure 3.42 depicts. Based on the fact that the trellis diagrams between two consecutive tracebacks are independent, the traceback process can be performed in parallel, while the algorithm continues processing the Viterbi butterflies for the remaining of the frame.

To support the punctured coding rates $R = 2/3$ and $R = 3/4$, most of hardware or software implementations insert dummy zero bits at the appropriate places of the input bit stream. However, our implementation follows a different approach avoiding a redundant processing block. During the processing of frames with punctured code, the hamming distance logic is different depending which bit was stolen during the puncture process. Figures 3.12 and 3.13 revealed that either the first bit A_i or the second B_i of the encoder output can be omitted. On the other hand, each trellis diagram transition produce a 2-bit output, so a dummy bit is placed at the appropriate place. Depending the position of the dummy bit, the hamming distance procedure should omit the difference between the dummy and the corresponding bit of the trellis output. This can be easily accomplished using the procedures of Listing 3.5.

```
1  static inline uint8_t
2  hamming_distance_Ai_puncture(const uint8_t trellis_out,
3                               const uint8_t received)
4  {
5    return bit_count((trellis_out ^ received) & 0b01);
6  }
7
8  static inline uint8_t
9  hamming_distance_Bi_puncture(const uint8_t trellis_out,
10                              const uint8_t received)
11 {
12   return bit_count((trellis_out ^ received) & 0b10);
13 }
```

Listing 3.5: Hamming distance with punctured support

Despite our efforts, the Viterbi decoder was not able to execute in realtime. Each BPSK modulated OFDM was decoded in approximately 8 μs, whereas for a 20 MHz channel should be below 4 μs. In order to meet this time constraint, again the SIMD instructions were utilized. The SIMD assisted variant is presented in Section 5.1.4.

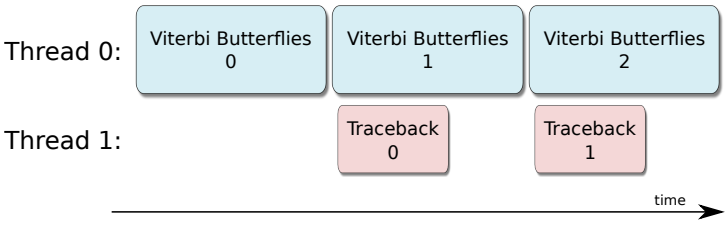


Figure 3.42: Parralel Viterbi decoding and traceback using two threads

3.4.11 Descrambling

De-scrambling is the last processing block of the IEEE 802.11a receiver. Its implementation is exactly the same with the scrambler based on the XOR properties. Knowing that

$$(p \oplus q) \oplus r = p \oplus (q \oplus r), \tag{3.63}$$

$$p \oplus p = 0 \tag{3.64}$$

and

$$p \oplus 0 = p \tag{3.65}$$

its easy to understand that if the scrambled data may again XORed with the same scrambling sequence, the original data prior scrambling will be retrieved.

One major problem that has to be bypassed is that the receiver does not know the scrambling sequence that the transmitter used to scramble its data. Even if our scrambler implementation used always the same scrambling sequence for performance reasons, this does not apply for regular IEEE 802.11a devices. In order the receiver to retrieve the scrambling sequence the standard provides the SERVICE field. This 2 byte field as discussed previously, has its 6 first bits explicitly set to zero prior scrambling. Based on Equation 3.65, we can conclude that these 6 bits of the SERVICE field will contain the same first 6 bits of the scrambler. Having the information of the initial bits of the scrambler is easy to produce the rest of the scrambling sequence using the same algorithm described by Figure 3.9.

The descrambler blocks waiting for data from the Viterbi decoder. For every data chunk retrieved, it checks if its is the first data chunk of the frame. If this is the case, the piggybacked frame length is retrieved and the scrambling algorithm is initialized with the 6 first bits of the SERVICE field. Then, for every data chunk retrieved, the corresponding scrambling sequence is produced and XORed with the received data. When the whole frame is processed the data are ready to be propagated towards the Media Access Control (MAC) layer. In the next chapter an interface will be presented that allows the PHY layer to pass the received data to the MAC layer, in order the OS to handle the rest of the network operations required (routing, filtering, port forwarding, e.t.c).

3.5 Sample Rate Conversion using FFT

Many SDR hardware devices, support only a limited and predefined number of different bandwidths that can operate. This is often related with hardware capabilities and restrictions. Unfortunately, telecommunication standards require the bandwidth of the SDR device to be exactly the same with the design specifications. In cases where this is not applicable, Sample Rate Conversion (SRC) is applied. However, conventional SRC methods use high order FIR filters to re-sample the processed signal to the target bandwidth of the device. These methods require a significant amount of computational resources [50], making realtime implementation of high bandwidth protocols practical unfeasible.

OFDM systems have a certain number of subcarriers N, equally spaced withing the channel's bandwidth BW. Therefore, the subcarrier spacing Δf can be derived from:

$$\Delta f = \frac{BW}{N} \tag{3.66}$$

After the IFFT the resulting time domain OFDM symbol has a duration of T_{FFT}, where

$$T_{FFT} = N \times f_{SR} \tag{3.67}$$

and

$$f_{SR} = \frac{1}{BW} \tag{3.68}$$

Assuming the SDR device provides a bandwidth BW_{SDR} conforming with (3.69), the same symbol time duration can be achieved by changing the number of the FFT bins.

$$BW_{SDR} \geq BW \tag{3.69}$$

The new number of the subcarriers N' can be derived from Equation 3.70.

$$N' = \frac{BW_{SDR}}{\Delta f} \tag{3.70}$$

By properly increasing the subcarriers number N', the resulting symbol duration remains the same, equals to T_{FFT}. In order the keep the spectral content integrity unchanged, the $N' - N$ additional subcarriers should be set to zero, equally distributed left and right to the original carriers before the IFFT procedure.

Most of the OFDM systems, incorporate a CP, which is a repetition of a portion of the time domain samples of each OFDM symbol. IEEE 802.11a uses a CP that has the duration of the $\frac{1}{4}$ of the symbol duration. This corresponds to $\frac{N}{4} = 16$ time domain samples. Accordingly if FFT SRC is applied, $\frac{N'}{4}$ time domain samples are used for the CP.

Another problem that arises from FFT SRC is the synchronization process during reception. In Section 3.4.2, synchronization methods that use time domain samples of the training sequences presented. With the a bandwidth different from the standard specifications, these synchronization methods can not be applied. However, the time domain of the training sequences can be re-calculated from the frequency domain using N' carriers and appropriate zero padding. The resulting time domain samples can be used for synchronization without any further modifications.

FFT SRC provides great flexibility at both transmitter and receiver. Applying such SRC method, introduces also another great feature to our system. The half- and quarter-clock additional operation modes of the IEEE 802.11a standard are supported out of the box by adjusting only the number of FFT subcarriers. In the case of the 10 MHz operation the subcarriers should be doubled, whereas in the 5 MHz mode they should quadrupled.

Chapter 4

Connection with the IEEE 802.11 Medium Access Control (MAC) Layer

The IEEE 802.11 Media Access Control (MAC) layer is responsible for managing a great part of the standard. It implements many tasks, including but not limited to proper and fair medium access, Acknowledgments (ACKs) and re-transmission mechanism, STA management, security and many others.

This chapter describes how our PHY implementation manage to co-operate with a IEEE 802.11 compliant MAC layer implementation and provides most of the features required for a proper communication.

4.1 General Description

Implementing from scratch a software implementation of the MAC layer can be a serious task, bigger than the PHY itself. On the other hand, a very basic implementation would be possible, but in lack of functionality, features and compatibility. While our approach has as goal the compatibility with commodity wireless devices that implement the IEEE 802.11a, our approach should accommodate a compatible MAC layer implementation. The solution came with the mac80211 [51], a Linux driver that implements most of the MAC layer functionality.

4.2 The mac80211 Driver

Over the last years, even more vendors have decided to follow a *SoftMAC* approach for their wireless devices. The term *SoftMAC* has been applied to situations where the Media Access Control (MAC) Sublayer Management Entity (MLME) is expected to be managed in software. Some of the tasks of the MLME are:

- Authenticate - Deauthenticate
- Associate - Disassociate - Reassociate
- Beacon broadcast

- Probe
- Timing Synchronization Function (TSF)

However, it is worth noting that time critical operations, like ACKs, re-transmissions and CSMA are still implemented in hardware.

The Linux *SoftMAC* solution is the mac80211 driver. This driver provides an API that can be used from the hardware driver in order to transmit and receiver data properly. For dealing with the user defined settings a second helper kernel module is used, the cfg80211. It is used as a middleware between the userspace and the mac80211 driver, allowing the user to alter several parameters of the standard. Such parameters may be the supported data rates, transmission power, operating mode (managed, ad-hoc, mesh, monitor) and many others. Each *SoftMAC* device, derives data and configuration using the API of mac80211. In addition, hardware devices report back data to the mac80211 driver, which is responsible to propagate them at upper layers. The architecture is summarized in Figure 4.1.

This approach has many advantages. First of all, hardware is more simple and thus cheap. Also, the same hardware device can operate in different modes without relying if its driver supports it. In addition, *SoftMAC* allow a unified way of configuration giving the ability to the OS to handle 802.11 wireless devices robustly and efficiently.

Taking advantage of the *SoftMAC* capabilities, our approach uses the mac80211 driver to provide to the PHY software implementation, fully compliant MAC layer capabilities. A great feature of this approach is that the OS completely ignores that the PHY actually runs on software and identifies the wireless interface as a normal one, with all the capabilities of parameterization enabled. However, in order to achieve that there is a need for an interface of communication between the mac80211 and the PHY software. The next section that follows, describes how our approach achieves that by providing a second kernel module, that acts like a glue between the mac80211 and the PHY software.

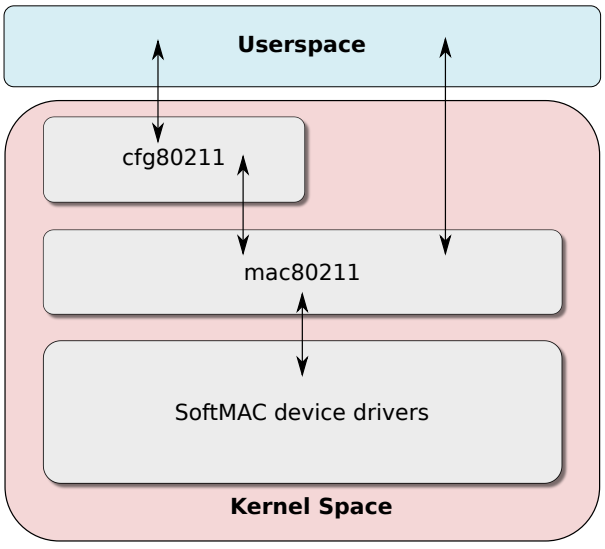


Figure 4.1: The mac80211 architecture

4.3 The ssdr Driver

The PHY software implementation encodes a PSDU derived from the MAC layer and transmits it over the air. Likewise, during reception it propagates the decoded PSDU towards the MAC. Running on userspace, the PHY software should be able to retrieve and send the PSDU data to the mac80211 driver. The Super Software Define Radio (ssdr) driver is a kernel module that its main purpose is to allow the data exchange between the userspace PHY implementation and the mac80211 driver as illustrated in Figure 4.2.

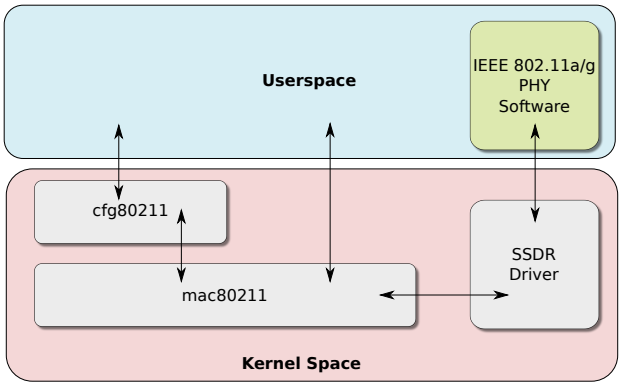


Figure 4.2: The mac80211 with the ssdr architecture

The ssdr driver provides a character device for each virtual interface that is created. Writing to this character device allows data to be delivered at the mac80211 driver. Respectively, PSDUs are retrieved by reading from the character device. This is a very convenient and efficient solution, because the PHY reads and writes PSDU data as it would read them from a regular file. Furthermore, to simplify even more the data exchange between the ssdr and the PHY, the following conventions where made. Data exchanged are equal in size. The size of the fixed size data chunk can be derived based on the Maximum Transmission Unit (MTU) of the interface that is always less that the maximum allowed PSDU. While reading or writing a fixed size data chunk is very convenient, somehow the actual length of the PSDU should be known. This information among with additional information required by both PHY and mac80211 is encapsulated at the beginning of each fixed data chunks exchanged.

The format of each fixed size data chunk during transmit and receive operations is presented at Figures 4.3a and 4.3b respectively. The data rate of the frame and the length of the PSDU should be known at both sides. The data rate is used by the PHY in order to use the appropriate modulation scheme, whereas the mac80211 uses this information to feed its rate control algorithm. The *Needs ACK* field is indicating that the current PSDU should be acknowledged by the other side. When the PHY transmitter handles a frame that has to be acknowledged, it should enable the re-transmission mechanism for this specific frame. Details of this mechanism will be presented at the following section. In addition, during a PSDU reception, the PHY calculates and reports back to the MAC the RSSI of the frame placing its value to the corresponding field of the ssdr header. The mac80211 makes use of this in order to report back to userspace applications an estimation of the connection quality.

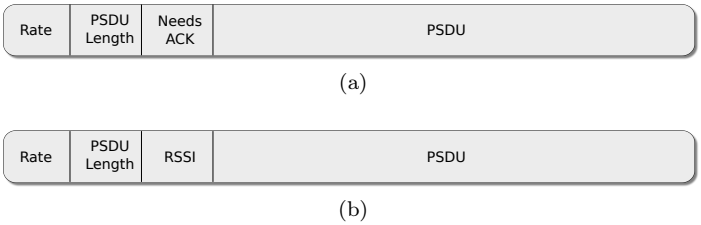


Figure 4.3: Structure of PSDU data: (a) retrieved by the ssdr driver (b) PHY to the ssdr driver

4.4 MAC Functions implemented at the PHY

So far all the operation that described where accomplished by the mac80211 in software. On the other hand, as noted previously, time critical operations are still implemented in hardware. These operations are completely ignored from the mac80211 driver expecting the hardware to implement their functionality. While in our case their is not such hardware, these operations should be implemented in the PHY software in order to achieve minimum response times. These operation fall under three headings: (1) CSMA, (2) ACKs and (3) re-transmissions.

The well-known CSMA mechanism allows multiple devices to share the same medium. To achieve that, before transmitting a frame the medium is sensed for a specific period of time. If the channel is idle the frame is transmitted, otherwise an exponential back-off mechanism is triggered. In our implementation, a processing block continuously measures the RSSI in order to be able to identify if the medium is idle or not. The rest of the CSMA mechanism is under research and development currently. For more information refer to the Future Work section (Section 7.1).

The second MAC operation that is implemented in the PHY software is the acknowledgement of specific frames. Some frame types depending their destination MAC address, should be acknowledged by the receiver. When a receiver successfully decodes a frame, checks its type and the destination MAC address. If the frame type indicates that the frame should be acknowledged, the destination MAC address is checked. If the MAC address in not a broadcast address and is the same with the receiver's MAC address, then immediately issued an ACK frame transmission utilizing the transmit flowgraph.

The re-transmission mechanism is an operation that is responsible for transmitting again those frames whose ACKs failed to received in a certain amount of time. The functionality is implemented in the last processing block of the PHY transmit chain. When this block transmits a frame that should be acknowledged it blocks waiting on an asynchronous blocking message queue. The receive chain is responsible to enqueue a message at this queue whenever an ACK is received that has the proper destination MAC address, unblocking with this way the receiver. But the transmitter should also be able to perform a re-transmission if time elapsed during waiting for the ACK. For this reason, a time limited version of the blocking dequeue operation was added in the asynchronous message queue functionality, taking advantage of the conditional variables of Boost [52] library as described in Algorithm 6. If the dequeue returned null, the transmitted should initiate a re-transmission. The procedure is repeated until an ACK is received or a limit of re-transmissions is reached, where the frame is dropped. In order to minimize the re-transmission delay and the CPU utilization, each frame is stored in a pre-allocated buffer, based on the fact that its maximum possible size is known a priori. Thus, when a re-

transmission is occurred, the buffered samples that the last processing block had stored, are sent directly to the SDR device.

```
Function dequeue_timed(time_usec)
    scoped_lock(mutex) ;
    while queue.is_empty () do
        /* not_empty is a condition variable                    */
        if !not_empty.timed_wait (mutex, time_usec) then
            return NULL ;
        end
    end
    return queue.dequeue () ;
end
```

Algorithm 6: A time limited dequeue operation

Chapter 5

Software Optimization Techniques

Real-time software processing is a fundamental property of an SDR system. However, many telecommunication systems require a significant amount of processing resources and thus their real-time execution is not guaranteed. Towards this goal, developers should take full advantage of existing technologies, exploit parallelism and alter those signal processing algorithms that were intended to run on hardware to meet the software processing needs. Furthermore, there are many situations where the problem requires to be restated, taking into consideration the software implementation capabilities and drawbacks.

In this chapter several performance issues are discussed among with optimization techniques that we developed in order our implementation to meet the real-time requirements of the IEEE 802.11 standard.

5.1 Vectorized programming

Modern CPUs provide instructions that are able to process multiple data simultaneously with a single instruction. These instructions are known as Single Instruction, Multiple Data (SIMD). Figure 5.1 illustrates the difference between the conventional way of operations against the SIMD approach. Using scalar operations (Figure 5.1a), requires eight add operations to be executed sequentially, whereas the SIMD approach (Figure 5.1b) uses only one instruction in order to produce the same result. The benefits of such programming model in both terms of throughput and latency are obvious.

Depending the CPU manufacturer there are vectorized instructions for a large variety of operations including integer or floating point addition-subtraction, multiplication, division and other mathematical operations. In addition, many helper instructions are provided for manipulating the vectorized data types. Signal processing, audio-video or 3-D graphics applications can drastically increase their performance by using vectorized instructions.

SIMD instructions find use also in conditional branches avoidance. In SDRs, decisions based on conditions are very often. Depending the sampling rate of the telecommunication protocol (e.g. 20 MSPS for the IEEE 802.11) conditional branches may be executed million times per second. SIMD instructions can eliminate conditional branches by computing both results for the true and the false evaluation of a condition. Then, there are instructions that can load from two SIMD registers the appropriate values based on a bit

pattern. The bit pattern can be produced by applying the desired SIMD logical operation. An example of such a case is presented at Figure 5.2.

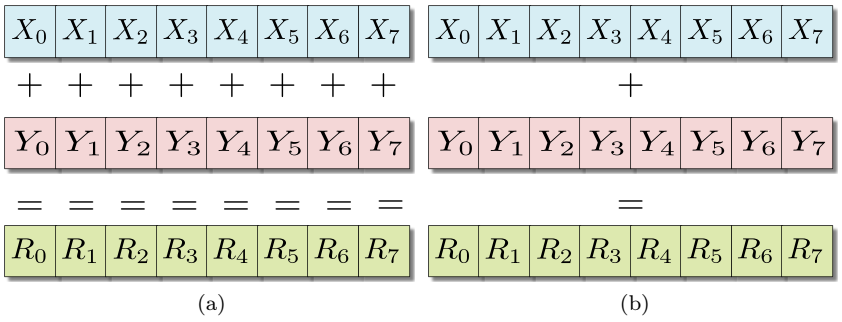


Figure 5.1: (a) Scalar operation (b) SIMD operation

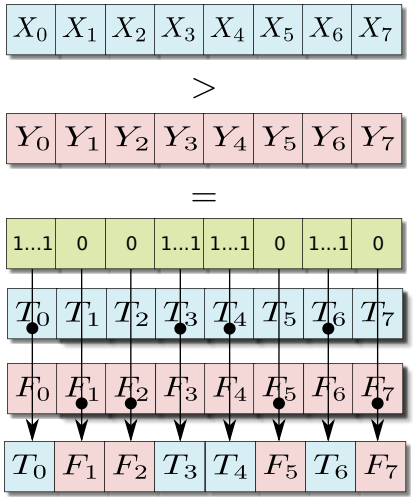


Figure 5.2: Conditional branch elimination using SIMD instructions

Although vectorized programming has a significant performance boost especially for real-time applications, their main drawback is that they provide a predefined set of instructions that do not cover all the possible application needs. Thus, in order to incorporate SIMD instructions and take advantage of their performance, developers should be able to restate their problem. This means often that the algorithm should be re-written from ground-up again. Some of such cases, are presented below. Our solutions using the Advanced Vector Extensions (AVX) instructions set, which support 256-bit registers, but the code can be easily ported into other SIMD intructions sets, as we are not using assembly code rather than compiler intrinsics.

5.1.1 Complex multiplication

Complex multiplication is major mathematical operation of the IEEE 802.11 software implementation. Synchronization, CFO correction and phase tracking are using extensively

complex multiplication. Even worst, these processing blocks handle million samples per second, so the need of high throughput SIMD enabled complex multiplication is imperative.

However, there is not an available SIMD instruction that can perform complex multiplication directly. Based on the definition of complex multiplication described in Equation 3.24, seems that with some permutations the partial multiplications can be computed easily using SIMD instructions. The main problem is the subtraction that is needed in the real part, whereas the imaginary part needs an addition. Hopefully, AVX provide the *vaddsubps(a, b)* instruction which subtracts and adds alternatively the floating point number *b* from *a*. The exact procedure is presented at Listing 5.1.

```
1  static inline void
2  complex_mul_avx(complex *out, const complex *x, const complex *y,
3                  const size_t len)
4  {
5    size_t     i;
6    __m256     x_simd;
7    __m256     y_simd;
8    __m256     tmp1, tmp2;
9    __m256     first_mul, second_mul;
10   __m256     result;
11
12   for(i = 0; i < len; i += (256 / 8) / sizeof(complex)) {
13     /*
14      * Load input data into AVX registers.
15      * Each complex number is represented as two interleaved floats,
16      * with the first float being the real and the second
17      * the imaginary part.
18      */
19     x_simd = _mm256_load_ps((float *) &x[i]);
20     y_simd = _mm256_load_ps((float *) &y[i]);
21
22     /* tmp1 = y[i], y[i], y[i + 2], y[i + 2], ..., y[i + 6], y[i + 6] */
23     tmp1 = _mm256_moveldup_ps(y_simd);
24
25     /* tmp2 = y[i+1], y[i+1], y[i+3], y[i+3], ..., y[i+7], y[i+7] */
26     tmp2 = _mm256_movehdup_ps(y_simd);
27
28     /* first_mul = x[0] * y[0], x[i+1] * y[0], ... , x[i+6] * y[i+6], x[i
           +7] * y[i+6] */
29     first_mul = _mm256_mul_ps(x_simd, tmp1);
30
31     /*  x_simd = x[i+1], x[i], x[i+3], x[i+2] , ... , x[i+7] , x[i+6] */
32     x_simd = _mm256_shuffle_ps(x_simd, x_simd, 0b10110001);
33
34     /* second_mul = x[i+1] * y[i+1] , x[i] * y[i+1], . . . , x[i+7] * y[i
           +7] , x[i+6] * y[i+7] */
35     second_mul = _mm256_mul_ps(x_simd, tmp2);
36
37     /*  result = x[i] * y[i] - x[i+1] * y[i], x[i+1] * y[i] + x[i] * y[i
           +1], ...*/
38     result = _mm256_addsub_ps(first_mul, second_mul);
39
40     _mm256_store_ps((float *) &out[i], result);
41   }
42 }
```

Listing 5.1: Complex multiplication using AVX intrinsics

To benchmark the benefits of SIMD enabled complex multiplication, two vectors with 102301 complex numbers each where used, performing 1000 iterations. Based on the execution times that are summarized at Table 5.1, the performance improvement is more than notable.

5.1.2 Type casting

The majority of the SDR hardware devices accommodate DACs or ADCs with up to 16-bit resolution. For this reason, each sample is represented with signed short integer data type.

Table 5.1: Performance comparison of scalar and SIMD instruction on an i7 2600K CPU

Operation	Scalar (seconds)	SIMD (seconds)	Speedup
Complex multiplication	1.8	0.14 (AVX)	12.85
Interleaved shorts to complex	7.02	0.42 (SSE4.1)	16.71
Complex to shorts	5.38	0.48 (SSE2)	11.2

However, often the signal processing is performed on floating point data types, avoiding overflows and accuracy issues. Therefore, samples should be cast to the appropriate data type depending if they are received from the device or are ought to be transmitted.

Type casting is not the only issue. Samples should be scaled also accordingly. For high bandwidth protocols, like IEEE 802.11a, casting and scaling may consume a notable amount of CPU resources. Consequently, our implementation exploits SIMD instructions to improve the processing throughput. Table 5.1 presents that using SIMD for type casting and scaling can lead to speedups above $10x$.

5.1.3 SIMD enabled FIR Filtering

FIR filters are extensively used in signal processing. These type of filters find many applications including auto or cross-correlation computations, moving sums, low-pass filtering and others. As described in previous sections, OFDM synchronization uses several FIR filters. The main drawback of these type of filters, is the high CPU resources that require. The computational overhead drastically increases as the filter order becomes bigger. The situation is even worst, when the input stream and filter coefficients are complex numbers, due to the additional operations required by the complex multiplication process.

```
1  void
2  fir_filter(float *out, float *in, float *coeffs,
3             size_t in_available, size_t filter_order)
4  {
5    for(i = 0; i < in_available; i++){
6      sum = 0;
7      for(j = 0; j < filter_order + 1; j++){
8        sum += in[i + j] * coeffs[j];
9      }
10     out[i] = sum;
11   }
12 }
```

Listing 5.2: Naive implementation of FIR filter

As Listing 5.2 presents, a simple software implementation of an FIR filter consists from two loops. For each available input item N multiplications and N additions are performed, where N is the order of the filter or in other words, the $N + 1$ filter coefficients.

A known method that is commonly used to reduce the computational resources required, is the FFT-based FIR filtering using overlap-add method [53]. This method performs an FFT followed by an IFFT of the input signal to achieve the desired result. Using the highly optimized FFTW implementation which has a complexity of $O(n \log n)$ for every size of n, significant performance gains can be achieved. However, the speedup can be achieved for those filter orders that are above a certain threshold. Depending the hardware this threshold may vary. For filter orders below this threshold the naive implementation

is faster than the FFT-based method. The throughput of each method is illustrated in Figure 5.3, where is shown clearly the threshold where the FFT-based method becomes faster.

To increase even more the throughput of the naive FIR implementation, SIMD instructions can be applied. With a first look many will argue that either the inner either the outer loop could be relatively easily vectorized using SIMD instructions. Although this would lead to a greater throughput, significant performance improvement can be achieved by vectorizing both the inner and the outer loop simultaneously, as suggested in [54]. Listing 5.3 reveals the concept of vectorizing inner and outer loops of the filter.

```
1  void
2  vectorized_fir_filter(float *out, float *in, float *coeffs,
3                        size_t in_available, size_t filter_order)
4  {
5    for(i = 0; i < in_available; i += 4){
6      for(j = 0; j < filter_order; j += 4){
7        out[i] += in[i + j]*coeffs[j] + in[i + j + 1]*coeffs[j + 1]
8                + in[i + j + 2]*coeffs[j + 2] + in[i + j + 3]*coeffs[j + 3];
9
10       out[i + 1] += in[i + j + 1]*coeffs[j] + in[i + j + 2]*coeffs[j + 1]
11                   + in[i + j + 3]*coeffs[j + 2] + in[i + j + 4]*coeffs[j +
                         3];
12
13       out[i + 2] += in[i + j + 2]*coeffs[j] + in[i + j + 3]*coeffs[j + 1]
14                   + in[i + j + 4]*coeffs[j + 2] + in[i + j + 5]*coeffs[j +
                         3];
15
16       out[i + 3] += in[i + j + 3]*coeffs[j] + in[i + j + 4]*coeffs[j + 1]
17                   + in[i + j + 5]*coeffs[j + 2] + in[i + j + 6]*coeffs[j +
                         3];
18     }
19   }
20 }
```

Listing 5.3: Inner and outer vectorized loops concept of an FIR filter

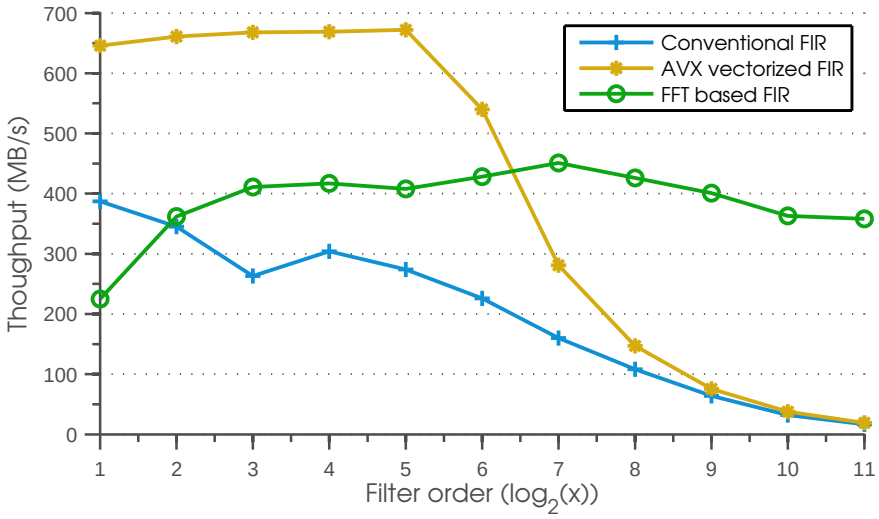


Figure 5.3: Throughput comparison of different FIR filter types

Although the concept of a inner and outer loop vectorized FIR filter seems to be relatively easy to implement there are some aspects that require attention. First of all,

a filter may have an order that is not entirely fit in an available SIMD register. For this reason an appropriate left zero padding should be applied at the filter coefficients as zero coefficients do not affect the result. Next, aligned and unaligned memory accesses should be spotted. Aligned is a memory access when the it begins from an index of the input or output vector that is integer multiple of the SIMD register. Otherwise it is considered unaligned. Aligned accesses are faster, so when possible should be preferred. As Listing 5.3 shows, the filter coefficients can be loaded into an AVX register using aligned access. The same applies on the store of the result at the output vector. However, the memory access on the input items vector can not be aligned. Moreover, there is a horizontal sum that should be performed. Such operation is not available in the AVX instruction set, so it should accomplished efficiently utilizing the provided instructions. The exact procedure is presented in Listing 5.4. The throughput comparison for different filter orders can be found in Figure 5.3. It is obvious that the vectorized FIR filter outperforms the FFT-based for filter orders less than 64. For greater number of coefficients, the $O(n \log n)$ complexity of the FFT-based filter leads to better throughput values. On the other hand, as was mentioned in the Synchronization section, the order of the filters used by our implementation is relative small. Using the AVX enabled FIR, a speedup of 100 up to 200 MB/s can be achieved.

```
1  void
2  vectorized_fir_filter_avx(float *out, float *in, float *coeffs,
3                            size_t in_available, size_t filter_order)
4  {
5    size_t    i,j;
6
7    float   *padded_coeffs;
8
9    __m256    tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, result;
10   __m256    coeffs_simd;
11   __m256    acc0, acc1, acc2, acc3, acc4, acc5, acc6, acc7;
12
13   __m128    acc_extract0, acc_extract1;
14
15   padded_coeffs = pad_left_zeros(coeffs, filter_order);
16
17   for(i = 0; i < in_available; i += 8){
18     /* Reset the accumulators */
19     acc0 = acc1 = acc2 = acc3 =
20     acc4 = acc5 = acc6 = acc7 = _mm256_setzero_ps();
21
22     for(j = 0; j < filter_order; j += 8){
23
24       /* First load the taps coefficients */
25       coeffs_simd = _mm256_load_ps(&padded_coeffs[j]);
26
27       tmp0 = _mm256_loadu_ps(&in[i + j ]);
28       tmp1 = _mm256_loadu_ps(&in[i + j + 1]);
29       tmp2 = _mm256_loadu_ps(&in[i + j + 2]);
30       tmp3 = _mm256_loadu_ps(&in[i + j + 3]);
31       tmp4 = _mm256_loadu_ps(&in[i + j + 4]);
32       tmp5 = _mm256_loadu_ps(&in[i + j + 5]);
33       tmp6 = _mm256_loadu_ps(&in[i + j + 6]);
34       tmp7 = _mm256_loadu_ps(&in[i + j + 7]);
35
36       /* Perform the multiplications and the accumulations */
37       tmp0 = _mm256_mul_ps(coeffs_simd, tmp0);
38       acc0 = _mm256_add_ps(acc0, tmp0);
39
40       tmp1 = _mm256_mul_ps(coeffs_simd, tmp1);
41       acc1 = _mm256_add_ps(acc1, tmp1);
42
43       tmp2 = _mm256_mul_ps(coeffs_simd, tmp2);
44       acc2 = _mm256_add_ps(acc2, tmp2);
45
46       tmp3 = _mm256_mul_ps(coeffs_simd, tmp3);
47       acc3 = _mm256_add_ps(acc3, tmp3);
48
49       tmp4 = _mm256_mul_ps(coeffs_simd, tmp4);
50       acc4 = _mm256_add_ps(acc4, tmp4);
```

```
51
52          tmp5 = _mm256_mul_ps(coeffs_simd, tmp5);
53          acc5 = _mm256_add_ps(acc5, tmp5);
54
55          tmp6 = _mm256_mul_ps(coeffs_simd, tmp6);
56          acc6 = _mm256_add_ps(acc6, tmp6);
57
58          tmp7 = _mm256_mul_ps(coeffs_simd, tmp7);
59          acc7 = _mm256_add_ps(acc7, tmp7);
60        }
61
62        /* Horizontally sum the AVX registers */
63        tmp0 = _mm256_hadd_ps(acc0, acc1);
64        tmp1 = _mm256_hadd_ps(acc2, acc3);
65        tmp2 = _mm256_hadd_ps(acc4, acc5);
66        tmp3 = _mm256_hadd_ps(acc6, acc7);
67
68        tmp0 = _mm256_hadd_ps(tmp0, tmp1);
69        tmp2 = _mm256_hadd_ps(tmp2, tmp3);
70
71        /* Break the 256 register in two parts of 128 bits */
72        acc_extract0 = _mm256_extractf128_ps(tmp0, 0);
73        acc_extract1 = _mm256_extractf128_ps(tmp0, 1);
74
75        acc_extract0 = _mm_add_ps(acc_extract0, acc_extract1);
76        result = _mm256_insertf128_ps(result, acc_extract0, 0);
77
78        acc_extract0 = _mm256_extractf128_ps(tmp2, 0);
79        acc_extract1 = _mm256_extractf128_ps(tmp2, 1);
80
81        acc_extract0 = _mm_add_ps(acc_extract0, acc_extract1);
82        result = _mm256_insertf128_ps(result, acc_extract0, 1);
83
84        _mm256_store_ps(&out[i], result);
85      }
86    }
```

Listing 5.4: Inner and outer vectorized loops implementation of FIR filter using AVX instructions

5.1.4 SIMD enabled Viterbi decoder

It is known that a software implementation of the Viterbi decoder, can greatly benefit from SIMD instructions [55]. In order our implementation to compete with the strict time constrains of the standard, a vectorized version of the algorithm had to be implemented.

Firstly, in order to get full advantage of the SSE instructions, we used unsigned short integers for the data representation. Therefore, each 128-bit SSE register can accommodate up to 8 unsigned short values.

Initially, the hamming distance, which is the most common operation during the Viterbi butterflies computation was vectorized. The hamming distance is computed by XORing two values and counting the number of 1 bits. This can be performed by computing the XOR of two SSE registers and extracting the unsigned short values. After that, the techniques that are described in Section 5.2.3 can be applied to retrieve the number of 1 bits. But this requires the data to be type cast from 128-bit SSE register to unsigned integers and back again to a 128-bit register, eliminating the performance gains. However, there is a more simple and optimum solution. It is known that the Viterbi handles incoming data in a per 2-bit basis. Moreover, the trellis output is 2 bits, so the hamming distance is computed having as input two 2-bit entities. In this case, bit count can be computed entirely on SSE using bitwise XOR, AND, shifting and addition instructions based on Algorithm 7.

At each Viterbi butterfly, the algorithm selects that transition that minimizes the accumulated error. If scalar data types are used, this is accomplished using conditional branches. With SIMD instructions these conditional branches can be eliminated. Furthermore, the minimum path comparison can now performed simultaneously for 8 distinct

Function *hamming_ dist(received, trellis_ out)*
    xored = received ⊕ trellis_out ;
    acc = xored & 0b01 ;
    acc += xorred ≫ 1 ;
    return *acc* ;
end

Algorithm 7: Hamming distance for 2-bit entities

trellis states, instead of one. This greatly increases the throughput of the algorithm. Figure 5.4 graphically presents how these operations can be accomplished.

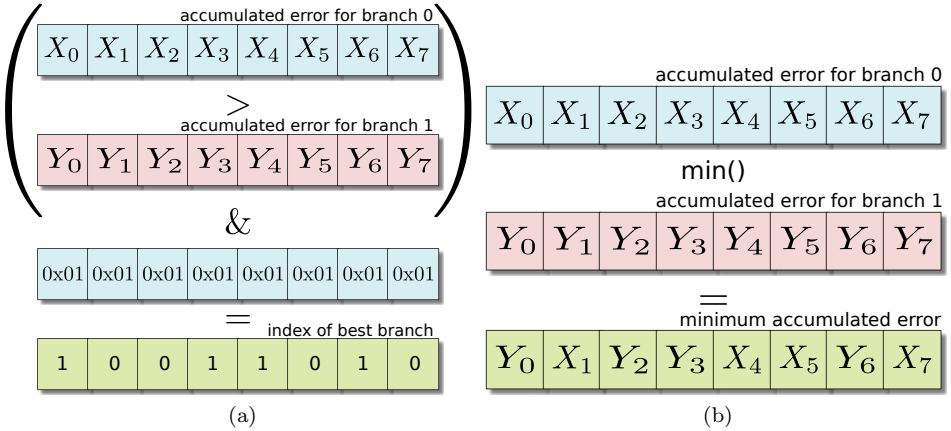


Figure 5.4: Using SIMD for (a) minimum branch selection, and (b) calculating the minimum accumulated error for 8 different Viterbi states concurrently

5.2 Bit manipulation

Bitwise operations is a very common characteristic of telecommunication systems. Although in hardware are very easy and fast, this not true for software. Many will argue that bitwise operations are fast in software too. Unfortunately, this is not completely true. Bitwise operations executing on software are fast when they performed in a per byte-multiple basis. But there are many occasions, where a per bit handling is necessary. To achieve this, many additional bit operations are required, reducing the overall performance of the system. An example of such a case is the bits reversal of a byte. While in hardware this would require just to invert the sequence of the wires, in software requires too many bitwise operations. However, there are techniques that can avoid unnecessary operations. In the following subsections, some techniques that our implementation uses are described.

5.2.1 Modulo by power of 2

Modulo operations are very common in bit manipulation. Furthermore, almost all the major processing stages of telecommunication systems that operate on a bit stream extensively use the modulo operation. Likewise division, modulo operations require more CPU cycles to execute, about 4 times more than addition or multiplication. However, there are

Table 5.2: Performance comparison of various bitwise operations on an i7 2600K CPU

Operation	Naive version	Optimized version	Speedup
Modulo with power of 2	3683 ms	527 ms	6.98
Bit Reversal 32-bit	8169 ms	2027 ms	4.03
Bit Reversal 64-bit	8169 ms	2103 ms	3.88
Bit Counting	6997 ms	526 ms	13.3

cases that the same result can be achieved without explicitly use the division or modulo operation. If the denominator is a power of two, division can easily performed with bit shift operations. Similarly for the modulo case, the desired result is obtained using one bitwise operation and a subtraction as Equation 5.1 presents.

$$x \bmod y = x \ \& \ (y-1) \tag{5.1}$$

Although modern compilers will take care of such cases and will produce optimized code, they fail to do so when the denominator is not a constant rather a parameter. Often the denominator is a power of 2 but not constant during the program execution. Such cases should be explicitly handled by developers using the optimized versions of each operation. The performance benefits of such optimization can be found at Table 5.2.

5.2.2 Reverse the bits in a byte

Most telecommunication standards has as convention to transmit over the medium the LSB first. For hardware implementations this is not an issue. On the other hand, this is an annoying issue for software solutions that process the data per byte. Bits of every byte should be flipped, wasting vital resources. The obvious solution is to perform some bit shift and bitwise AND, OR operations. In [56] there is a very useful collection with several bitwise hacks that can boost the performance of several well known bit operations.

Depending the host architecture our implementation uses the techniques provided in Listing 5.5. For 64-bit systems bit reversing requires 3 operations, whereas for 32-bit 7 operations greatly reducing the amount of operations required with a conventional algorithm. Performance results are described at Table 5.2.

5.2.3 Bit Counting

Counting the active bits is another very common operation that is heavily used in parity creation and check, convolutional encoding and at the Viterbi decoder. Especially at the case of Viterbi decoder, for every two bits of the input bit stream, the number of bits that are 1's should be calculated. For this reason, the need for an efficient bit counting algorithm is more that imperative.

Although the bit counting could be easily be implemented using lookup tables, many modern CPUs provide a hardware instruction known as *popcnt*. With this instruction the bit count overhead can be significantly reduced. Table 5.2 reveals the difference between the naive version of bit count and the *popcnt* instruction.

```
#ifdef __x86_64__
  reversed_byte = (byte * 0x0202020202ULL & 0x010884422010ULL)
          % 1023;
#else
  reversed_byte = ((byte * 0x0802LU & 0x22110LU)
            | (byte * 0x8020LU & 0x88440LU))
          * 0x10101LU >> 16;
#endif
```

Listing 5.5: Efficient bit reversing for 64 and 32-bit architectures

Chapter 6

Evaluation

In this section the performance evaluation of the system is discussed. Firstly, the evaluation of the system by the physical layer properties point of view is presented. More over a performance analysis in terms of the achievable latency and throughput in real case scenarios is performed. Last but not least, the CPU utilization and the computational requirements of each processing block are discussed.

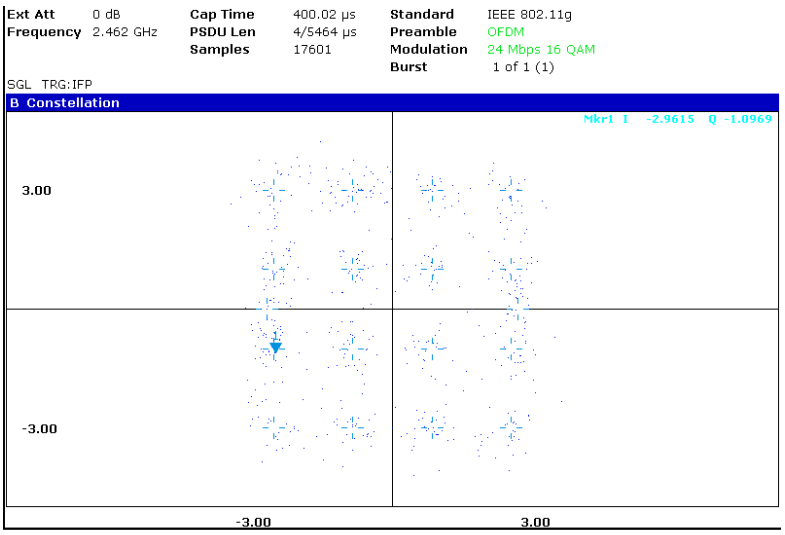
The experiments are performed using the *Noctar* PCI-e card at both sides. The necessary horsepower is provided by two different host computers. The first one accommodates a 4-core i7 2600K CPU operating at 3.4 GHz. The other one, hosts two 6-core Xeon 2630 v2 CPUs with operating frequency of 2.6 GHz. The decision on the hardware characteristics was made to investigate the performance behavior depending the number of available cores and operating frequency.

6.1 Physical Layer Performance

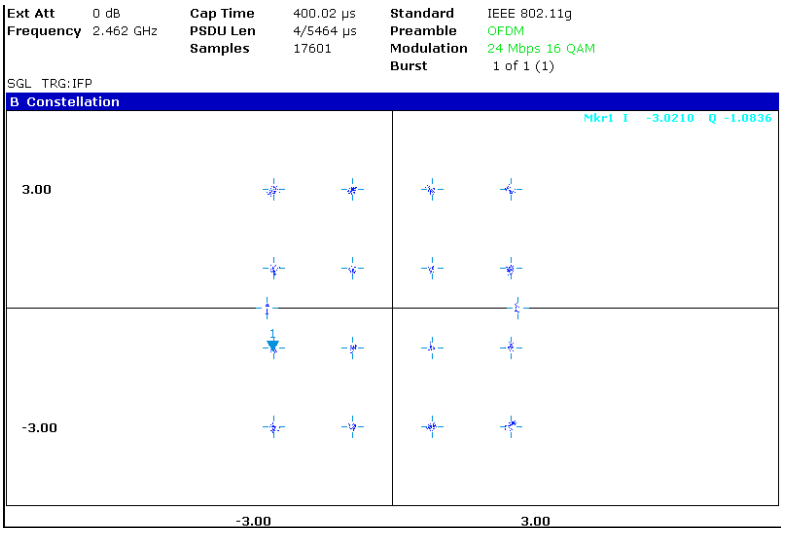
In this work the *Noctar* PCI-e SDR card was used, due to the ability to keep a constant communication data rate with the host PC and the low latency of the PCI-e interconnect. Unfortunately, the analog part of this card has a severe flaw that it was unknown to the manufacturer itself and discovered during this work.

The In-phase and the Quadrature part of the transmit chain are subjected to a great phase imbalance. The effects of this kind of imbalance greatly reduce the performance of multi-carrier systems like OFDM [57]. The difference between the problematic transmission of Noctar card and the USRP B210, which do not suffer from I/Q imbalance can been seen in Figure 6.1. The screen-shots retrieved using the R&S® FSVR40 real-time spectrum analyzer. For this reason, some measurements (e.g. RX 64-QAM) were impossible to be performed.

Despite the I/Q imbalance our decoder can handles modulations up to 16-QAM without any significant performance degradation. The performance of the system can be derived from the number of re-transmissions that occur. Figure 6.5 reveals that the number of re-transmissions is on par regardless the modulation. How the information of re-transmissions can be retrieved from the Cumulative Distribution Function (CDF) plots is explained at the next section. The Error Vector Magnitude (EVM) of the received constellation points among with other statistics for the 24 Mbps 16-QAM modulation, can be retrieved from Figure 6.2. Even with an EVM of -14 dB, our system still is able to decode the frames without any problem, despite the threshold of the standard that is defined at -16 db for



(a)



(b)

Figure 6.1: 16-QAM constellation using (a) Noctar card (b) USRP B210

this particular modulation.

A consequence of the above result is the stability of our system in low SNR enviroments. Studies [58] have shown that there is a strong relationship between EVM and SNR. More precisely for constellations like those used by the standard, the SNR is the inverse of the EVM observation. As a result, our system can operate in low SNR environments, consequence of the two step time and frequency synchronization and the two stage phase tracking mechanism. The extra computational resources that were supplied to the system at the synchronization phase, have not been in vain rather their contribution is vital for the overall system performance.

Ext Att	0 dB	**Cap Time**	400.02 µs	**Standard**	IEEE 802.11g
Frequency	2.462 GHz	**PSDU Len**	4/5464 µs	**Preamble**	OFDM
		Samples	17601	**Modulation**	24 Mbps 16 QAM
				Burst	1 of 1 (1)

SGL TRG:IFP

Result Summary

Bursts: 1 of 1	**Min**	**Mean**	**Limit**	**Max**	**Limit**	**Unit**
EVM All Carr.	**19.60**	**19.60**	15.85	**19.60**	15.85	%
	- 14.16	**- 14.16**	- 16.00	**- 14.16**	- 16.00	dB
EVM Data Carr.	**19.12**	**19.12**	15.85	**19.12**	15.85	%
	- 14.37	**- 14.37**	- 16.00	**- 14.37**	- 16.00	dB
EVM Pilot Carr.	**24.60**	**24.60**	39.81	**24.60**	39.81	%
	- 12.18	**- 12.18**	- 8.00	**- 12.18**	- 8.00	dB
IQ Offset	**- 20.41**	**- 20.41**	- 15.00	**- 20.41**	- 15.00	dB
Gain Imbalance	**- 0.26**	**- 0.26**		**- 0.26**		%
	- 0.02	**- 0.02**		**- 0.02**		dB
Quadrature Err	**- 5.64**	**- 5.64**		**- 5.64**		°
Freq. Err	**- 10076.37**	**- 10076.37**	± 61550	**- 10076.37**	± 61550	Hz
Symb Clock Err	**25.49**	**25.49**	± 25	**25.49**	± 25	ppm
Burst Power	**- 18.12**	**- 18.12**		**- 18.12**		dBm
Crest Factor	**8.83**	**8.83**		**8.83**		dB

Figure 6.2: Noctar PHY statistics at 2.462 GHz for 24 Mbps 16-QAM modulation

6.2 Latency and Throughput Measurements

Latency plays a key role at the responsiveness of a system. A software implementation has several components that can introduce latency. For this reason latency measurements are performed with two different approaches. Firstly, for the case of transmission, the delay of processing a packet by the time it is received from the MAC layer until it is delivered to the hardware is measured. For the case of reception as the processing delay is considered, the delay from the frame arrival indication until it is propagated to the MAC layer. To avoid possible re-transmissions that would affect the measurements, broadcast UDP packets where used, due to the fact that these type of messages do not require re-transmissions. The processing delay during TX and RX for different packet sizes is presented in Table 6.1 and 6.2 respectively. The resulting values is an average of over 5000 frames.

Table 6.1: TX processing delay for different packet sizes and modulation schemes

Modulation	**64 bytes**	**128 bytes**	**256 bytes**	**512 bytes**	**1024 bytes**
i7 2600K @ 3.4 GHz					
BPSK	98 μs	124 μs	126 μs	179 μs	275 μs
QPSK	84 μs	94 μs	134 μs	183 μs	269 μs
16-QAM	77 μs	93 μs	117 μs	179 μs	275 μs
64-QAM	66 μs	82 μs	95 μs	139 μs	219 μs
2 × Xeon 2630 v2 @ 2.6 GHz					
BPSK	200 μs	252 μs	280 μs	291 μs	413 μs
QPSK	186 μs	200 μs	266 μs	323 μs	375 μs
16-QAM	160 μs	188 μs	233 μs	309 μs	377 μs
64-QAM	154 μs	162 μs	203 μs	228 μs	328 μs

Table 6.2: RX processing delay for different packet sizes and modulation schemes

Modulation	**64 bytes**	**128 bytes**	**256 bytes**	**512 bytes**	**1024 bytes**
i7 2600K @ 3.4 GHz					
BPSK	300 μs	386 μs	570 μs	916 μs	1612 μs
QPSK	244 μs	312 μs	399 μs	598 μs	978 μs
16-QAM	208 μs	264 μs	360 μs	476 μs	696 μs
2 $\times$ Xeon 2630 v2 @ 2.6 GHz					
BPSK	513 μs	585 μs	784 μs	1110 μs	1698 μs
QPSK	471 μs	547 μs	688 μs	818 μs	1282 μs
16-QAM	459 μs	521 μs	618 μs	822 μs	1153 μs

The result clearly show the vast difference between the transmission and reception processing requirements. The results are consistent with the CPU utilization measurements that are presented in the following section.

On the other hand, even if these delays provide information about the correlation between the packet size and latency, an application using the SDR system may subjected to different delay values due to the hardware buffering or/and the re-transmissions that may occur. For this reason, latency for different packet sizes is measured from the application point of view, with the re-transmission mechanism enabled. The tool that is used to retrieve the measurements is the well known *ping*. The minimum, average and maximum Round Trip Time (RTT) timings of 5000 packets for different modulation schemes and packets sizes are summarized in Table 6.3.

Table 6.3: RTT in milliseconds for different packet sizes and modulation schemes

Modulation		**64 bytes**	**128 bytes**	**256 bytes**	**512 bytes**	**1024 bytes**
BPSK	min	2.955	3.184	3.520	4.298	5.621
	avg	17.218	15.519	18.686	18.024	21.701
	max	1043	1035	1031	1039	1050
QPSK	min	2.99	3.192	3.7	4.33	5.64
	avg	16.202	15.895	15.248	19.251	20.51
	max	1023	1052	1034	1034	1023
16-QAM	min	2.9	3.231	3.662	4.28	5.329
	avg	17.328	14.273	19.66	15.513	22.678
	max	1021	1052	1038	1039	1024

The above table can not provide comprehensive information about the latency that an application will face. There are cases, where the delay of a packet is quite large, thus increasing the average, but these occasions are rare. These large delays occur due to the re-transmissions and the large ACK timeout threshold. For this reasons we provide the CDF of the RTTs in Figures 6.3 and 6.4. The CDF plots give great amount of information about the system performance. Regardless the modulation scheme, for packets up to 256 bytes the 80% of the frames have RTT less than 5 ms. Moreover, the figures reveal that there is a lower bound of the best achievable RTT which is very close to 3 ms. An interesting observation is the 'steps' that each CDF curve has. These steps correspond to the number of re-transmissions occurred during the data exchange. The results prove that more than 80% of the ping-pong exchanged frames need less than, or 2 re-transmissions. This is quite impressive, recalling the constellation quality of the 16-QAM modulation

due to the phase I/Q imbalance. In addition, as expected, large frames are subjected to a higher re-transmission probability. The larger transmission-reception duration leads to a higher risk of a discontinuity on the stream or an error due to the poor constellation quality.

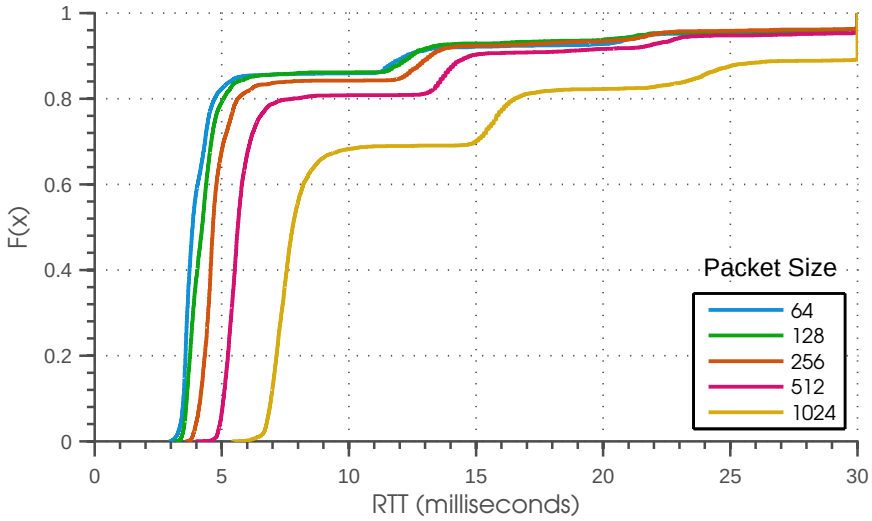


Figure 6.3: RTT CDF using BPSK modulation

Figure 6.5 provides a comparison of the RTT values among different modulations. For simplicity reasons only packet with sizes 64 and 1024 are presented, but the rest of the packet lengths are on par with these results. This figure uncover an interesting aspect of the software implementation. The system can not benefit the higher throughput of higher order constellations. It is clear that the latency achieved with BPSK is almost the same with that of QPSK and 16-QAM, comparing always the same number of re-transmissions. It is not yet clear the cause of this phenomenon, but a possible reason could be the Viterbi decoder, whose processing delay is increasing as the constellation order becomes higher, eliminating the speed gain due to the increased data rate. In addition, another possible explanation is the overhead of the software stack itself. Towards this belief, the findings of Figures 6.3 and 6.4 indicate that for small packet sizes from 64 to 256 bytes, the RTT measurements are quite similar. This suggests a constant overhead of the software stack, that is greater that the data transfer delay of small packet lengths.

Throughput is another vital metric that characterize the performance of a telecommunication system. While the delays introduced by the software implementation are completely different from the expected delays of a hardware solution, it is imperative to measure the throughput not only using UDP, which is the standard method, but also using TCP. In both cases, the well known *iperf* measurement tool is used.

6.2.1 UDP Throughput

The previous findings indicate that the system behaves differently depending on the host computer. For this reason the throughput measurements are performed in two steps, each with a duration of 2 minutes. The first, uses the one host computer as the receiver of the transmitted UDP frames, whereas at the next step the host becomes the transmitter. The

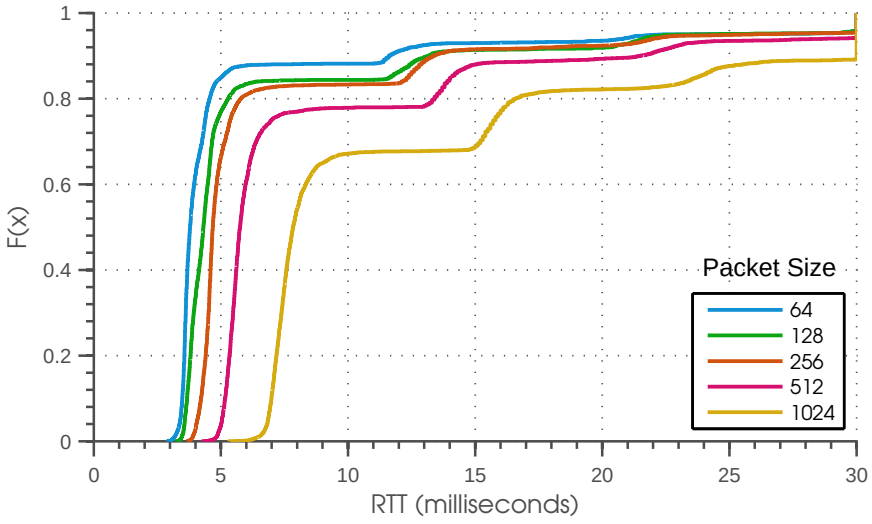


Figure 6.4: RTT CDF using 16-QAM modulation

results are depicted in Figure 6.6. For small packet sizes the performance of the system it terms of throughput is low. This is the expected behavior, recalling the minimum achievable latency of the system previously discussed. In addition, as the packet length increases, so does the throughput because the latency becomes smaller relative to the transmission delay.

The figure also reveals the performance differences between the two different hosts. The i7 2600K outperforms in terms of the achievable throughput the host with the two Xeon CPUs for all the different packet lengths, conforming with the processing delay findings.

Last but not least, Figure 6.7 indicates that the packet loss ratio is below 2.5% for all the scenarios, which is quite encouraging for a wireless system like this.

6.2.2 TCP Throughput

At this section we try to investigate how TCP performs using the SDR system. The adaptive rate control mechanism of TCP it is expected to be affected by the increased packet loss ratio that the SDR network introduces. Again the experiments are performed using both sides as the recipient of the TCP stream. With the default parameters and using the i7 2600K host as receiver, the mean throughput is 522 Kbits/s, whereas for the dual Xeon host the throughput is 378 Kbits/s. The lower TCP throughput is a result of the TCP acknowledgement mechanism. Despite the small size of the TCP ACK, the latency of the SDR system has a significant impact on the TCP performance.

In addition, even if the findings have the same trend with those of UDP, we analyzing further TCP behavior by altering the default values of the congestion window. For all the window configurations, the dual Xeon CPU has a lower throughput as Table 6.4 reveals. Another interesting finding is that both hosts perform better when the window can take values from 256 to 8192 bytes. This suggests that when the underline telecommunication protocol is SDR based, the parameters of the upper layers should be adjusted for better performance.

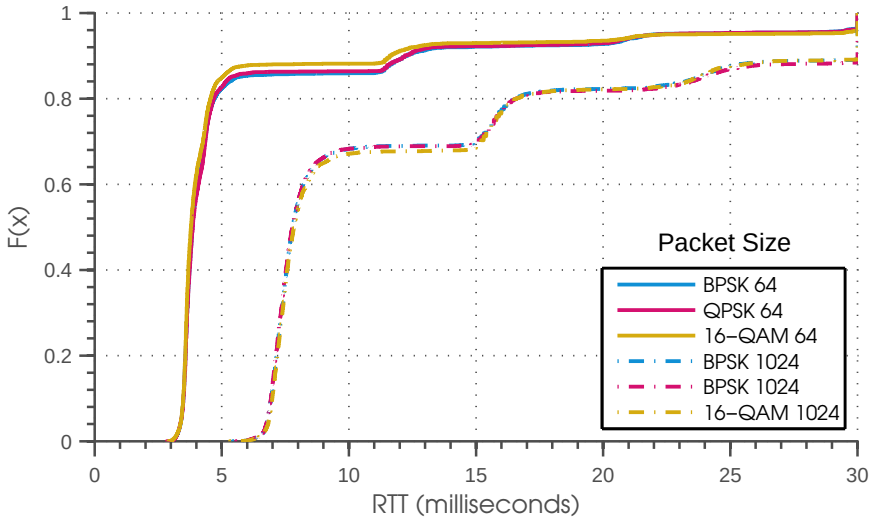


Figure 6.5: RTT CDF for different modulation schemes

Table 6.4: TCP throughput in Kbits/s for different window size configurations

CPU type	**TCP window configuration (min-default-max)**					
	64-128-1K	128-256-2K	256-512-4K	512-1K-8K	1K-2K-16K	default
i7 2600K	32.3	245	316	703	538	522
2 × Xeon 2630 v2	30.1	150	235	571	301	378

6.3 CPU Utilization

The computational resources consumption is key characteristic of all software systems. In this section the CPU utilization of the system is presented. A rough estimation of the percentage of the occupied CPU resources under different usage scenarios is presented in Table 6.5. This table indicates that the host computer has enough resources to execute other tasks regardless the load of the SDR wireless network.

Table 6.5: Userspace CPU utilization of the IEEE 802.11a software transceiver

CPU Type	**Idle**	**TCP Server**	**TCP Client**	**Full Duplex**
i7 2600K @ 3.4 GHz	44%	48%	45%	53%
2 × Xeon 2630 v2 @ 2.6 GHz	29%	33%	32%	36%

An obvious observation is the high CPU usage when the system is idle. Until a frame is arrived, the energy detection algorithm is constantly performing computations. As described previously, this is accomplished with an FIR filter, which is a quite computational demanding operation due to the high rate of incoming data. Even worse, the synchronization blocks that follows are forced to constantly perform calculations, even if it is not necessary, due to limitation of the GNU Radio stack discussed in Section 3.2. Moreover, during reception the receive chain could be stopped. This would require the flowgraph to change and re-designed. Even if GNU Radio supports this feature, its poor implementation

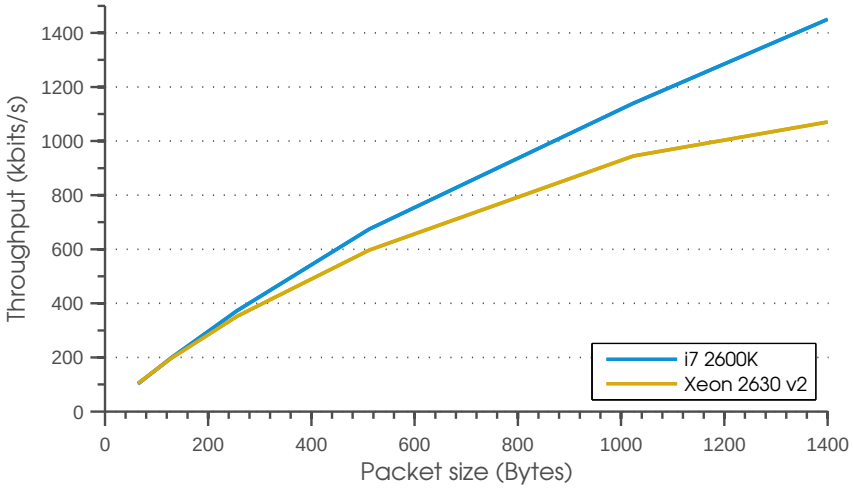


Figure 6.6: UDP throughput measurements

leads to large reconfiguration times, that are prohibitive for the IEEE 802.11a case.

Having performed a coarse CPU utilization approximation, a more fine grained analysis follows. Its goal is to present how the different components of the software implementation contribute to the overall CPU utilization. This is accomplished using the Oprofile [59] statistical profiling tool.

The computational resources that each component of the system utilizes under a fully loaded bidirectional data exchange is illustrated in Figure 6.8. As expected a great part of the total CPU utilization corresponds to the synchronization process, which is continuously performing calculations even if it not necessary. Moreover, he synchronization process includes three FIR filters. In order to emphasize the significance of the software optimization on this filters, their contribution to the total computational resources of the system is separately presented. The results indicate that the FIR filters correspond to the 34%, whereas the other synchronization components to the 19%. In total, synchronization process corresponds to more than 50% of the overall instructions executed by the SDR system. Furthermore, the overhead of the GNU Radio platform and thread synchronization is more than notable.

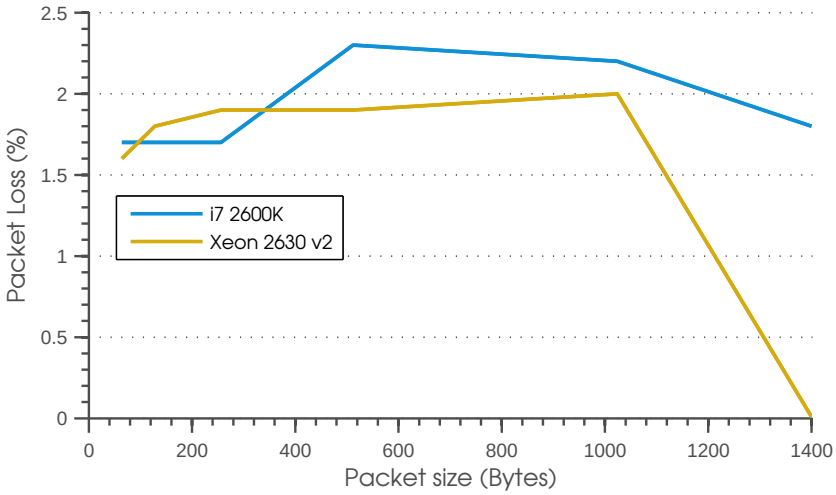


Figure 6.7: UDP packet loss ratio

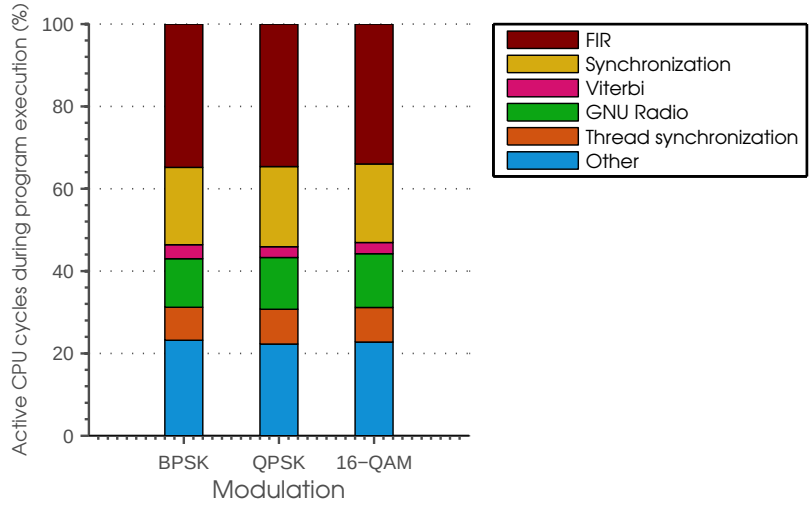


Figure 6.8: Software components' CPU utilization distribution during full duplex communication

Chapter 7

Conclusions

In this work we presented a software implementation of the IEEE 802.11a physical and MAC layer. Despite the limitations and difficulties our system provides a robust and unified system that can offer seamless SDR based connectivity. Moreover, our implementation provides such an abstraction level, so that applications totally ignore the underline communication protocol and continue to operate normally without any change. To our knowledge this is the second implementation after [8] that achieves this and the first one using an SDR software stack like GNU Radio.

The overall system performance is evaluated with real life applications. Even if the system can not achieve the data rates of the IEEE 802.11a standard, the throughput and latency achieved are quite promising. Furthermore, the analysis revealed that 80 % of the frames need less that 2 re-transmissions even with the severe flaw of the RF hardware. With the improvement of the RF hardware and some more optimizations the overall performance can be drastically increased.

Our implementation provides a system that is going beyond the scope of just a software implementation of IEEE 802.11a. As Cognitive Radios become more and more popular, numerous theoretical works are already established. Our system could be the basis for experimenting and evaluating the performance of these cognitive solutions. The level of abstraction provides not only great flexibility to the researchers but contributes to the validity of evaluating work itself too. By seamless using real life applications utilizing the wireless interface that our system provides, results are retrieved more easily and are more realistic.

Last but not least, this work revealed that software implementation of demanding and complex telecommunication systems are possible. The host computers with the running system, were able to perform other tasks too as the overall CPU utilization was below 50%, even under heavy bidirectional communication. Some of the limitations and drawbacks that do not allow software implementations to compete the hardware were presented. These aspects can be taken into consideration during the development phase of new telecommunication standards, allowing the SDR based solutions to be more competitive and robust, opening an enormous new era in the telecommunications field.

7.1 Future Work

Despite the robust and reliable system that this work presents, there are many aspects that can be improved in order to achieve smaller latency timings and grater throughput.

Some of them are already under development, whereas others under research.

The most severe problem of any GPP implementation is the quite strict timing constraints of the acknowledgment. According the standard, when a STA sends a frame an ACK should be received after Short Interframe Space (SIFS) time, otherwise a retransmission is triggered. IEEE 802.11a defines the SIFS to $16\mu s$, which is impossible for the software to deal with. In our system the ACK timeout threshold is increased an order of magnitude. This of course reduces the effective throughput of the system and make it incompatible with normal IEEE 802.11a devices. Towards lower ACK timeout timings, we already have under development an ACK caching mechanism. It is true that if an ACK have been sent to a STA there is high probability another ACK to be sent again at the same STA in the near future. Furthermore, the information of each ACK for the same STA remains the same and the number of the different STAs that exchange information, in short term is limited to few dozens. Based on these facts, a caching mechanism can greatly increase the overall performance of the system. When an ACK is about to be sent, the mechanism checks if an already cached ACK exists. If yes, the system immediately sends the already pre-calculated and stored samples, bypassing the encoding process. Consequently, the transceiver will delay to respond with an ACK only at the first time that communicates with a specific STA. This is not a problem as the re-transmission mechanism will handle the ACK timeout. The rest of the received frames will use the cached ACK. A similar approach was used at [8].

Another issue that is under research is the CSMA/Collision Avoidance (CA) mechanism. Before a frame transmission, the mechanism calculates the RSSI of the received samples. If it is above a threshold, the transmitter falls-back in a random back-off. The main problem of a software implementation of this mechanism is the buffering that occurs in the different processing blocks of the receiver. By the time the RSSI block indicates that the channel is idle, due to the fact that these samples were buffered, the channel may not be really idle. This in general is a quite difficult to cope with limitation of SDRs.

The most demanding process of our IEEE 802.11a software implementation is the frame detection and synchronization, which is performed continuously for every received sample. It is true that this process should be performed only until the start of the SIGNAL OFDM symbol. After that or during a frame transmission, there is no need for the synchronization process to continue executing. Ideally the corresponding blocks could be paused for the appropriate time and restarted later. This of course requires the ability from the software platform to be able to dynamically change the flowgraph. While GNU Radio stack includes such a feature, its implementation details revealed that the procedure is accomplished in a quite poor, in terms of performance, way. When a reconfigure is triggered, the scheduler stops all the active threads, redesign the whole flowgraph and execute the new one form the beginning. The reconfiguration process has a delay in the order of dozens milliseconds, which of course is unacceptable for the IEEE 802.11a standard. Towards faster reconfiguration timings, we tried to extend the GNU Radio capabilities by designing a smartest and more robust approach, such that only the depending from the reconfiguration processing blocks are affected without stopping all the active blocks. Despite the better reconfiguration timings, the process could not be accomplished in the order of few microseconds. The main reason is the internal structure and representation of the flowgraph blocks of GNU Radio.

The above problem and all others that emerged during this work gave us the belief, that a software stack should deal with them at the design process. So with the experience gained from the implementation of the IEEE 802.11a, we intend to design and implement a SDR stack that can efficient handle both stream- and bursty-oriented telecommunication protocols.

Bibliography

[1] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 28, no. 1-2, pp. 7–27, 2001.

[2] Wireshark. [Online]. Available: http://www.wireshark.org

[3] M. J. Meeuwsen, O. Sattari, and B. M. Baas, "A full-rate software implementation of an IEEE 802.11 a compliant digital baseband transmitter," in *Signal Processing Systems, 2004. SIPS 2004. IEEE Workshop on*. IEEE, 2004, pp. 124–129.

[4] Y. Tang, L. Qian, and Y. Wang, "Optimized software implementation of a full-rate IEEE 802.11 a compliant digital baseband transmitter on a digital signal processor," in *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE*, vol. 4. IEEE, 2005, pp. 5–pp.

[5] J. Garcia and R. Cumplido, "On the design of an FPGA-Based OFDM modulator for IEEE 802.11a," in *Electrical and Electronics Engineering, 2005 2nd International Conference on*. IEEE, 2005, pp. 114–117.

[6] A. T. Tran, D. N. Truong, and B. M. Baas, "A complete real-time 802.11 a baseband receiver implemented on an array of programmable processors," in *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*. IEEE, 2008, pp. 165–170.

[7] C. R. Berger, V. Arbatov, Y. Voronenko, F. Franchetti, and M. Puschel, "Real-time software implementation of an IEEE 802.11 a baseband receiver on Intel multicore," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*. IEEE, 2011, pp. 1693–1696.

[8] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, "Sora: high-performance software radio using general-purpose multi-core processors," *Communications of the ACM*, vol. 54, no. 1, pp. 99–107, 2011.

[9] G. Zacheo, D. Djukic, A. Dorni, F. Babich, and F. Ricciato, "A Software-Defined Radio Implementation of an 802.11 OFDM Physical Layer Transceiver," in *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on*. IEEE, 2012, pp. 1–4.

[10] B. Bloessl, M. Segata, C. Sommer, and F. Dressler, "An IEEE 802.11 a/g/p OFDM Receiver for GNU Radio," in *Proceedings of the second workshop on Software radio implementation forum*. ACM, 2013, pp. 9–16.

[11] F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, "Efficient utilization of SIMD extensions," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 409–425, 2005.

[12] J. Kim, S. Hyeon, and S. Choi, "Implementation of an SDR system using graphics processing unit," *Communications Magazine, IEEE*, vol. 48, no. 3, pp. 156–162, 2010.

[13] U. Ramacher *et al.*, "Software-defined radio prospects for multistandard mobile phones," *IEEE Computer*, vol. 40, no. 10, pp. 62–69, 2007.

[14] J. Mitola, "The software radio architecture," *Communications Magazine, IEEE*, vol. 33, no. 5, pp. 26–38, 1995.

[15] J. Mitola III, "Software radios: Survey, critical evaluation and future directions," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 8, no. 4, pp. 25–36, 1993.

[16] R. Lackey and D. W. Upmal, "Speakeasy: The military software radio," *Communications Magazine, IEEE*, vol. 33, no. 5, pp. 56–61, 1995.

[17] D. L. Tennenhouse and V. G. Bose, "The SpectrumWare approach to wireless signal processing," *Wireless Networks*, vol. 2, no. 1, pp. 1–12, 1996.

[18] R. Vidano, "SPEAKeasy II-an IPT approach to software programmable radio development," in *MILCOM 97 Proceedings*, vol. 3. IEEE, 1997, pp. 1212–1215.

[19] Field programmable RF ICs: LMS6002D. [Online]. Available: http://www.limemicro.com/products/LMS6002D

[20] AD9361: RF Agile Transceiver. [Online]. Available: http://www.analog.com/en/rfif-components/rfif-transceivers/ad9361/products/product.html

[21] GNU Radio. [Online]. Available: http://gnuradio.org

[22] Python.org. [Online]. Available: https://www.python.org

[23] SWIG software development tool. [Online]. Available: https://www.swig.org

[24] Ettus Research. [Online]. Available: http://home.ettus.com

[25] bladeRF - the USB 3.0 Superspeed Software Defined Radio. [Online]. Available: http://nuand.com

[26] HackRF One: An Open Source SDR Platform. [Online]. Available: http://greatscottgadgets.com/hackrf

[27] IEEE 802.11 Working Group and others, "IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999)*, p. C1, 2009.

[28] Noctar PCI-e SDR card. [Online]. Available: http://www.pervices.com/support-noctar

[29] P. Van Eetvelt, G. Wade, and M. Tomlinson, "Peak to average power reduction for OFDM schemes by selective scrambling," *Electronics letters*, vol. 32, no. 21, pp. 1963–1964, 1996.

[30] S. Mneney, *An Introduction to Digital Signal Processing.* River Publishers, 2008.

[31] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, pp. 260–269, 1967.

[32] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo-codes," *Communications, IEEE Transactions on*, vol. 44, no. 10, pp. 1261–1271, 1996.

[33] R. Garello, G. Montorsi, S. Benedetto, and G. Cancellieri, "Interleaver properties and their applications to the trellis complexity analysis of turbo codes," *Communications, IEEE Transactions on*, vol. 49, no. 5, pp. 793–807, 2001.

[34] J. Terry and J. Heiskala, *OFDM Wireless LANs: A theoretical and practical guide.* Sams Publishing, 2002.

[35] R. v. Nee and R. Prasad, *OFDM for wireless multimedia communications.* Artech House, Inc., 2000.

[36] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.

[37] T. M. Schmidl and D. C. Cox, "Robust frequency and timing synchronization for OFDM," *Communications, IEEE Transactions on*, vol. 45, no. 12, pp. 1613–1621, 1997.

[38] L. S. Cutler and C. L. Searle, "Some aspects of the theory and measurement of frequency fluctuations in frequency standards," *Proceedings of the IEEE*, vol. 54, no. 2, pp. 136–154, 1966.

[39] J. Rutman, "Characterization of phase and frequency instabilities in precision frequency sources: Fifteen years of progress," *Proceedings of the IEEE*, vol. 66, no. 9, pp. 1048–1075, 1978.

[40] T. Pollet, M. Van Bladel, and M. Moeneclaey, "BER sensitivity of OFDM systems to carrier frequency offset and Wiener phase noise," *Communications, IEEE Transactions on*, vol. 43, no. 234, pp. 191–193, 1995.

[41] L. Rugini and P. Banelli, "BER of OFDM systems impaired by carrier frequency offset in multipath fading channels," *Wireless Communications, IEEE Transactions on*, vol. 4, no. 5, pp. 2279–2288, 2005.

[42] P. H. Moose, "A technique for orthogonal frequency division multiplexing frequency offset correction," *Communications, IEEE Transactions on*, vol. 42, no. 10, pp. 2908–2914, 1994.

[43] S. Chang and E. Powers, "Efficient frequency-offset estimation in OFDM-based WLAN systems," *Electronics Letters*, vol. 39, no. 21, pp. 1554–1555, 2003.

[44] M. Canet, I. Wassell, J. Valls, and V. Almenar, "Performance Evaluation of Fine Time Synchronizers for WLANs," in *Proc. 13th European Signal Processing Conference, pp. CD-Edited, Antalya-Turk*, 2005, pp. 4–8.

[45] A. Demir, A. Mehrotra, and J. Roychowdhury, "Phase noise in oscillators: a unifying theory and numerical methods for characterization," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, vol. 47, no. 5, pp. 655–674, 2000.

[46] *ISO/IEC 14882:2011 Information technology – Programming languages – C++.*

[47] ADF4351: Wideband Synthesizer with Integrated VCO Datasheet. [Online]. Available: http://www.analog.com/static/imported-files/data_sheets/ADF4351.pdf

[48] G. D. Forney Jr, "The Viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[49] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[50] T. Hentschel and G. Fettweis, "Sample rate conversion for software radio," *Communications Magazine, IEEE*, vol. 38, no. 8, pp. 142–150, 2000.

[51] Linux wireless. [Online]. Available: http://wireless.kernel.org

[52] The Boost C++ Libraries. [Online]. Available: http://www.boost.org

[53] A. V. Oppenheim and R. W. Schafer, *Discrete-time signal processing.* Prentice-hall Englewood Cliffs, 2009, vol. 3.

[54] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Efficient vectorization of the FIR filter," in *Proc. 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC).* Citeseer, 2005, pp. 432–437.

[55] F. De Mesmay, S. Chellappa, F. Franchetti, and M. Püschel, "Computer generation of efficient software Viterbi decoders," in *High Performance Embedded Architectures and Compilers.* Springer, 2010, pp. 353–368.

[56] S. E. Anderson, "Bit twiddling hacks," 2005. [Online]. Available: https://graphics.stanford.edu/~seander/bithacks.html

[57] M. Windisch and G. Fettweis, "Performance degradation due to I/Q imbalance in multi-carrier direct conversion receivers: A theoretical analysis," in *Communications, 2006. ICC'06. IEEE International Conference on*, vol. 1. IEEE, 2006, pp. 257–262.

[58] R. A. Shafik, S. Rahman, and R. Islam, "On the extended relationships among EVM, BER and SNR as performance metrics," in *Electrical and Computer Engineering, 2006. ICECE'06. International Conference on.* IEEE, 2006, pp. 408–411.

[59] J. Levon and P. Elie, "Oprofile: A system profiler for linux," 2014.