Software Defined Radio OFDM Implementation, Mixed Signal Circuit Design and EBG Antenna Design

Joseph Nguya Wamicha

A dissertation submitted to the Department of Electrical Engineering,
University of Cape Town, in fulfilment of the requirements
for the degree of Master of Science in Electrical Engineering.

Cape Town, January 2011

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author .....

Cape Town
January 2011

Abstract

So how did my interest in software defined radio and antennas come about? It all started in the month of February, 2006 when two of us deranged and hungry friends were trying our hands at entrepreneurship. We managed to convince one of the 60 kbps - 100kpbs CDMA2000 mobile networks at the time, Popote, to provide a video streaming premium rated service to their clients. The market was there and the mobile network was interested; all we had to do was deliver. So for months we hacked at the video streaming codecs and protocols. We finally managed to get video streaming working to the mobile device working (which was quite a feat at the time as video streaming codecs weren't nearly as versatile as today), but no matter how hard we tried to optimise the codecs, we still couldn't get a high definition video stream to the mobile cdma2000 device. It goes without saying that we lost a potentially lucrative product, Popote could not survive since they were unable to provide the premium rated services larger GSM networks such as Safaricom (of MPESA fame) were providing, and yes, we burnt our hands at entrepreneurship like so many others before us and continued to remain hungry and broke.

There was a positive outcome from the experience though. The question still remained: why were we unable to efficiently stream a video, despite our having used highly optimized video streaming algorithms? It was then that it struck me that this was a problem I'd never be able to solve using software alone. The bottle neck was in the hardware and I'd need to get into it's murky guts to solve this problem once and for all. It aroused inside me a curiosity inside me about wanting to find out in detail how mobile networks work so that I could finally solve my problem. This curiosity finally led me into UCT, where I pursued my research. During my research, I discovered that the two bottle necks were with the antenna and base station hardware. I sought to understand how these two modules of the mobile network work and it is the results that form the bulk of my thesis. Since I believe that the only way Africa can develop is by developing it's own intellectual property using open source software tools, I sought to use only open source software and hardware during my research. Besides, open source software is free of licensing costs and is a lot more affordable for the price conscious African market. It is also open to anyone who wants to understand how things work. Hopefully with the end of my Masters research project, I can finally set this curious case to rest.

Due to it's potential to revolutionize modern digital communications, Software Defined Radio (SDR) has become a hot research topic in recent years. This is because using Software Defined Radio, we can

re-create radio communications blocks such as modulators, demodulators, filters and amplifiers from their traditional hardware implementations into new next generation, re-configurable sofware based radio communications components. The software radio components will run on a base hardware platform such as the traditional x-86 computing platform, GPUs (General Processing Units) and more recently FPGAs (Field Programmable Gate Arrays).

In theory, since the software running on the base hardware is inter-changeable, this would make software defined radio communications a lot more versatile than its hardware counterpart. The software defined radio may for example act as a WiMAX base station today, and through altering the software running on the base hardware, act as an LTE base station tomorrow. Without having to do any modifications on the base hardware, the Software Defined Radio is also upgradeable. As a result, software defined radios would possibly have much longer lifetimes than pure hardware-based implementations, thereby making them a lot cheaper to invest in, in the long run. For example, through a software upgrade, the base hardware may change from being an LTE base station to an Advanced LTE base station.

In the first part of my thesis I seek to verify the plausibility of SDR causing a revolution in next generation radio communications by doing an actual OFDM implementation for the IEEE, 802.11g WLAN (Wireless Local Area Network) specification using the Gnuradio SDR platform.

In the second part of my thesis I examine open source electronic design automation using gEDA, by designing a front-end for a new low cost FPGA board called Rhino (Reconfigurable Hardware Interface for Computing and Radio). Rhino is capable of far more powerful DSP (Digital Signal Processing) than the more popular Gnuradio, and would provide communications engineers with a much better platform for wireless communications prototyping. Since the Rhino FPGA board does not have ADC or DAC chips for analog to digital or digital to analog conversion processes respectively, my work was to design the Rhino Expansion Board which is an ADC/DAC mezzanine board designed for Gnuradio RF Acquisition daughterboards to plug into the Rhino FPGA board. This would enable the SDR community to utilize the processing power of the Rhino FPGA Board. Also, the Rhino expansion board would allow re-usability of already proven hardware, instead of designing new RF Acquisition hardware from scratch.

In the second part of my thesis I also seek to establish whether open source EDA has matured enough for it to be used by Industry. If it is established that this is indeed the case, then this could potentially lower the cost of designing new hardware, which would be a huge benefit to lowering the barriers of entry to technology manufacturing for the nascent African electronics industry.

Finally in the last part of my thesis I use the open source Gnuradio SDR platform to test an EBG (Electronic Band Gap) edge-fed microstrip patch antenna. The open source Openmoko mobile phone can be used to collect receive sensitivity (power) and antenna range measurements. EBG structures are used in patch antenna designs since they allow us to make meta-materials with new EM properties that are not

ordinarily found in nature. As a result, further research that may be carried out on EBG antennas could potentially be used to design far more sensitive nano antennas in future.

We recommend that the pdf version of the thesis be read, to aid in better navigation of the content using the hyperlinks.

Acknowledgements

I would like to extend a very special thanks to my supervisor Dr. Simon Winberg for his incredibly good insight, advice and remarkable help throughout the duration of my research. I would also like to thank Dr. Alan Langman, Simon Scott, Jonathan Ward, the Rhino and Radar Lab teams, the iPay team and especially Henty (who provided me with a part-time job and flexible working hours without which I would not have been able to survive the year at UCT), family and friends for their incredible guidance, encouragement and support during the writing of this thesis.

This work is dedicated to my mum and dad.

Contents

Glossary

10GigE — 10 Gigabit Ethernet. 71, 72

1GigE — Gigabit Ethernet. 11, 14, 70

ADC — Analog to Digital Converter. 8, 12, 70, 72–77, 81, 93

AP — Access Point. 46, 47

ARM — Advanced RISC Machine. 15, 71–73

ASIC — Application Specific Integrated Circuit. 5, 6

BPSK — Binary Phase Shift Keying. 34, 42, 46

BSSID — Basic Service Set Identifier. 63

CAD — Computer Aided Design. 78

CDMA — Carrier Division Multiple Access. 58

CPLD — Complex Programmable Logic Device. 10, 11

CRC — Cyclic Redundancy Check. 46, 49

DAC — Digital to Analog Converter. 8, 11, 70, 72–76, 78, 81, 93

DDC — Digital Down Conversion. 12, 70

DDR — Double Data Rate. 12

DFT — Discrete Fourier Transform. xx, 37, 39, 40

DRC — Design Rule Checker. 78

DSP — Digital Signal Processing. 71, 72

DUC — Digital Up Conversion. 11, 70

DVB — Digital Video Broadcasting. 1, 6, 148

EBG — Electromagnetic Bandgap Structures. 2, 4, 5, 7, 9, 105, 108–112, 141, 142, 147, 148

EDA — Electronic Design Automation. 78, 80

EM — Electromagnetic. 106, 109

FDTD — Finite Difference Time Domain Method. 7

FEC — Forward Error Correction. 148

FEM — Finite Element Method. 7

FFT — Fast Fourier Transform. xix, xx, 34, 37, 40

FMC — FPGA Mezzanine Connector. 71–78, 83, 95

FPGA — Field Programmable Gate Array. 2, 3, 6, 8, 10–14, 19, 21, 22, 25–27, 31, 70–78, 149

gEDA — GPL Electronic Design Automation (an open source software suite for EDA). 78–80, 149

Gnuradio — The most popular open source software defined radio project to date. 1, 3, 6, 8, 10–14, 17, 26, 27, 33, 58, 59, 61, 63, 67, 70–73, 75, 76, 79, 148

GPIO — General Purpose Input Output. 71

GUI — Graphical User Interface. 137

HDL — Hardware Description Language. 72

IDFT — Inverse Discrete Fourier Transform. xx, 37, 40

IEEE — Institute of Electrical and Electronics Engineers. 1, 3, 6, 71

IF — Intermediate Frequency. 11, 12, 21, 22, 73, 74, 78

IFFT — Inverse Fast Fourier Transform. xx, 37, 40, 52

ISI — Inter Symbol Interference. 41

LPC — Low Pin Count. 71, 74–76

LPF — Low Pass Filter. 12

LTE — Long Term Evolution. 1, 148

LVDS — Low Voltage Differential Signaling. 22, 74, 75, 77, 78

MAC — Medium Access Control. 41, 43, 46–49, 63

MB — Microblaze processor. 14

MIMO — Multiple-Input Multiple-Output. 42, 71

MoM — Method of Moments. 7

NAK — Negative Acknowledgement. 49

NAND — A binary operator composite of NOT AND; negation of AND function. 15, 16

NCO — Numerically Controlled Oscillator. 22, 25

NOR — A binary operator composite of NOT OR; negation of OR function. 15, 16

OFDM — Orthogonal Frequency Division Multiplexing. xix, 1–4, 6, 8, 10, 12, 14, 19, 26, 33–36, 40–43, 45, 46, 50, 52–54, 56, 58, 59, 61, 63, 65, 67, 72, 147–149

PC — Personal Computer. 3, 10–14, 16, 26, 27, 70, 71

PCB — Printed Circuit Board. 78, 79, 149

PLCP — Physical Layer Convergence Procedure. 34, 41–46, 52, 53, 61

PLL — Phase Locked Loop. 22, 25, 31

PMC — PCI Mezzanine Card. 3, 10–12, 21, 22, 25, 70, 72–78, 85, 97

PPDU — PLCP Protocol Data Unit. 53

PSDU — PLCP Service Data Unit. 42–46

QAM — Quadrature Amplitude Modulation. 35, 36, 46, 148

QPSK — Quadrature Phase Shift Keying. 35, 46

RF — Radio Frequency. xix, 3, 6, 8, 10, 11, 19, 21, 22, 25, 26, 70–73, 76, 148, 149

RFX2400 — Flex 2400 daughter board from Ettus Research. 2, 3, 8, 10, 11, 19, 21–23, 25, 26, 31, 54, 59, 61, 67, 74, 75, 78

Rhino — Reconfigurable Hardware Interface for Computing and Radio. 1–4, 6, 8, 19, 22, 25, 48, 58, 59, 61, 63, 65, 67, 71–79, 81, 83, 85, 87, 89, 93, 95, 97, 99, 101, 103, 148, 149

RMSA — Rectangular Microstrip Antennas. 107

SD card — Secure Digital Card. 10, 11, 13, 14, 16

SDR — Software Defined Radio. 1–3, 5, 8, 10, 58, 70, 147, 148

SMA — SubMiniature version A. 21, 26, 71, 73, 76, 111

SPI — Serial Peripheral Interface Bus. 74–76

SSID — Service Set Identifier. 58, 59, 61, 63, 65, 67

UCT — University of Cape Town. 1, 2, 12, 31

UHD — Universal Hardware Driver written by Ettus Research. 11, 12, 26

USB — Universal Serial Bus. 15, 16, 71

USRP — Universal Software Radio Peripheral from Ettus Research. 31

USRP2 — Universal Software Radio Peripheral 2 from Ettus Research. 2, 3, 5, 6, 8, 10–12, 14, 19, 22, 25–27, 31, 54, 59, 61, 67, 70–74, 76, 148

VCTCXO — Voltage Controlled Temperature Compensated Crystal Oscillator. 22, 25

VHDL — VHSIC Hardware Description Language; VHSIC: very-high-speed integrated circuit. 72, 79, 80

WEP — Wired Equivalent Privacy. 48

WiFi — WiFi is a registered IEEE trademark. The trademark is used on 802.11 compliant devices.. 15

WiMAX — Worldwide Inter-operability for Microwave Access. 1, 6, 148

WLAN — Wireless Local Area Network. 1–3, 6, 10, 18, 19, 41, 42, 44–49, 52, 56, 58, 59, 61, 63, 65, 67, 147, 148

List of Figures

List of Tables

List of Symbols

B	—	Transmitted RF bandwidth
c	—	Speed of light
λ	—	Wavelength
N_{DSC}	—	Number of data subcarriers per OFDM symbol
N_{CBPSC}	—	Number of coded bits per OFDM subcarrier
N_{CBPS}	—	Number of coded bits per OFDM symbol
R	—	Coding Rate
N_{DBPSC}	—	Number of data bits per OFDM subcarrier
N_{DBPS}	—	Number of data bits per OFDM symbol
N_{DATA} = Total 802.11 PLCP Frame data bits		
N_{PAD} = Total 802.11 PLCP Frame pad bits		
ΔF	—	Subcarrier frequency spacing
T_{FFT}	—	OFDM subcarrier duration or period
T_{GI}	—	OFDM guard interval between each transmitted subcarrier
T_{SYM}	—	OFDM symbol duration or period
N_{SYM}	—	Number of OFDM symbols per second
CR	—	Code bit rate per second
DR	—	Data bit rate per second
$\bar{E}$	—	The electric field intensity in volts per meter (V/m).
$\bar{B}$	—	The magnetic flux density in webers per square meter ($Wb/$
$\bar{H}$	—	The magnetic field intensity in volts per meter (A/m).
$\bar{D}$	—	The electric flux density in coulombs per square meter (C/m
$\bar{J}$	—	The electric charge density in coulombs per cubic meter ($C/$
$\nabla\times$	—	The curl operator.
$\nabla.$	—	The divergence operator.
$\bar{D}$	—	$\epsilon_0\bar{E}$ in free space
$\bar{B}$	—	$\mu_0\bar{H}$ in free space
ϵ_0	—	Permittivity of free space
ϵ_r	—	Relative permittivity of substrate
ϵ_{eff}	—	Effective permittivity of substrate
μ_0	—	Permeability of free space
ρ	—	Charge density

$f(t)$	Waveform in the time domain
c_0	DC offset of the waveform
a_n	Fourier coefficient for the sine component of the waveform
b_n	Fourier coefficient for the cosine component of the waveform
c_n	Fourier coefficient for the basis function of the waveform represented using only cosine components
n	Waveform harmonic which is an integer multiple of the fundamental waveform frequency
ϕ_n	Phase shift of sine or cosine waveform component
f	Fundamental waveform frequency
N	The total number of harmonics of the waveform
ω	Radial frequency
f	Linear frequency
N	Total number of samples taken for the waveform in the case of IDFT and IFFT. In the case of DFT and I
k	Waveform sample which is an integer multiple.
τ	Time difference between each sample point. t
$f(n)$	waveform in time domain represented as a function of k $F(k)$ frequency domain sample points
$F(k)$	waveform in frequency domain represented as a function of n $f(n)$ time domain harmonics
L	—
L_{eff}	—
ΔL	—
W	—
h	—
c	—
f_0	—
BW	—
λ_0	—

Chapter 1

Introduction

1.1 Background

The thesis was the result of research into what factors in wireless communications could be improved so as to improve mobile network data rates. In addition, the OFDM transmitter that was to be developed as a result of the research could possibly be used for OFDM passive radar applications in future. All wireless protocols from WLAN and DVB to WiMAX and LTE, essentially have an OFDM core. Since WLAN chips and related tools and applications are more readily available, the WLAN protocol was selected for the OFDM implementation described in this thesis. WLAN networks have also become very pervasive in the recent past and the existing installations could potentially be useful for acquiring data on moving targets. An OFDM implementation done using SDR would provide the UCT Radar Remote Sensing Group with an versatile platform for taking measurements and analyzing the effects of WLAN network implementations on different OFDM Radar applications. An added benefit of the OFDM SDR implementation is that a single platform could be used to simulate different passive radar conditions that would be encountered in a real-life environment. The OFDM implementation was to be done using Gnuradio intially, and eventually transferred to Rhino, once development of the generation one Rhino Board was complete.

1.2 User Requirements

The following user requirements had to be met by the glsIEEE 802.11 WLAN OFDM implementation:

1. The 802.11g protocol had to be implemented as described in the IEEE 802.11-2007 specification document.
2. The OFDM symbols transmitted by the SDR platform had to be successfully demodulated and received by conventional off-the-shelf WLAN chips.

3. We had to find a way to take antenna receive sensitivity measurements off the WLAN chip that was receiving our OFDM transmission.

4. An antenna with improved directivity had to be fabricated using custom materials and results demonstrated.

5. A mixed signal circuit was to be designed for the new FPGA board from UCT called Rhino. The implemented OFDM code could potentially be transferred to this new and more powerful board in future.

1.3 Scope and Limitations

1. In order to take the receive sensitivity and power range measurements on a receiving device, the SDR OFDM implementation only needed to be a transmitter.

2. The OFDM symbols were to be transmitted at between 2.4 GHz and 2.5 GHz. This is because the research group only had the RFX2400 daughter board available which has an operational frequency range of between 2.25 GHz and 2.9 GHz.

3. The electronic chip components used on the Rhino expansion board were to be as close as possible to the USRP2, to ensure compatibility with the USRP2 daughter boards.

4. The EBG edge-fed microstrip antenna had to be designed using 20 mil thick, Rogers R04003C substrate material, since this was what was available in the labs.

1.4 Dissertation Overview

Chapter 2: Literature Review

Chapter 2 gives a brief outline of some of the literature currently available on the topics of Software Defined Radio (SDR), the Universal Software Radio Peripheral 2 (USRP2) and EBG Patch Antenna Designs.

Chapter 3: Research Methodology

In Chapter 3 we describe the steps and processes that will be followed while conducting our research, so as to ensure our objectives are met.

Chapter 4: Setting up the Gnuradio, USRP2 and Openmoko OFDM Testbed

Chapter 4 describes all the steps and procedures that were followed in order to setup our OFDM test bed. The test bed we setup made use of the Gnuradio, USRP2, and Openmoko projects. We show how Gnuradio was run on the host PC and was used to generate the WLAN OFDM symbols. The chapter also describes how the USRP2 was used to transmit the OFDM symbols using the RFX2400 RF transmitter board attached onto its PMC connectors. Finally, we look at how the Neo Freerunner phone from Openmoko provided a battery powered mobile hand-held device that could be used to detect the transmitted WLAN OFDM symbols at a distance from the USRP2-RFX2400 transmitter.

Chapter 5: RFX2400 Description

In chapter 5 we give a detailed description of the RFX2400 daughter board that is manufactured by Ettus Research. The receive and transmit chains of the RFX2400 are also carefully examined. The RFX2400 RF aquisition and transmission board will be used to transmit our OFDM symbols.

Chapter 6: SDR OFDM Implementation using Gnuradio

In chapter 6 we give a detailed step by step explanation of the process we used to implement the IEEE 802.11g WLAN specification using Gnuradio.

Chapter 7: Analysis and Results of the OFDM SDR Implementation using off-the-shelf WLAN chips

Chapter 7 presents the results of our SDR OFDM implementation by showing how several off the shelf chips were able to detect our Gnuradio Rhino SSID beacon frame WLAN OFDM transmission.

Chapter 8: Rhino Expansion Board Design

Chapter 8 presents the design considerations that were taken into account while designing the Rhino expansion board. Since the Rhino expansion board would act as a hardware bridge between the Rhino FPGA board and the RF daughter boards from the Gnuradio project, this chapter looks at how compatibility with the gnuradio daughter boards can be maintained. We also look at reducing the number of components on the Rhino expansion board so as to reduce the cost of upgrading from the USRP2 to the Rhino FPGA board, for academic research institutions.

Chapter 9: Rhino Expansion Board, Schematics, Netlist and Bill of Materials

Chapter 9 presents the final Rhino expansion board schematics and bill of materials that were generated after going through the initial design process.

Chapter 10: Edge Fed EBG Microstrip Patch Antenna Generator Script

In this chapter we describe the theory behind the Edge Fed EBG Microstrip Patch antenna that was fabricated. We also look at the mpatchebgcalc.py script which was developed to help with the rapid prototyping of new EBG patch antenna designs.

Chapter 11: Simulation, Results and Analysis of the Fabricated Microstrip Patch Antennas

This chapter presents the directivity and bandwidth results of our edge fed microstrip patch antennas with EBG structures. The patch antennas were fabricated using Rogers R04003 substrate material.

Chapter 12: Conclusions and Recommendations

Based on the work and findings of the research conducted during the writing of this thesis, this chapter presents conclusions that were made and makes some recommendations for future research.

Appendix

The Appendix presents the OFDM Code, Rhino expansion board schematics, Rhino expansion board bill of materials, EBG antenna design code and antenna return loss graph plots.

Bibliography

This bibliography section contains a list of all the reference text used during our research.

Chapter 2

Literature Review

2.1 Overview

This chapter gives a brief outline of some of the literature currently available on the topics of Software Defined Radio (SDR), the Universal Software Radio Peripheral 2 (USRP2) and EBG Patch Antenna Designs.

2.2 Software Defined Radio Literature Review

Most radio systems in production today are currently implemented using Application Specific Circuits (ASICs) that usually only allow communications using a single type of waveform and modulation mode. Though ASIC communications circuits were relatively cheap to mass produce, devices using ASIC chips could communicate using only one physical layer protocol. Thus, two electronic devices using ASIC circuits fabricated for different communication protocols were unable to speak to each other. In addition, with the rapid evolution of digital communications that has occurred in the last 20 years, electronic devices using ASICs for communications were quickly becoming obsolete after only a few years of use. These were the challenges being experienced when *Joe Mitola* first coined the term *Software Radio* in his classic paper *The Software Radio Architecture* ([55]). In his classic paper, *Joe Mitola* proposes the partitioning of channel coding into the following functional components: antenna, radio frequency, analog to digital conversion (reception), digital to analog conversion (transmission) and digital signal processing. Signal processing functions such as modulators, demodulators, mixers, filters and amplifiers, which vary between different communication protocols, could now be implemented using software. This would now allow an electronic device to communicate using different communication protocols by simply reconfiguring the abstract software application layer, thereby overcoming the more prohibitively expensive option of having to replace the underlying communications hardware.

In order to accurately model real life network conditions and to prove that the software defined radio

could indeed replace the ASIC communications chip in future, the OFDM software defined radio implementation would have to be compatible with ASIC chips already in the field. So as to ensure compatibility with off-the-shelf OFDM based communications chips, an open standard would have to be implemented on the software defined radio. The open standards could possibly be implemented were the IEEE 802.11 WLAN standard ([13]), the IEEE 802.16 WiMAX standard ([14]) and the DVB-T2 standard ([12]). The Gnuradio project already comes with generic OFDM signal processing blocks ([26]) so ideally it would be possible to reuse some of their code for our OFDM implementation.

2.3 USRP2 Literature Review

The USRP2 is a mixed signal circuit and FPGA board rolled together into 1 board ([24]). It supports a variety of daughter boards operating at frequency ranges between 1 MHz and 6 GHz ([27]). By designing the Rhino expansion board around the USRP2 board, we could be assured of the Rhino FPGA board's ability to work with RF daughter boards that cover a very wide range of frequencies. The Rhino expansion board would therefore allow RF daughter boards from the Gnuradio project to be used with the more powerful Rhino FPGA board for Telecommunications and Radar radio spectrum research problems ([76]). In general, the USRP2 was unfortunately not very well documented in literature.

2.4 EBG Patch Antenna Designs Literature Review

In the past 40 years, patch antennas have become extremely popular due to their small form factor and conformability, which is especially suitable for use inside embedded devices. ([68]). Patch antennas unfortunately have very low directivity, and increasing directivity normally comes with the unfortunate, mutually exclusive effect of reducing the corresponding 3 dB bandwidth ([67]).

Advances in nanotechnology have enabled great breakthroughs in processor chip fabrication, hence allowing Moore's law to still hold true 30 years after it was first promulgated ([8]). However, there appears to have been very little research done into using nanotechnology to improve the materials and hence quality characterization of current antenna designs. As prophetically predicted almost 25 years ago by Eric Drexler's classic nanotechnology text, *Engines of Creation: The Coming Era of Nanotechnology* ([59]), custom nano materials such as carbon nanotubes, nanowires, nanotubes and quantum dots have been discovered, allowing us to fabricate stronger materials, more coherent lasers and more energy efficient electrical appliances. These nano materils could therefore be potentially used to fabricate new antenna designs with far greater receive sensitivity ([59], [19]).

Since the required nanotechnology fabrication facilities are not available in Cape Town, the custom material we will use to fabricate our antenna with will have to be constructed at the macro level. Photonic

Band Gap structures were the answer we found to our predicament. These structures could be fabricated at both the macro and nano level ([54]). Photonic band gaps can be used to control the flow of the various frequency components of light. By extension, the same photonic or electronic band gap structures could therefore be used to control propagation of electromagnetic waves at different frequencies ([112]). If we can prove that fabricating electromagnetic bandgap (EBG) structures at the macro level helps to improve the quality characteristics of the antenna, then by extension fabricating the antennas using custom nano materials could potentially have enormous benefits on future antenna designs.

Papers showing the benefits of EBG structures can be found in various texts ([45], [5]). There however exists no literature showing how the EBG dimensions can be predictably fabricated to influence electromagnetic wave propagation accordingly. All the literature texts encountered seem to start with random EBG dimensions, and then use antenna modelling software to converge on the appropriate dimension to enhance the far field for example. The thesis should therefore investigate how we can more predictably determine the dimensions of the EBG antennas to be fabricated.

Antenna modelling software can be used to converge on the best antenna dimensions. The antenna modelling software uses techniques such as Method of Moments (MoM), Finite Difference Time Domain Method (FDTD), Finite Element Method (FEM) and genetic algorithms to converge on the best solution. Some of the more popular simulation software includes Feko, Antenna Magus and Ansoft HFSS ([3]).

Chapter 3

Research Methodology

3.1 Overview

This chapter describes the steps and research process that will be used to conduct our research so as to ensure our objectives are achieved.

3.2 SDR OFDM Implementation

An SDR OFDM implementation will be done on the USRP2 and working at 2.4 GHz. The OFDM symbols would be generated by the Gnuradio open source SDR platform. The RFX2400 daughter board will be used for the experiments since it is able to transmit and receive at between 2.25GHz and 2.9GHz. We will also need to show that our OFDM symbols can be correctly decoded and signal power measured at the receiver.

3.3 Mixed Signal Circuit Design

The Rhino expansion board will be designed for the more powerful Rhino FPGA board being built. The Rhino expansion board will essentially consist of an ADC, DAC, Voltage Controlled Temperature Compensated Crystal Oscillator and Synchronization Clock for generating the clocks when sampling a signal during RF transmission or reception. The Rhino expansion board will also act as a hardware bridge to Gnuradio's USRP2 RF acquision and transmission daughter boards.

3.4 Designing, Modelling and Fabricating an improved Patch Antenna

An antenna with improved directivity was to be designed using custom meta materials. Since there are no nanotechnology facilities available in Cape Town to design new materials with at the nano level, these meta materials were to be designed using EBG structures at the macro level. The antenna design would be modelled using the Feko Modelling software. The improved antenna design could potentially be used to improve transmit data rates in future since less data rate errors would occur even with less receive power strength (hence allowing us to improve our coding rate).

Chapter 4

Setting up the Gnuradio, USRP2 and Openmoko OFDM Testbed

"Free software for building radios is troublesome to some people. In the US, we've run into opposition from the Motion Picture Association of America and its attempt with the Broadcast Flag to restrict the kinds of receivers that can be built for over-the-air digital TV. The US Federal Communications Commission has issued a Notice of Proposed Rule Making (NPRM) concerning Cognitive Radio Technologies and Software Defined Radios. Several troublesome issues are raised in the NPRM, including restricting the sale of high-speed digital-to-analog converters, requirements for digital signatures or similar methods to keep unauthorized software out of software radio hardware and new restrictions on radios built for the amateur radio market." - Eric Blossom, founder and architect of the open source Gnuradio SDR project.

4.1 Overview

This chapter describes all the steps and procedures that were followed in order to setup our OFDM test bed. The test bed made use of the Gnuradio, USRP2, and Openmoko projects. Gnuradio was run on the host PC and was used to generate the WLAN OFDM symbols. The USRP2 was used to transmit the OFDM symbols using the RFX2400 RF Transmitter board attached onto its PMC connectors. Finally, the Neo Freerunner phone from Openmoko provided a battery powered mobile hand-held device that could be used to detect the transmitted WLAN OFDM symbols at a distance from the USRP2-RFX2400 transmitter.

4.2 Gnuradio Boot process

Each time the USRP2 was powered up, the Xilinx XC9572 CPLD chip would read the first Megabyte of memory from the SD card and write it to the FPGA (Field Programmable Gate Array). The FPGA would use this data configure itself and then request for the aeMB microblaze processor firmware from

the next section of memory in the SD card. Once the aeMB processor firmware had been loaded onto the FPGA, the FPGA would then switch the CPLD to passthru mode and take over control of the SD card. If the FPGA boot process was successful, then LEDs D (firmware loaded) and F (FPGA loaded) on the USRP2 would both light up.

4.3 Gnuradio Transmit Chain

During the transmit chain, the Gnuradio host PC code was used to send out the digital waveform via the 1GigE ethernet port on the host PC to the 1GigE ethernet port (0826-1X1T-23-F Gigabit Ethernet Single Port MagJack) on the USRP2.

From the 1GigE ethernet port, the digital waveform was then sent out to the FPGA, where the FPGA performs a Digital Up Conversion (DUC) of the signal. During DUC, the baseband (centered around zero frequency) waveform is modulated onto a digital IF (Intermediate Frequency) waveform of 100 MHz. Using the Gnuradio C++ signal processing blocks, the interpolation used by the FPGA during DUC was configured from the host PC via the set_interp method:
http://gnuradio.org/doc/doxygen/classusrp2__sink__base.html.

After the digital waveform was up converted, the FPGA then sent the IF frequency waveform to the dual channel 16-bit, 400 MSamples/sec ADC9777 DAC from Analog Devices. On the first channel, the FPGA sent the I component of the complex digital signal, while on the second channel, the Q component of the complex digital signal was sent.

After the DAC received the separate I and Q channel signals, each separate I and Q channel was converted into it's analog waveform equivalent. The DAC then sent through the I and Q channel analog waveform through a single transmit PMC connector. Please note that the Gnuradio daughter board is only an RF-Transmission and RF-Acquistion (RF reception) mezzanine board.

During transmission, LED A would light up when the UHD (Gnuradio Universal Hardware Driver) FPGA code was used:
(http://www.ettus.com/uhd_docs/manual/html/usrp2.html).

To test transmission, the usrp2_siggen.py script, which produces a single tone at a specified frequency, was used. The usrp2_siggen_gui.py script can also be used for this test. Since we tested transmission using the RFX2400 daughter board which works at between 2.25GHz and 2.9GHz, the following command was used to test signal transmission using the RFX2400:

```
#usrp2_siggen.py -f 2.4e9 -e eth0
```

Successful transmission of the 2.4GHz signal was verified by detecting and viewing the transmitted waveform using the Agilent E4407B spectrum analyzer in the UCT Microwave Lab.

4.4 Gnuradio Receive Chain

During the receive chain, the complex analog IF waveform was acquired by the Gnuradio daughter board and then sent to the USRP2 via a PMC connector. The complex I and Q channel analog signal came in through the second receive PMC connector. The analog I and Q channel analog signals are at an IF frequency of 100MHz. From the PMC connector on the USRP2, the IF frequency analog waveform then travelled to the 14-bit, 100 MSamples/sec LTC 2284 ADC from Linear Technologies, where each separate I and Q analog waveform signal was converted into its digital equiavalent. The 2 complex separate I and Q channels were then sent to the FPGA for Digital Down Conversion (DDC). During DDC, the FPGA would take the IF signal through a Low Pass Filter (LPF) and decimation stage to produce two separate I and Q channels at baseband frequency. The separate baseband I and Q digital waveforms were then sent to the host PC via the 1 GigE ethernet port.

During reception, LED C would light up when the UHD (Gnuradio Universal Hardware Driver) FPGA code was used:
http://www.ettus.com/uhd_docs/manual/html/usrp2.html.

To test reception the usrp2_fft.py application was used. The usrp2_fft.py command used to view a 10dBm 2.4 GHz signal being transmitted by the HP 8350B Oscillator is shown below:

```
#usrp2_fft.py -e eth0 -f 2.4e9
```

4.5 Installing Gnuradio on the Host PC

Ubuntu Linux version 10.04 LTS (long term release, code named Lucid Lynx) and Ubuntu Linux version 10.10 (code named Maverick Meerkat), was used for the Gnuradio OFDM implementation. Both the 32 and 64 bit Ubuntu Linux versions were tested on the 64 bit Intel 2.17GHz, dual core test machine. In order for the Gnuradio OFDM code implementation to work on the 64 bit machine, the DDR 667 RAM had to be upgraded from 2GB to 4 GB. This is because the Gnuradio buffer or memory allocation signal processing block is not 64 bit compatible, since it would allocate 64 bits of memory for a 32 bit memory location. The 64 bit version of the Gnuradio OFDM implementation therefore gobbles up double amount of memory used by the 32 bit version (See gr_vmcircbuf_sysv_shm_factory). The

following steps were followed to install Gnuradio:

```
#apt-get -y install git-core

#apt-get -y install libfontconfig1-dev libxrender-dev \
libpulse-dev swig g++ automake libtool python-dev \
libfftw3-dev libcppunit-dev libboost-all-dev \
libusb-dev fort77 sdcc sdcc-libraries libsdl1.2-dev \
python-wxgtk2.8 subversion git-core guile-1.8-dev \
libqt4-dev python-numpy ccache python-opengl \
libgsl0-dev python-cheetah python-lxml doxygen
qt4-dev-tools libqwt5-qt4-dev libqwtplot3d-qt4-dev \
pyqt4-dev-tools

#git clone http://gnuradio.org/git/gnuradio.git

#cd gnuradio

#./bootstrap && make && make check && make install
```

Gnuradio was now installed on the host PC.

4.6 Preparing the USRP2 boot SD card

The microblaze firmware image (for the aeMB microblaze processor) and FPGA code were written onto the SD card using the u2_flash_tool found under the Gnuradio directory that was checked out from the git repository (as shown in the previous section titled *Installing Gnuradio on the Host PC*). The microblaze firmware and FPGA code were downloaded from the following link:
http://code.ettus.com/redmine/ettus/projects/public/wiki/U2binaries.
The precompiled stable firmware image used for our experiments was the txrx_raw_eth_20100608.bin image, while the stable precompiled FPGA image used was the u2_rev3-20100603.bin image. The following commands were used to write the images onto the card:

```
#gnuradio/usrp2/firmware/u2_flash_tool --dev=/dev/sdb \
 -t s/w txrx_raw_eth_20100608.bin -w
#gnuradio/usrp2/firmware/u2_flash_tool --dev=/dev/sdb \
 -t fpga u2_rev3-20100603.bin -w
```

where devsdb is the device onto which the SD card was mounted on Linux.

Whenever any changes were made to the FPGA image or whenever we needed to use the latest FPGA image from the ettus git repository, the following steps were followed in order to successfully recompile the FPGA image:

1. Xilinx ISE tools was installed on the Intel 2.66 GHz,quad-core 64-bit Ubuntu 10.4 LTS PC. The following guide was used for the ISE installation:
http://ubuntuforums.org/archive/index.php/t-1547435.html. The ISE binaries located in the $/opt/Xilin$ directory, were included in the export PATH.

2. The patched MB-gcc binary from Gnuradio was installed as described on the following page: http://gnuradio.org/redmine/wiki/1/USRP2UserFAQ.

The firmware image was compiled as shown below:

```
#cd usrp2/top/u2_rev3/
#make
#ls build/u2_rev3.bin
```

The FPGA image was compiled as shown below:

```
#cd usrp2/top/u2_rev3/
#make
#ls build/u2_rev3.bin
```

The USRP2-Gnuradio platform was now ready for use as our 802.11g OFDM symbols transmitter.

4.7 Additional Notes

1. Please note that the Gnuradio daughter board had to be plugged into the USRP2 and Gnuradio installed on the host PC in order for this test to be conducted successfully. http://gnuradio.org/redmine/wiki/
2. The 1GigE ethernet port is registered as eth0 device on our test Gnuradio host PC. Depending on the number of ethernet ports on your PC, it may sometimes be registered as ethX, where X is a number eg eth1, eth2...

4.8 Introduction to Openmoko

Openmoko is an open source mobile phone hardware and related software project based in Taiwan (http://wiki.openmoko.org/wiki/Main_Page). Their latest open source phone, the Neo FreeRunner, was used as the OFDM client mobile device for our test bed setup. The Neo FreeRunner from Openmoko has the following features:

- High resolution touch screen (43mm x 58mm, 480x640 pixels)
- 128MB SDRAM memory
- 256 MB integrated flash memory (expandable with microSD or microSDHC card)
- microSD slot supporting up to 16GB SDHC (Secure Digital High Capacity) cards (Supported microSD cards). Minimum Class 4 SD card supported.
- Internal GPS module

- Bluetooth module
- 802.11 b/g WiFi ar6k chip
- 400Mhz ARM processor
- 2 3D accelerometers
- 2 LEDs illuminating the two buttons on the rim of the case (one bicolor [blue|orange] behind the power button, 1 unicolor [red] behind the aux button)
- Tri-band GSM and GPRS
- USB Host function with 500mA power, allowing you to power USB devices for short periods (will drain the FreeRunner battery faster)
- The tri-band Neo FreeRunner is is available for the GSM bands of North America (850/1800/1900 Mhz) and the rest of the World (900/1800/1900 Mhz)

The Neo FreeRunner has two boot loaders available. The first and default bootloader available for the FreeRunner is U-Boot, which comes pre-installed inside the FreeRunner's NAND flash memory. One may also boot into the FreeRunner's operating system using the U-Boot bootloader via NOR flash. To log into U-boot via NAND flash, the following steps were followed:

- Both the power and AUX buttons were held down for 5 to 8 seconds.
- We were then logged into U-Boot inside the NAND flash.
- The AUX button was used to select one of the options and the Power button used to boot into the selected operating system.

To log into U-boot via NOR flash, the following steps were followed:

- The AUX button was held down. Then while the AUX button was still being held down, the Power button was pressed.
- The AUX button was then released to log us into the NOR flash's U-boot loader menu.
- We could then choose the following options from the NOR flash's U-boot loader menu: Boot, Boot from MicroSD, Set console to USB, Set console to Serial, Reset and Power off

4.9 Installing QtMoko on the NeoFreeRunner

The following operating systems are currently supported by the FreeRunner: Android, Debian, GameRunner, Gentoo, Hackable:1, Mer, Mokotouch, Neovento, OpenWrt, Qalee, Qt Extended Improved, QtMoko, SHR and Slackware. We finally settled for the QtMoko Operating System due to its excellent debian package support. Since QtMoko's kernel image is unfortunately larger than 2 Megabytes, the U-Boot loader is unable to load QtMoko's kernel image into memory. To overcome U-Boot's kernel image size limitation, the Qi bootloader can be used as an alternative. The following steps were followed to install the Qi bootloader onto the FreeRunner's NAND flash memory:

- The latest qi bootloader was downloaded from the following link: http://downloads.openmoko.org/distro/
- dfu-util was installed onto the host PC (apt-get -y install dfu-util).
- We booted into NOR flash as explained above. We then run dfu-util command on the host PC to install the qi bootloader onto the Neo Frerunner's NAND memory as follows:

```
# dfu-util -a u-boot -R -D qi-s3c2442-1.0.2+gitr3b8513d8b3d9615ebda605de4bda18371aa3f359.udfu
```

To install QtMoko on the NeoFreeRunner, the following steps were followed:

- An ext3 partition was created on the microSD card.
- The latest qtmoko-debian-*.tar.gz file was downloaded and untarred onto the micro SD card. The image was downloaded from the following link: http://sourceforge.net/projects/qtmoko/files/
- The FreeRunner was rebooted so as to load the newly installed microSD qtmoko kernel image using the qi bootloader.

To ssh into the Freerunner running Qtmoko from a host PC, USB networking first had to be enabled on the host PC as follows:

```
$sudo ip address add 192.168.0.200/24 dev usb0
$sudo ip link set dev usb0 up
$sudo route add -host 192.168.0.202 dev usb0
$ssh -X root@192.168.0.202
$passwd
```

QtMoko comes with no root password configured by default. To secure the root login with a password, use the passwd command as shown above.

To give QtMoko Internet access via our host PC, the following iptables rules had to be added on our Linux host machine:

```
$sudo iptables -t nat -A PREROUTING -p tcp -s 192.168.0.202 \\
-d 192.168.0.200 --dport domain -j DNAT --to-destination gateway-ip
$sudo iptables -t nat -A PREROUTING -p udp -s 192.168.0.202 \\
-d 192.168.0.200 --dport domain -j DNAT --to-destination gateway-ip
```

QtMoko often lost its date after being rebooted. To configure the correct date for QtMoko, the following date command was used after ssh'ing into the FreeRunner (Format: MMddhhmmyyyy):

```
$ssh -X root@192.168.0.202
#date 072322462010
```

The following qtmoko debian repositories were used for the /etc/apt/sources.list file:

```
deb http://ftp.de.debian.org/debian lenny main
deb http://ftp.de.debian.org/debian lenny contrib
deb http://ftp.de.debian.org/debian lenny non-free
deb http://ftp.de.debian.org/debian unstable main
deb http://pkg-fso.alioth.debian.org/debian unstable main
# so we can get latest packages
deb http://security.debian.org/ lenny/updates main
deb http://qtmoko.meurisse.org /
# Backports.org repository
#deb http://www.backports.org/debian/ lenny-backports main contrib non-free
deb http://ftp.debian.org squeeze main contrib non-free
```

Gnuradio was installed on QTMoko using the command below:

```
#aptitude -y install \gls{gnuradio}-apps \gls{gnuradio}-companion \gls{gnuradio}-doc \gls{gnuradio}-examples \gl
```

To get GPRS working on the qtmoko, we used the following ppp options configured inside the /etc/ppp/peers/dialup946693536 file:

```
115200
lcp-echo-failure 0
lcp-echo-interval 0
novj
nobsdcomp
novjccomp
nopcomp
noaccomp
crtscts
ipcp-accept-local
noipdefault
modem
idle 120
usepeerdns
defaultroute
connect-delay 5
noauth
persist
```

To list all the WLAN beacons currently available to the FreeRunner, click on the Q icon on the QtMoko, navigate to Applications, Terminal and run the following command:

```
#iwlist eth0 scanning
```

Chapter 5

RFX2400 Description

"Let the future tell the truth, and evaluate each one according to his work and accomplishments. The present is theirs; the future, for which I have really worked, is mine." - Nikola Tesla, , who was considered a mad scientist by many during his time, but who is now widely acknowledged as being the father of wireless communications.

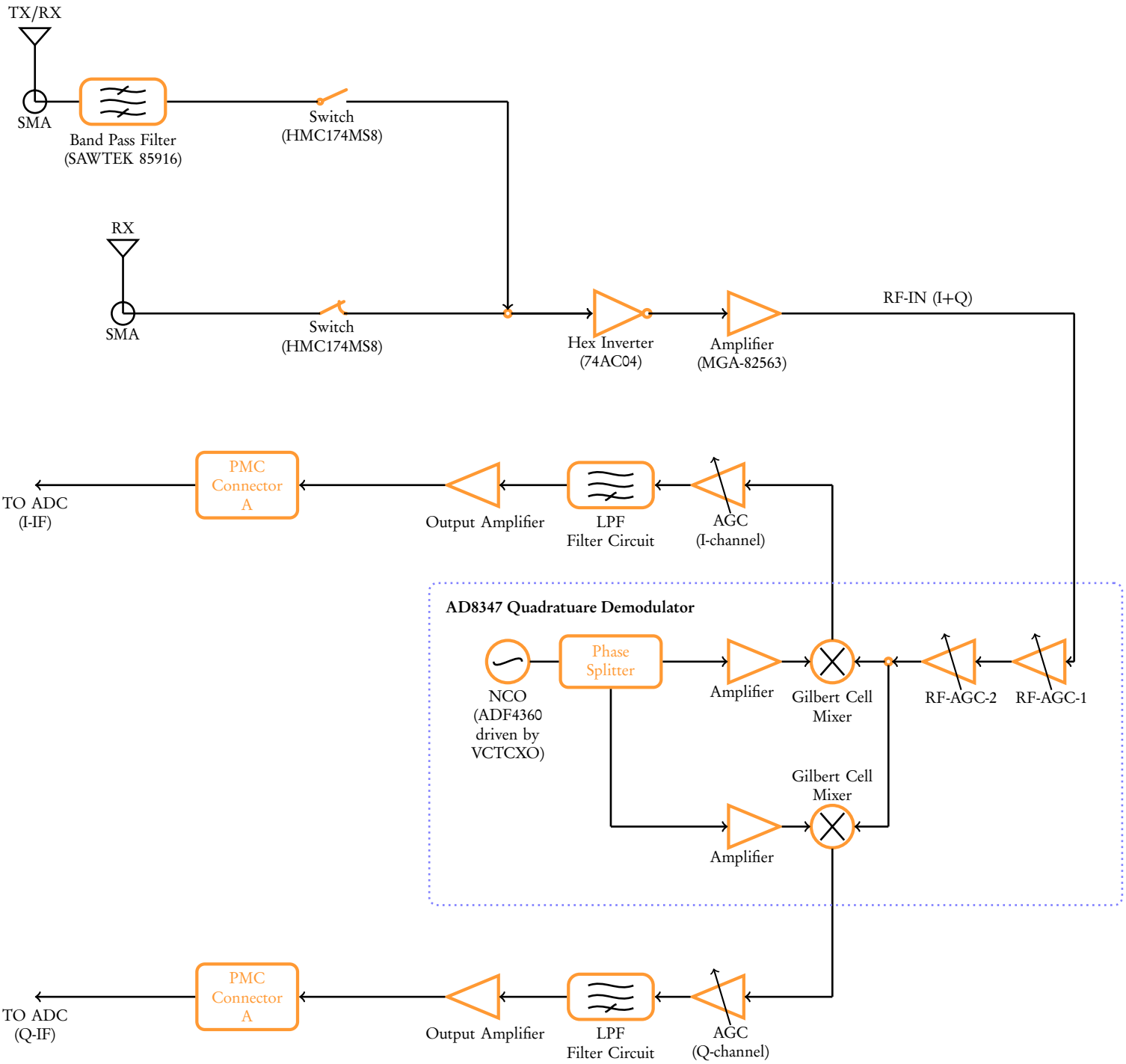
5.1 Overview

The 802.11g WLAN channels fall within the frequency range of approximately 2.4 GHz and 2.5 GHz. Since the RFX2400 USRP2 RF daughter board is able to transmit and receive at between 2.25GHz and 2.9GHz, this daughter board was to be used for the WLAN OFDM symbol transmission. The Mixed Signal Circuit that was to be designed to transmit the OFDM symbols coming from the Rhino or USRP2 FPGA board would also be using this RF Acquisition and Transmission daughter board. It was therefore imperative to understand every aspect of the board's functionality, as summarized in this chapter.

5.2 RFX2400 Receive Chain Block Diagram

A block diagram of the RFX2400 is shown on the next page:

Figure 5.1: RFX2400 Daughter Board Receive Chain Block Diagram



5.3 The RFX2400 Receive Chain Components

- **SMA**: The surface mount antenna used on the RFX2400 should resonate at a frequency of between 2.25GHz and 2.9GHz.
- **SAWTEK 85916**: This is a surface mountable, hermetically sealed band pass filter that attenuates or filters all frequencies outside of the 2.25GHz to 2.9GHz range. It has a 50 Ωmatched input impedance.
- **2 HMC174MS8** Switches: There are two such switches on the RFX2400 Board. The first switch is on the TX/RX line and is used to switch this line between receive and transmit mode. If both SMA antennas are being used as receive channels, the second switch is used to switch between two receive channels ie to switch between using the TX/RX channel only and the and RX channel only (Unlike some models of the gnuradio daughter boards, the RFX2400 though does not receive signals from both TX/RX and RX antennas simultaneously). The HMC174MS8 switch has a single positive supply of between +3V and +10V. The switches work best at a frequency of between 900MHz and 1.8GHz to 2.2GHz. Isolation loss is between 17dB and 21dB. The rise/fall switching latency is 10ns, while the on/off switching latency is 24ns.
- **National Semiconductor 74AC04 Hex Inverter**: The Inverter is used to switch the HMC174MS8 switch between the TX/RX and RX channels, depending on the input received (normally from the FPGA via the PMC Connector). It has a supply voltage of between -0.5V and +7V.
- **MGA-82563 Amplifier**: This surface mountable amplifier is able to work at a range of between 0.1 to 6 GHz. For an input RF frequency of 2 GHz for example, it's able to provide a gain of upto 13.2 dB, with a 2.2 dB noise figure. It has a single +3V supply voltage.
- **AD8347 Quadrature Demodulator**: The AD8347 has one positive and one negative RF input. The demodulator has a 90MHz demodulation bandwidth and outputs an IF signal that is between approximately 10MHz and 100MHz. The optimum baseband IF output frequency is 64MHz. It has a single supply voltage of between 2.7V and 5.5V. The RF inputs to the AD8347 go through the following path while inside the quadrature demodulator:

  RF Amplifier 1: The RF signal (I and Q are still combined at this stage) is taken through the first automatic gain controller. The RF amplifier 1 gain can normally be controlled from the FPGA.

  RF Amplifier 2: The RF signal output from RF-amplifier 1 (I and Q are still combined) is taken through the second automatic gain controller. This gain can also be controlled from the FPGA through inputs to the AD8347's gain control input section. Like the RF-amplifier 1, the RF amplifier 2 Gain is normally controllable from the FPGA through inputs to the AD8347's gain control input section.

  Quadrature Gilbert-cell mixer 1: The output from RF amplifier 2 is split into two channels. The first output channel is taken to the I signal mixer. This mixer has a second I (in-phase) IF signal input from the AD8347 Phase Splitter.

Quadrature Gilbert-cell mixer 2: The second output from RF amplifier 2 is taken to the Q (quadrature) signal mixer. This mixer has a second Q (quadrature-phase) IF signal input from the AD8347 Phase Splitter.

Phase splitter: The phase splitter is driven by a Numerically Controlled Oscillator (NCO) in order to generate the in phase and quadrature phase IF signal outputs. The NCO, is in turn driven by a Voltage Controlled Temperature Compensated Crystal Oscillator (VCTCXO). The Phase Splitter's IF I and Q signal outputs are at an optimum frequency of **64MHz**.

AD4360-0 Voltage Controlled Oscillator: This has a single supply voltage of between 3.0V and 3.5V. It has a frequency divider (ie divide by 2, 4, 6 or 8) which is programmable from the FPGA. RF output from the Oscillator is **tunable** between 840 MHz and 900 MHz. It is able to detect and lock both digital and analog oscillator signals (The RF Output is held back until the PLL detects and locks to the desired oscillator output frequency). The output frequency is between 2.4 GHz and 2.725 GHz. RF Output power of between -6.5 dBm and -13 dBm is also programmable from the FPGA. The AD4360-0 is driven by a Voltage Controlled Temperature Compensated Crystal Oscillator (VCTCXO).

IF I-signal Amplifier 1: This amplifies the in-phase IF frequency output from the phase splitter.

IF Q-signal Amplifier 2: This amplifies the quadrature-phase IF frequency output from the phase splitter.

After the I and Q signals have passed through their respective mixers, the original RF input signal will have been quadrature demodulated into separate I and Q signals.

IF I-signal AGC 1: After mixing, the IF I-signal will go through an automatic gain controller (whose gain is also controlled from the FPGA).

IF Q-signal AGC 2: After mixing, the IF Q-signal will go through an automatic gain controller (whose gain is also controlled from the FPGA).

The I or Q signal gain after the first 2 RF amplifiers and a single mixer with input from a phase splitter amplifier is between -13dB and +1dB.

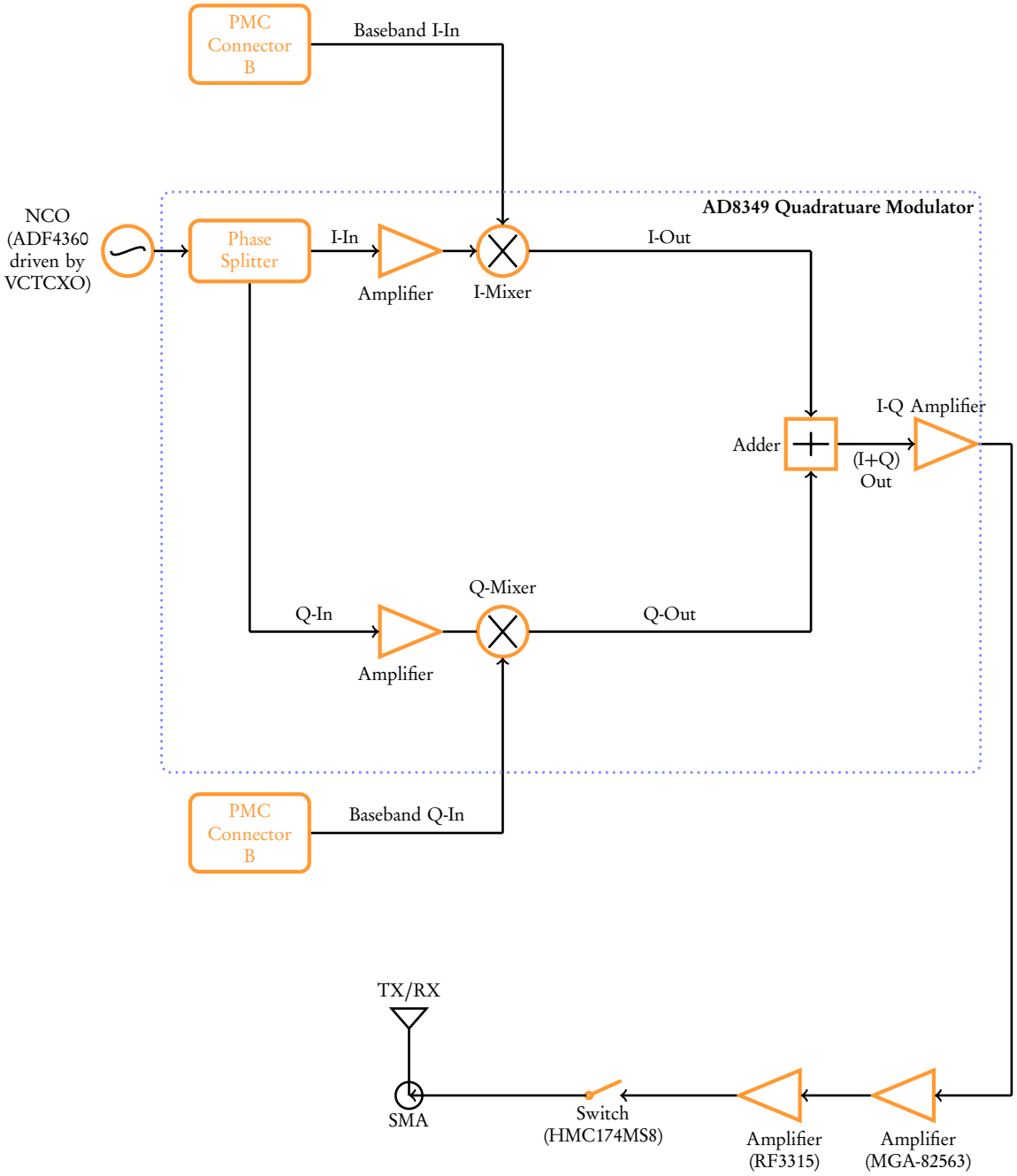
The I or Q signal output by the AD8347, after going through the 2 RF amplifiers, mixer with input from the phase splitter amplifier and a third automatic gain controlled amplifier can altogether provide a gain of between -30 dB to + 39.5 dB.

- **Low Pass Filter**: The RFX 2400 contains circuitry to attenuate all frequencies below the allowed IF clock frequency.
- **Amplifier**: The IF frequency is taken through some RFX2400 amplification circuitry, before finally being taken to the Rhino Expansion Board or USRP2 FPGA board via a PMC Connector. Both the I and Q signals are transmissited on LVDS lines (4 lines go into the PMC connector).
- **PMC Connector**: The I-signal pair (positive and negative) and Q-signal pair are taken to the Rhino expansion Board or USRP2 FPGA board via the PMC connector.

5.4 RFX2400 Transmit Chain Block Diagram

The RFX2400 Transmit Chain Block diagram is shown on the next page:

Figure 5.2: RFX 2400 Daughter Board Transmit Chain Block Diagram



5.5 The RFX2400 Transmit Chain Components

- **PMC Connector**: The Baseband I-signal pair (positive and negative) and Q-signal pair come into the RFX2400 from the Rhino expansion board or USRP2 FPGA board via the PMC Connector.

- **AD8349 Quadrature Modulator**: The AD8349 has a frequency range of between 800 MHz and 2.7 GHz. It can take in baseband input frequencies of between 20 MHz and 200 MHz. It has a modulation bandwidth of 200 MHz and typical output power level of 2.1 dBm at 2.1GHz. It has single power supply of between 4.5V and 5.5V, and a 50 Ωoutput matched impedance. The AD8349 takes the I and Q signal baseband inputs from the Rhino expansion board through the following path while inside the quadrature modulator:

  I-Mixer 1: The mixer takes in two inputs; an RF I-signal from the phase splitter and the baseband I-signal coming in from the Rhino expansion board or USRP2 FPGA board via the PMC connector.

  Phase Splitter: The Phase Splitter is driven by a Numerically Controlled Oscillator (NCO) in order to generate the in phase and quadrature phase RF signal outputs. The NCO, is in turn driven by a Voltage Controlled Temperature Compensated Crystal Oscillator (VCTCXO).

  Phase Splitter Amplifier 1: This amplifies the phase splitter's I-signal before it goes into I-Mixer 1.

  AD4360-0 Voltage Controlled Oscillator: This has a single supply voltage of between 3.0V and 3.5V. It has a frequency divider (ie divide by 2, 4, 6 or 8) which is programmable from the FPGA. RF output from the oscillator is **tunable** between 840 MHz and 900 MHz. It is able to detect and lock both digital and analog oscillator signals (The RF Output is held back until the PLL detects and locks to the desired oscillator output frequency). The output frequency is between 2.4 GHz and 2.725 GHz. RF Output power of between -6.5 dBm and -13 dBm is also programmable from the FPGA. The AD4360-0 is driven by a Voltage Controlled Temperature Compensated Crystal Oscillator (VCTCXO).

  Q-Mixer 2: The mixer takes in two inputs; an RF Q-signal from the phase splitter and the baseband Q-signal coming in from the Rhino expansion board or USRP2 FPGA board via the PMC Connector.

  Phase Splitter Amplifier 2: This amplifies the Phase splitter's Q-signal before it goes into Q-Mixer 2.

  Adder: This combines the separate I and Q RF signals into one. The output of the Adder is now a Quadrature Modulated RF signal.

  I-Q Amplifier: The I-Q amplifier amplifies the quadrature modulated RF signal before it gets output by the AD8349.

- **MGA-82563 Amplifier**: This surface mountable amplifier is able to work at a range of between 0.1 to 6 GHz. For an input RF frequency of 2 GHz for example, it is able to provide a gain of upto 13.2 dB, with a 2.2 dB noise figure. It has a single +3V supply voltage.

- **RF3315 Amplifier**: This RF signal amplifier has a frequency range of between 300 MHz and 3 GHz, a single 5V power supply and a typical +12dBm gain at 2.0 GHz. Typical RF input power is at +20 dBm.
- **2 HMC174MS8** Switch: When this switch is activated, the TX/RX channel is in transmit mode, otherwise it's in receive mode.
- **SMA**: An antenna that resonates at a frequency of between 2.25 GHz and 2.9 GHz is connected to the TX/RX SMA.

5.6 Troubleshooting the RFX2400 daughter board and the resulting modifications

We initially had problems sending or receiving OFDM waveforms using the RFX2400 daughter board. To troubleshoot USRP2-RFX2400 receive, we used the HP 8350B Sweep Oscillator/Signal Generator to transmit a single tone 2.4GHz, 10 dBm RF signal. Though the signal was correctly detected by the Agilent E4407 Spectrum Analyzer, we could detect nothing coming off the USRP2ś RFX2400 board while using Gnuradioś usrp2_fft.py script to observe the waveform being transmitted by the sweep oscillator. The command that was used to call the script is shown below:

```
#./usrp2_fft.py --interface=eth0 --freq=2.4e9
```

Listing 5.1: A command

A diagram showing the incorrect single tone signal being received from the signal generator on the USRP2's RFX2400 board is shown below:

To troubleshoot the USRP2-RFX2400 transmit chain, the usrp2_siggen.py script was used. The usrp2_siggen.py command that was used is shown below:

```
#./usrp2_siggen.py -f 2.4e9 -e eth0
```

Listing 5.2: A command

The supposed single tone signal being transmitted by the RFX2400-USRP2 board was not being picked up by the Agilent E4407 Spectrum Analyzer. We spent over three weeks trying to troubleshoot both the host PC Gnuradio code and the USRP2 Spartan 3 FPGA code but we still could not understand why transmit or receive was not working. We scoured the Gnuradio mailing lists for days on end, but no one seemed to be experiencing the same problems. This was one of the most puzzling and unexpected system outputs we had come across in several years.

We eventually began making progress during the fourth week when we installed the more recent UHD FPGA code from Gnuradio. The txrx_uhd_20100706.bin and u2_rev3_uhd_20100706.bin images were

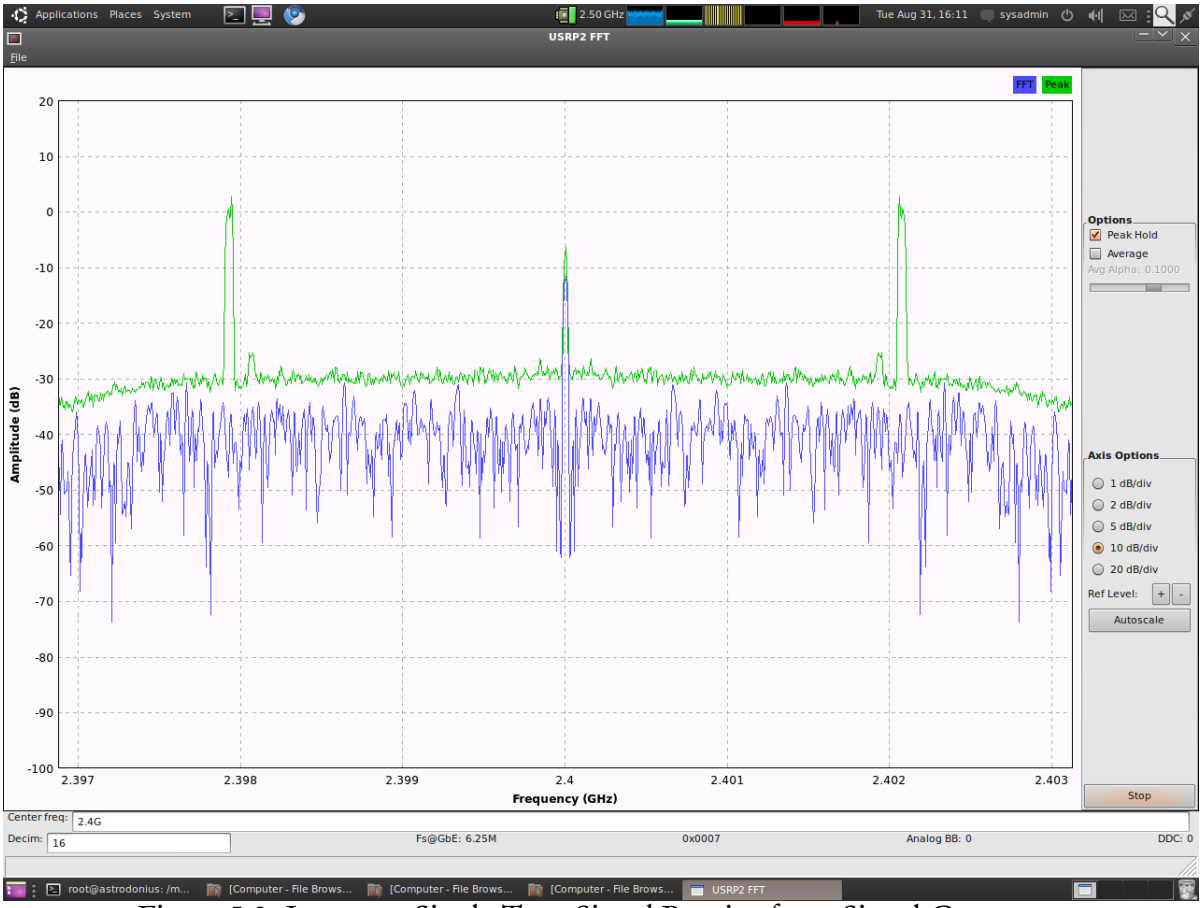


Figure 5.3: Incorrect Single Tone Signal Receive from Signal Generator

used. Oddly enough, our daughter board was not being recognized by the new Gnuradio host PC code so that none of the test scripts provided could run successfully. To get example Gnuradio scripts to run, the USRP2_FPGA_COMPAT_NUM define statement had to be changed from 0 to 5 and the USRP2_FW_COMPAT_NUM changed from 1 and 6 inside host/lib/usrp/usrp2/fw_common.h. Once we were able to successfully run the host PC test scripts, we were able to confirm without a doubt that our daughter board was not being recognized by the USRP2 FPGA code:

```
root@astrodonius:git-uhd/host/examples# ./benchmark_rx_rate

Creating the usrp device with: ...
Target recv sock buff size: 50000000 bytes
Actual recv sock buff size: 131071 bytes

Warning:
   The recv buffer is smaller than the requested size.
   The minimum recommended buffer size is 50000000 bytes.
   See the USRP2 application notes on buffer resizing.

Warning: unknown dboard-id or dboard-id combination: unknown (0x0007)
   -> defaulting to the unknown board type
Warning: unknown dboard-id or dboard-id combination: unknown (0x000b)
   -> defaulting to the unknown board type
RX samples per packet: 362
TX samples per packet: 363
Recv pirate num frames: 89
Using Device: Simple USRP:
 Device: usrp2 device
 Mboard: usrp2 mboard0 - rev 4:0
```

```
 RX DSP: usrp2 ddc0
 RX Dboard: usrp2 dboard (rx unit)
 RX Subdev: Unknown - unknown (0x0007)
 TX DSP: usrp2 duc0
 TX Dboard: usrp2 dboard (tx unit)
 TX Subdev: Unknown - unknown (0x000b)

OTesting receive rate 0.500000 Msps (10.000000 second run)

   Received packets: 13813
   Received samples: 5000306
   Lost samples: 0
   Lost packets: 0 (approximate)
   Sustained receive rate: 0.500000 Msps


Testing receive rate 1.000000 Msps (10.000000 second run)

   Received packets: 27625
   Received samples: 10000250
   Lost samples: 0
   Lost packets: 0 (approximate)
   Sustained receive rate: 1.000000 Msps


Testing receive rate 2.000000 Msps (10.000000 second run)

   Received packets: 55249
   Received samples: 20000138
   Lost samples: 0
   Lost packets: 0 (approximate)
   Sustained receive rate: 2.000000 Msps


Testing receive rate 4.000000 Msps (10.000000 second run)

   Received packets: 110498
   Received samples: 40000276
   Lost samples: 0
   Lost packets: 0 (approximate)
   Sustained receive rate: 4.000000 Msps


Testing receive rate 8.333333 Msps (10.000000 second run)

   Received packets: 230203
   Received samples: 83333486
   Lost samples: 0
   Lost packets: 0 (approximate)
   Sustained receive rate: 8.333333 Msps


Testing receive rate 16.666667 Msps (10.000000 second run)

   Received packets: 460190
   Received samples: 166588780
   Lost samples: 78192
   Lost packets: 216 (approximate)
   Sustained receive rate: 16.658847 Msps
```

```
Testing receive rate 25.000000 Msps (10.000000 second run)
./lib/usrp/usrp2/fw_common.h
   Received packets: 683928
   Received samples: 247581936
   Lost samples: 2418160
   Lost packets: 6680 (approximate)
   Sustained receive rate: 24.758184 Msps
Done!

root@git-uhd/host/examples# ./rx_timed_samples

Creating the usrp device with: ...
Target recv sock buff size: 50000000 bytes
Actual recv sock buff size: 131071 bytes

Warning:
   The recv buffer is smaller than the requested size.
   The minimum recommended buffer size is 50000000 bytes.
   See the USRP2 application notes on buffer resizing.

Warning: unknown dboard-id or dboard-id combination: unknown (0x0007)
   -> defaulting to the unknown board type
Warning: unknown dboard-id or dboard-id combination: unknown (0x000b)
   -> defaulting to the unknown board type
RX samples per packet: 362
TX samples per packet: 363
Recv pirate num frames: 89
Using Device: Simple USRP:
 Device: usrp2 device
 Mboard: usrp2 mboard0 - rev 4:0
 RX DSP: usrp2 ddc0
 RX Dboard: usrp2 dboard (rx unit)
 RX Subdev: Unknown - unknown (0x0007)
 TX DSP: usrp2 duc0
 TX Dboard: usrp2 dboard (tx unit)
 TX Subdev: Unknown - unknown (0x000b)

Setting RX Rate: 6.250000 Msps...
Actual RX Rate: 6.250000 Msps...
Setting device timestamp to 0...

Begin streaming 1000 samples, 3 seconds in the future...
OGot packet: 362 samples, 3 full secs, 0.000000 frac secs
Got packet: 362 samples, 3 full secs, 0.000058 frac secs
Got packet: 276 samples, 3 full secs, 0.000116 frac secs

Done!

root@astrodonius:git-uhd/host/examples# ./tx_timed_samples

Creating the usrp device with: ...
Target recv sock buff size: 50000000 bytes
Actual recv sock buff size: 131071 bytes

Warning:
   The recv buffer is smaller than the requested size.
   The minimum recommended buffer size is 50000000 bytes.
   See the USRP2 application notes on buffer resizing.

Warning: unknown dboard-id or dboard-id combination: unknown (0x0007)
   -> defaulting to the unknown board type
```

```
Warning: unknown dboard-id or dboard-id combination: unknown (0x000b)
   -> defaulting to the unknown board type
RX samples per packet: 362
TX samples per packet: 363
Recv pirate num frames: 89
Using Device: Simple USRP:
 Device: usrp2 device
 Mboard: usrp2 mboard0 - rev 4:0
 RX DSP: usrp2 ddc0
 RX Dboard: usrp2 dboard (rx unit)
 RX Subdev: Unknown - unknown (0x0007)
 TX DSP: usrp2 duc0
 TX Dboard: usrp2 dboard (tx unit)
 TX Subdev: Unknown - unknown (0x000b)

Setting TX Rate: 6.250000 Msps...
Actual TX Rate: 6.250000 Msps...
Setting device timestamp to 0...

Sent 1000 samples

Done!

root@astrodonius:git-uhd/host/examples# ./tx_waveforms

Creating the usrp device with: ...
Target recv sock buff size: 50000000 bytes
Actual recv sock buff size: 131071 bytes

Warning:
   The recv buffer is smaller than the requested size.
   The minimum recommended buffer size is 50000000 bytes.
   See the USRP2 application notes on buffer resizing.

Warning: unknown dboard-id or dboard-id combination: unknown (0x0007)
   -> defaulting to the unknown board type
Warning: unknown dboard-id or dboard-id combination: unknown (0x000b)
   -> defaulting to the unknown board type
RX samples per packet: 362
TX samples per packet: 363
Recv pirate num frames: 89
Using Device: Simple USRP:
 Device: usrp2 device
 Mboard: usrp2 mboard0 - rev 4:0
 RX DSP: usrp2 ddc0
 RX Dboard: usrp2 dboard (rx unit)
 RX Subdev: Unknown - unknown (0x0007)
 TX DSP: usrp2 duc0
 TX Dboard: usrp2 dboard (tx unit)
 TX Subdev: Unknown - unknown (0x000b)

Setting TX Rate: 6.250000 Msps...
Actual TX Rate: 6.250000 Msps...

Setting TX Freq: 0.000000 Mhz...
Actual TX Freq: 0.000000 Mhz...

Setting TX Gain: 0.000000 dB...
Actual TX Gain: 0.000000 dB...

Done!
```

Listing 5.3: A command

After banging our heads against the table for hours on end, we finally found the two root sources of our transmit and receive problems. To solve the first problem, the USRP2 FPGA code needed to be updated so it would recognize RFX2400 boards made before the 12th of December, 2006. Due to fate or pure bad luck, the UCT RFX2400 board had been bought only six months before this cut-off date, the 2nd of June, 2006. The reason no one else on the gnuradio mailing lists seemed to be experiencing the same problems was because most other USRP2 users were using much newer RFX2400 boards. The command that was used to burn the correct device id to the RFX2400 board is shown below:

```
#\gls{usrp}/host/apps/burn-db-eeprom -A -t rfx2400_mimo_b --force
```

Listing 5.4: A command

To solve the second problem, we had to resolder the jumpers on the RFX2400 board so that the 100 MHz USRP2 clock was used to clock the RFX2400 board, and not the RFX2400's on internal 64 MHz clock. Using the 64 MHz internal RFX2400 clock would have worked for the first generation USRP FPGA board which also had an internal 64 MHz clock, but this would definitely not have worked while using the USRP2 due to synchronization problems that would prevent the PLL from ever locking onto a given frequency. To disable the RFX2400 board internal clock during receive, a zero ohm resistor or jumper had to be moved from refdes number 64 to 84. To disable the RFX2400 board internal clock during transmit, a jumper had to be moved from refdes number 142 to 153. To connect the RFX2400 board to the USRP2 internal clock for receive, a jumper had to be moved from 35 to 36. To connect the RFX2400 board to the USRP2 internal clock for transmit, a jumper had to be moved from 117 to 115.

After making these four modifications, we were finally able to successfully send and receive single tone single tone signals using the RFX2400-USRP2 boards. A 2.4GHz single tone signal was correctly received from the RFX2400-USRP2 board as shown in the diagram below:

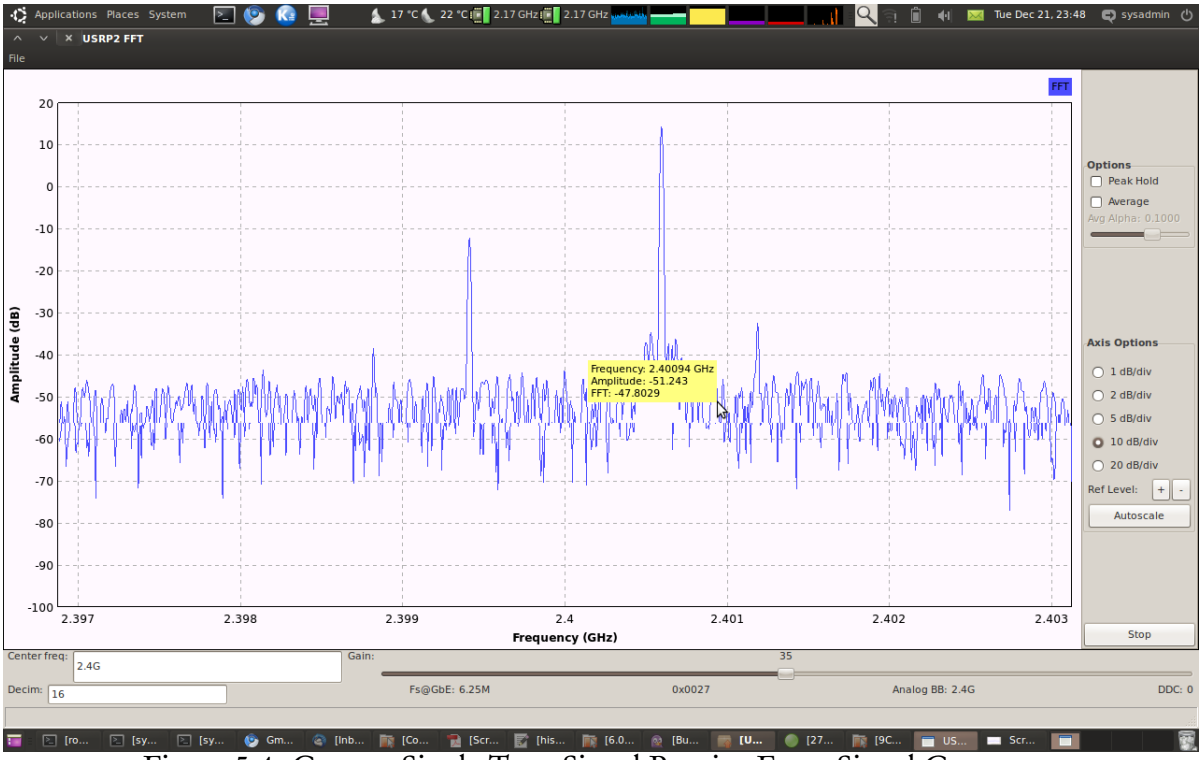


Figure 5.4: Correct Single Tone Signal Receive From Signal Generator

Chapter 6

SDR OFDM Implementation using Gnuradio

"Let the future tell the truth, and evaluate each one according to his work and accomplishments. The present is theirs; the future, for which I have really worked, is mine." - Nikola Tesla, the inventor of wireless communications and AC current transmission.

6.1 Overview

This chapter describes the software defined OFDM implementation on Gnuradio. The chapter will often refer to the gr_wlan code inside the DVD which accompanies this thesis. The gr_wlan code wraps and refactors the Gnuradio OFDM code from ftw https://www.cgran.org/wiki/ftw80211ofdmtx. The code verbatim quoted below is written in python. It however makes use of underlying C++ signal processing blocks by using SWIG (http://www.swig.org).

6.2 A Description of the WLAN 802.11 Physical Layer

The 802.11g specification prescribes the use of 64 subcarriers, which includes 48 data subcarriers and 4 pilot subcarriers. The remaining 12 subcarriers will from the guard intervals for the OFDM transmission. An additional empty subcarrier is also added in the middle (where the DC component would be). During the initial training sequence that is used to help the 802.11 receiver chip detect the incoming OFDM signal, the empty DC subcarrier is used by the chip to help it determine what center frequency it should try and tune to. The 802.11g specification prescribes the use of 14 possible channels with the following center frequencies: 2.412 GHz (Channel 1), 2.417 GHz (Channel 2), 2.422 GHz (Channel 3), 2.427 GHz (Channel 4), 2.432 GHz (Channel 5), 2.437 GHz (Channel 6), 2.442 GHz (Channel 7), 2.447 GHz (Channel 8), 2.452 GHz (Channel 9), 2.457 GHz (Channel 10), 2.462 GHz (Channel 11), 2.467 GHz (Channel 12), 2.472 GHz (Channel 13), 2.484 GHz (Channel 14).

Definitions:
N_{DSC} = Number of data subcarriers per OFDM symbol = 48
N_{CBPSC} = Number of coded bits per OFDM subcarrier
N_{CBPS} = Number of coded bits per OFDM symbol = $N_{CBPSC} \times N_{DSC}$
R = Coding Rate
N_{DBPSC} = Number of data bits per OFDM subcarrier = $N_{CBPSC} \times R$
N_{DBPS} = Number of data bits per OFDM symbol = $N_{CBPS} \times R$

For a 20 MHz channel:
ΔF = Subcarrier frequency spacing = $20MHz/64 = 0.3125MHz$
T_{FFT} = OFDM subcarrier duration or period = $1/\Delta F = 3.2\mu secs$
T_{GI} = OFDM guard interval between each transmitted subcarrier = $T_{FFT}/4 = 0.8\mu secs$
T_{SYM} = OFDM symbol duration or period = $T_{FFT} + T_{GI} = 4.0\mu secs$
N_{SYM} = Number of OFDM symbols per second = $1/T_{FFT} = 0.25 \times 10^6$

CR = Code bit rate per second = $N_{DSC} \times N_{CBPSC} \times N_{SYM}$
DR = Data bit rate per second = $N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = N_{DSC} \times N_{DBPSC} \times N_{SYM}$

6.2.1 BPSK

The 802.11 specification prescribes that the first 802.11g PLCP OFDM symbol which consists of the rate bits(4), reserved bit(1), length bits (12), parity bit(1) and tail bits(6), be encoded using the BPSK format. For the BPSK modulation format, bits are encoded onto the in-phase(I) and quadrature phase (Q) component of each subcarrier. The I and Q components of the subcarrier frequency are 180° out of phase with each other, and have the same amplitude.The two phases can be represented by the following 2 complex constellation points in space: $-1+0j$ and $+1+0j$.

While using BPSK, 2 possible constellation points are possible for each subcarrier, with one point representing bit 0 and the other bit 1. This means that only 1 bit at a time can be coded for the BPSK modulation format. Since there are 48 data subcarriers, then it is possible to code 48 bits per OFDM symbol. BPSK can also be used to modulate the 802.11g data bits. If the 802.11g OFDM data bits are also encoded using the BPSK phase shift keying modulation format with a data coding rate of 3/4, then the maximum data rate achievable for a 20 MHz bandwidth channel is 9 Mbits/second (There are 36 data bits per OFDM symbol). When a data coding rate of 1/2 is used, the maximum data rate achievable for a 20 MHz bandwidth channel while using the BPSK modulation format is 6 Mbits/second. The non data bits that form part of the OFDM symbol are used for the error correction.[13, Pg 597]

Where $R = 1/2$,

$$DR = N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = 48 \times 1 \times 1/2 \times (0.25 \times 10^6) = 6Mbits/sec$$

Where $R = 3/4$,

$$DR = N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = 48 \times 1 \times 3/4 \times (0.25 \times 10^6) = 9Mbits/sec$$

6.2.2 QPSK

The QPSK modulation format allows bits to be encoded onto 4 different complex constellation points in space. The following complex points in space were used during QPSK modulation: $-0.7071 - 0.7071j, -0.7071 + 0.7071j, +0.7071 - 0.7071j$ and $+0.7071 + 0.7071j$. Each of these four complex constellation points are 90° out of phase with each other. Since four points in space are available to represent either bit 0 or bit 1, it is possible to encode 2 bits at a time during each frequency cycle. Since there are 48 data subcarriers, the maximum number of coded bits per OFDM symbol while using the QPSK modulation mode is 96. Thus, for a data coding rate of 3/4, the maximum data rate achievable for a 20 MHz bandwidth channel is 18 Mbits/second and for a data coding rate of 1/2, the maximum data rate achievable for a 20 MHz bandwidth is 12 Mbits/second.

Where $R = 1/2$,

$$DR = N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = 48 \times 2 \times 1/2 \times (0.25 \times 10^6) = 12Mbits/sec$$

Where $R = 3/4$,

$$DR = N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = 48 \times 2 \times 3/4 \times (0.25 \times 10^6) = 18Mbits/sec$$

6.2.3 16-QAM

The 16-QAM modulation format allows bits to be encoded onto 16 different complex constellation points in space. The following 16 complex points in space were used during 16-QAM modulation: $-0.9487 - 0.9487j, -0.9487 - 0.3162j, -0.9487 + 0.9487j, -0.9487 + 0.3162j, -0.3162 - 0.9487j, -0.3162 - 0.3162j, -0.3162 + 0.9487j, -0.3162 + 0.3162j, 0.9487 - 0.9487j, 0.9487 - 0.3162j, 0.9487 + 0.9487j, 0.9487 + 0.3162j, 0.3162 - 0.9487j, 0.3162 - 0.3162j, 0.3162 + 0.9487j$ and $0.3162 + 0.3162j$. To avoid interference, points with the same phase have different amplitudes, while points with the same amplitude have a different phase. Since 16 points in space are available to represent either bit 0 or bit 1, it is possible to encode 4 bits at a time during each frequency cycle. Since there are 48 data subcarriers, the maximum number of coded bits per OFDM symbol while using the 16-QAM modulation mode is 192. Thus, for a data coding rate of 3/4,

the maximum data rate achievable for a 20 MHz bandwidth channel is 32 Mbits/second and for a data coding rate of 1/2, the maximum data rate achievable for a 20 MHz bandwidth is 24 Mbits/second.

Where $R = 1/2$,

$$DR = N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = 48 \times 4 \times 1/2 \times (0.25 \times 10^6) = 24 Mbits/sec$$

Where $R = 3/4$,

$$DR = N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = 48 \times 4 \times 3/4 \times (0.25 \times 10^6) = 32 Mbits/sec$$

6.2.4 64-QAM

The 64-QAM modulation format allows bits to be encoded onto 64 different complex constellation points in space. The following 64 complex points in space were used during 64-QAM modulation: $-1.0801 - 1.0801j, -1.0801 - 0.7715j, -1.0801 - 0.1543j, -1.0801 - 0.4629j, -1.0801 + 1.0801j, -1.0801 + 0.7715j, -1.0801 + 0.1543j, -1.0801 + 0.4629j, -0.7715 - 1.0801j, -0.7715 - 0.7715j, -0.7715 - 0.1543j, -0.7715 - 0.4629j, -0.7715 + 1.0801j, -0.7715 + 0.7715j, -0.7715 + 0.1543j, -0.7715 + 0.4629j, -0.1543 - 1.0801j, -0.1543 - 0.7715j, -0.1543 - 0.1543j, -0.1543 - 0.4629j, -0.1543 + 1.0801j, -0.1543 + 0.7715j, -0.1543 + 0.1543j, -0.1543 + 0.4629j, -0.4629 - 1.0801j, -0.4629 - 0.7715j, -0.4629 - 0.1543j, -0.4629 - 0.4629j, -0.4629 + 1.0801j, -0.4629 + 0.7715j, -0.4629 + 0.1543j, -0.4629 + 0.4629j, 1.0801 - 1.0801j, 1.0801 - 0.7715j, 1.0801 - 0.1543j, 1.0801 - 0.4629j, 1.0801 + 1.0801j, 1.0801 + 0.7715j, 1.0801 + 0.1543j, 1.0801 + 0.4629j, 0.7715 - 1.0801j, 0.7715 - 0.7715j, 0.7715 - 0.1543j, 0.7715 - 0.4629j, 0.7715 + 1.0801j, 0.7715 + 0.7715j, 0.7715 + 0.1543j, 0.7715 + 0.4629j, 0.1543 - 1.0801j, 0.1543 - 0.7715j, 0.1543 - 0.1543j, 0.1543 - 0.4629j, 0.1543 + 1.0801j, 0.1543 + 0.7715j, 0.1543 + 0.1543j, 0.1543 + 0.4629j, 0.4629 - 1.0801j, 0.4629 - 0.7715j, 0.4629 - 0.1543j, 0.4629 - 0.4629j, 0.4629 + 1.0801j, 0.4629 + 0.7715j, 0.4629 + 0.1543j$ and $0.4629 + 0.4629j$. To avoid interference, points with the same phase have different amplitudes, while points with the same amplitude have a different phase. Since 64 points in space are available to represent either bit 0 or bit 1, it is possible to encode 6 bits at a time during each frequency cycle. Since there are 48 data subcarriers, the maximum number of coded bits per OFDM symbol while using the 64-QAM modulation mode is 288. Thus, for a data coding rate of 3/4, the maximum data rate achievable for a 20 MHz bandwidth channel is 54 Mbits/second and for a data coding rate of 2/3, the maximum data rate achievable for a 20 MHz bandwidth is 48 Mbits/second.

Where $R = 2/3$,

$$DR = N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = 48 \times 6 \times 2/3 \times (0.25 \times 10^6) = 48 Mbits/sec$$

Where $R = 3/4$,

$$DR = N_{DSC} \times N_{CBPSC} \times R \times N_{SYM} = 48 \times 8 \times 3/4 \times (0.25 \times 10^6) = 54 Mbits/sec$$

6.2.5 The Fast Fourier Transform

The 802.11 specification uses the IFFT and FFT to represent time domain and frequency domain signals respectively. The FFT makes use of a Fourier Series solution for a repeating periodic waveform, to represent a time domain waveform in the frequency domain. The Fourier series does this by making use of the fact that any complex waveform in space can be represented as a combination of specific constituent sine and cosine waveforms, each of which has it's own specific frequency and amplitude. Each sine or cosine waveform is represented as a basis function with a specific fourier series coeffecient, where the fourier series coefficient represents the variation in amplitude of the sine or cosine basis function waveform.

Fourier Equation Theory Definitions:
$f(t)$ = waveform in the time domain.
c_0 = dc offset of the waveform. The dc offset is used to shift the central point of the waveform up or down the given amplitude scale.
a_n = fourier coefficient for the sine component of the waveform.
b_n = fourier coefficient for the cosine component of the waveform.
c_n = fourier coefficient for the basis function of the waveform represented using only cosine components.
n = a waveform harmonic which is an integer multiple of the fundamental waveform frequency or resolution frequency, where $n = 0, 1, 2, \ldots (N-1)$.
ϕ_n = phase shift of sine or cosine waveform component = $\tan^{-1}(\frac{b_n}{a_n})$.
f = fundamental waveform frequency or resolution frequency = $\frac{1}{T}$
N = the total number of harmonics of the waveform.
ω = radial frequency = $2\pi f$
f = linear frequency = $\frac{\omega}{2\pi}$

DFT and FFT Theory Definitions:
n = a waveform harmonic which is an integer multiple of the fundamental waveform frequency or resolution frequency
N = total number of samples taken for the waveform in the case of IDFT and IFFT. In the case of DFT and FFT, it is the total number of harmonics.
k = a waveform sample which is an integer multiple.
τ = is the time difference between each sample point. $t = k\tau$
$f(n)$ = waveform in time domain represented as a function of k $F(k)$ frequency domain sample points, where $k = 0, 1, 2, \ldots (N-1)$.
$F(k)$ = waveform in frequency domain represented as a function of n $f(n)$ time domain harmonics, where $n = 0, 1, 2, \ldots (N-1)$.

The Fourier Series Explained:

The Fourier Series equation for any time domain waveform in space is given as:

$$\begin{aligned} f(t) &= c_0 + \sum_{n=1}^{N} a_n sin(n\omega t) + \sum_{n=1}^{N} b_n cos(n\omega t)\text{ , where} \\ c_0 &= \frac{1}{T}\int_0^T f(t)dt \\ a_n &= \frac{2}{T}\int_0^T f(t)sin(n\omega t)dt \\ b_n &= \frac{2}{T}\int_0^T f(t)cos(n\omega t)dt \end{aligned} \tag{6.1}$$

The same equation can be represented as follows using the linear frequency instead of the radial frequency:

$$f(t) = c_0 + \sum_{n=1}^{N} a_n sin(2\pi f nt) + \sum_{n=1}^{N} b_n cos(2\pi f nt) \tag{6.2}$$

And by replacing the fundamental frequency with it's equivalent period, eq. (6.2) becomes:

$$f(t) = c_0 + \sum_{n=1}^{N} a_n sin(\frac{2\pi nt}{T}) + \sum_{n=1}^{N} b_n cos(\frac{2\pi nt}{T}) \tag{6.3}$$

Since a sine wave is 90° out of phase with a cosine wave of similar amplitude, the sine component of the waveform may be represented as:

$$\begin{aligned} sin(2\pi f t) &= cos(2\pi f t + \frac{\pi}{2}) \\ sin(2\pi f t) &= cos(2\pi f t + \phi_n),\text{ where } \phi_n = \frac{\pi}{2} \end{aligned} \tag{6.4}$$

Thus, the cosine component of the waveform would be represented as:

$$\begin{aligned} cos(2\pi f t) &= cos(2\pi f t + 0) \\ cos(2\pi f t) &= cos(2\pi f t + \phi_n),\text{ where } \phi_n = 0 \end{aligned} \tag{6.5}$$

The Fourier series equation represented using only cosine components now becomes:

$$\begin{aligned} f(t) &= c_0 + \sum_{n=1}^{N} c_n cos(2\pi f nt + \phi_n) \\ f(t) &= c_0 + \sum_{n=1}^{N} c_n cos(\omega nt + \phi_n) \\ f(t) &= c_0 + \sum_{n=1}^{N} c_n cos(\frac{2\pi nt}{T} + \phi_n) \end{aligned} \tag{6.6}$$

The complex representation of the Fourier equation in eq. (6.6), using phasor notation, is given as:

$$\begin{aligned} f(t) &= c_0 + \sum_{n=1}^{N} c_n \, e^{j2\pi f nt} \\ f(t) &= c_0 + \sum_{n=1}^{N} c_n \, e^{j\omega nt} \\ f(t) &= c_0 + \sum_{n=1}^{N} c_n \, e^{j\frac{2\pi nt}{T}} \end{aligned} \tag{6.7}$$

By including the dc component c_0 into the summation, eq. (6.7) becomes:

$$\begin{aligned} f(t) &= \sum_{n=0}^{N-1} c_n \, e^{j2\pi f nt} \\ f(t) &= \sum_{n=0}^{N-1} c_n \, e^{j\omega nt} \\ f(t) &= \sum_{n=0}^{N-1} c_n \, e^{j\frac{2\pi nt}{T}} \end{aligned} \tag{6.8}$$

The DFT Explained:

For a discrete fourier series, the fundamental frequency f is given by:

$$f = \frac{1}{\tau N} \tag{6.9}$$

the time taken by the waveform is given by:

$$t = k\tau \tag{6.10}$$

By substituting for eq. (6.9) and eq. (6.10) into eq. (6.8), we get the following IDFT transform equation:

$$\begin{aligned} f(n) &= \sum_{k=0}^{N-1} c_n \, e^{j2\pi n \frac{k\tau}{\tau N}} \\ &= \sum_{k=0}^{N-1} c_n \, e^{j2\pi nk \frac{1}{N}} \\ &= \frac{1}{N} \sum_{n=0}^{N-1} c_n \, e^{j2\pi nk \frac{1}{N}} \\ &= \frac{1}{N} \sum_{n=0}^{N-1} F(k) \, e^{j2\pi nk \frac{1}{N}} \end{aligned} \tag{6.11}$$

In eq. (6.11), we divide N in order to normalize the transform.

The DFT for the periodic waveform is the inverse of the IDFT and is given as:

$$F(k) = \frac{1}{N} \sum_{n=0}^{N-1} f(n) \, e^{-j2\pi nk \frac{1}{N}} \tag{6.12}$$

The FFT Explained:

While using DFT a repeating waveform is reconstructed by multiplying N distinct sample points of the basis function coefficient, and adding them $N+1$ times. This means that a single calculation would be repeated $N^2 + N$ times. Thus, a 512 point DFT would require 262656 calculations. The FFT works by breaking down an N-point DFT representation of a time domain signal into N FFT signals of 1 point each and then synthesizing it into one frequency. With the FFT, an N-point FFT requires $2N$ calculations, so that a 512 point FFT would require only 1024 calculations. Computation of the FFT thus consumes far less computing resources and is hence much faster than the computation of the DFT, thereby reducing the sampling time dramatically for huge samples. The most popular Cooley and Tukey FFT implementation uses a radix of 2. if we therefore have non base two total number of sample points, we zero pad the samples so that the total number of samples is a base two number.
[103, Pg 225-239]

Finally, during OFDM symbol reception, the FFT is used to shift a signal out of the time domain and into the frequency domain. During OFDM symbol transmission, the IFFT is used to shift a signal out of the frequency domain and into the time domain.[13, Pg 612]

6.2.6 OFDM Pilot Subcarriers

4 pilot OFDM symbols of equal amplitude and power, and which are encoded with pseudorandom bits, are inserted in the midst of the data subcarriers during OFDM transmission, so as to help with the receiver with timing synchronization of the incoming OFDM signal and to prevent interference between subcarriers.

6.2.7 802.11 Convolution Encoder and Code Puncturing

Convolution encoding involves encoding the data bits with with error correction bits, that will form part of the code word. The 802.11 specification recommends the use of the Viterbi algorithm for decoding the convolution encoded bits. In order to reduce the number of transmitted bits and improve the data rate, code puncturing involves ommitting some of the bits in the transmitter. At the receiver, the ommitted bits are replaced with zero bits.[13, Pg 604]

6.2.8 OFDM Data Interleaving

OFDM data interleaving involves encoding the same code bits onto several OFDM subcarriers, so as to improve frequency diversity. Improved frequency diversity will in turn improve the likelihood of the transmitted code bits being successfully reconstructed at the receiver.[18, Page 54] In addition, adjacent code bits are mapped onto non-adjacent subcarriers to improve reliability during the first permutation of the data interleaving process. During the second permutation of the data interleaving process, code bits are alternately mapped from the most significant to the least significant bits of the constellation and vise versa.[13, Pg 605]

6.2.9 802.11 Scrambling Code

Each WLAN client device will have its own scrambling/descrambling code that it applies to transmitted or received MAC data. This scrambling code is used to link each OFDM transmission to a unique user and in the case of highly secure scramble codes, it prevents other users from listening in on another client's WLAN communication packets.

6.2.10 OFDM Cyclic Prefix

The cyclic prefix code word is inserted at the end of the transmitted PLCP frame so as to act as a guard interval which prevents OFDM ISI (Inter Symbol Interference) with a subsequent PLCP frame that may be subsequently transmitted. The PLCP frame is composed of several OFDM symbols.

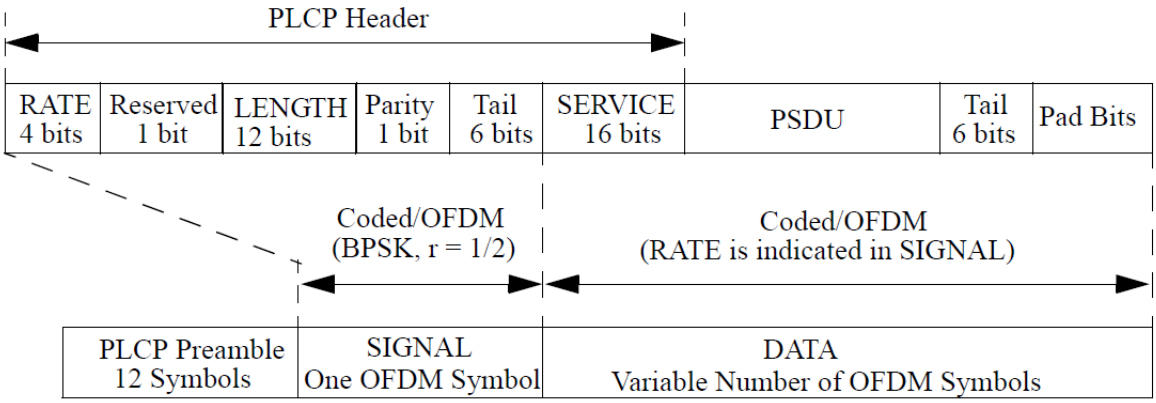


Figure 6.1: PLCP Frame Format [13]

6.3 The 802.11 PLCP Frame Format Implementation

The 802.11 PLCP frame format consists of the PLCP preamble, PLCP header, PSDU data, the tail bits and the pad bits.[13, Pg 600-610]

6.3.1 PLCP Preamble

For a 20MHz WLAN OFDM channel, the PLCP preamble consists of 10 short symbols of $0.8\mu secs$ each, two guard intervals of $0.8\mu secs$ each, and two long symbols of $3.2\mu secs$ each. This makes for a total preamble time of $8\mu secs$. For a 10 MHz channel, the timings are doubled, and for a 5 MHz channel, the timings are quadrupled.

Each short OFDM symbol consists of 12 subcarriers and are used by the WLAN receiver chip for signal detection, automatic gain control and diversity selection in the case of MIMO. Each long symbol consist of consists of 53 subcarriers, including a zero value subcarrier at DC. The long symbols are used by the receiver for timing synchronization. The PLCP preamble gnuradio signal processing block was ported to gnuradio companion using the gr_wlan/grc/gr_wlan_ofdm_preamble.xml file.

```
preamble = gr_wlan.ofdm_preamble(length, N_sym, gr_wlan_preamble)
```

Listing 6.1: PLCP Preamble (See line 399 of the wlan_utils.py file)

6.3.2 PLCP Header

The PLCP header was spread across two OFDM symbols. The first OFDM symbol containing part of the PLCP header was the *Signal* OFDM symbol which consisted of the *rate bits* (4 bits), *reserved bit* (1 bit), *length* bits (12 bits), *parity* bit (1 bit) and the *tail* bits (6 bits). The Signal OFDM symbol was encoded using the BPSK modulation mode at a coding rate of $frac12$ and was not scrambled. The second OFDM

symbol containing part of the PLCP header was a *Data* OFDM symbol and consisted of 16 *Service* bits.

```
def gr_wlan_make(payload, regime, symboltime, frame_type):
```

Listing 6.2: PLCP Header (See line 139 of the wlan_utils.py file)

Rate bits

The Rate bits were used to indicate the modulation mode and coding rate used to modulate the data OFDM symbols. When using the gr_wlan code to do the OFDM modulation, the rate which was used was dependent upon the regime option argument given.

```
    rate_bits = 0x0d
    if (regime == "1"):
        rate_bits = 0x0d
    elif (regime == "2"):
        rate_bits = 0x0f
    elif (regime == "3"):
        rate_bits = 0x05
    elif (regime == "4"):
        rate_bits = 0x07
    elif (regime == "5"):
        rate_bits = 0x09
    elif (regime == "6"):
        rate_bits = 0x0b
    elif (regime == "7"):
        rate_bits = 0x01
    elif (regime == "8"):
        rate_bits = 0x03
```

Listing 6.3: Rate bits (See lines 164-180 of the wlan_utils.py file)

Reserved bit

This bit is currently not being used by the 802.11g specification.

Length bits

12 Length bits were used to indicate the number of PSDU octets to be transferred by the physical layer to the MAC layer.

```
    app = 0
    for i in range (0,12):
        app = app | (((packet_len >> i) & 1) << (11 - i))
    Length = app
```

Listing 6.4: Length bits (See lines 183-186 of the wlan_utils.py file)

Parity bit

This is a positive (even) parity bit for bits 0-16 (Rate bits and Length bits).

```
    # generate parity check bit in SIGNAL OFDM symbol
    parA = parB = 0
    for k in range (0,12):
        parA += ((Length >> k) & (0x01))
        parB += ((rate_bits   >> k) & (0x01))
    parity_bit =  (parA + parB) % 2
```

Listing 6.5: Parity bit (See lines 188-193 of the wlan_utils.py file)

Tail bits

The tail bits consisted of 6 bits set to 0. The tail bits were used to enable the WLAN receiver chip to decode the length of bits indicated in the PLCP Header Length at the rate prescribed in the PLCP Rate field.

```
Signal_field = (rate_bits << 20) | (Length << 7) | Signal_tail | (parity_bit << 6)
```

Listing 6.6: Tail bits (See line 139 of the wlan_utils.py file)

Service bits

Though the service bits are part of the preamble, they form part of the PLCP Data Frame and not the PLCP signal frame like the rest of the PLCP header. The first 7 bits were used for scrambler initialization and were set to zero. The remaining 9 bits are reserved for future use by the 802.11 specification and were also set to zero.

```
    # generate 16 all-zero SERVICE bits
    SERVICE = chr(0x00) + chr(0x00)
```

Listing 6.7: Service bits (See lines 188-193 of the wlan_utils.py file)

6.3.3 PSDU Data

This forms part of the PLCP data frame and comes after the service bits that form part of the PLCP header. The PSDU data (together with the service, tail and pad fields which also form part of the PLCP data frame) were scrambled with a 127 length frame-synchronous scrambler. All the PLCP data octets, including the PSDU data, were transmitted in bit serial streams, least significant bit first.

6.3.4 Tail bits

The tail bits form part of the PLCP data frame and come after the PSDU data. The 6 bits were scrambled and set to zero, and used to improve the error probability, for subsequent OFDM symbols, of the convolution decoder in the WLAN chip by returning it to the zero state.

6.3.5 Pad bits

The pad bits, which vary in length, also form part of the PLCP data frame. Pad bits were added to the PLCP Data frame so that it became a multiple of N_{DBPS} (number of data bits per OFDM symbol). The length of the pad bits was calculated as follows:

Definitions:
N_{DBPS} = Number of data bits per OFDM symbol = $N_{CBPS} \times R$
N_{SYM} = Number of OFDM symbols per second = Math.ceil((Service bits + PSDU data bits + Tail bits) / N_{DBPS})
Service bits = 16
PSDU bits = 8× PSDU Length
Tail bits = 6
N_{DATA} = Total data bits = $N_{SYM} \times N_{DBPS}$ $N_{PAD} = N_{DATA}$ - (Service bits + PSDU data bits + Tail bits)

```
    # generate tail and padding bits
    if ((N_pad + 6) % 8 == 0):
        app = ''
        for i in range (0, (N_pad + 6)/8):
            app += chr(0x00)
        TAIL_and_PAD = app
    else:
        app = ''
        for i in range (0, (N_pad + 6)/8):
            app += chr(0x00)
        # add one more byte if N_pad + 6 is not a multiple of 8
        # we remove those bits again after the convolutional encoder
        TAIL_and_PAD = app + chr(0x00)
```

Listing 6.8: Pad bits (See lines 195-207 of the file wlan_utils.py)

6.4 The PLCP Data Frame OFDM Encoding Process

The PLCP data frame consists of the Service bits, PSDU octet data bits, the tail bits and the pad bits. The PLCP data bits were first taken through a scrambling code:

```
(pkt_scrambled, Length) = wlan_packet_utils.scrambler(pkt, Length)
```

Listing 6.9: Code scrambler (See line 139 of the wlan_tx.py file)

After the scrambling was complete, the PLCP data frame bits were then taken through a *convolution encoder* of coding rate $\frac{1}{2}$, $\frac{2}{3}$ or $\frac{3}{4}$. The coding rate used depended on the desired transmit data rate. Higher data rates were achieved through code puncturing (ommitting some bits during transmission since they can be reconstructed at the receiver), with the 802.11 WLAN specification recommending the use of the Viterbi code puncturing algorithm.

```
pkt_coded = wlan_packet_utils.conv_encoder(pkt_scrambled, Length, options.regime, N_cbps, N_bpsc, N_sym, N_rate)
```

Listing 6.10: Convolution encoder (See line 139 of the wlan_tx.py file)

The PLCP data bits were then data interleaved and spread across several data subcarriers, so that in the event of an error burst affecting a certain number of non-adjacent bits, there were still enough contiguous bits at the receiver to perform successful error correction and recover the original transmitted OFDM signal.

The PLCP data bits were then interleaved across 48 OFDM subcarriers and modulated using BPSK, QPSK, 16-QAM or 64-QAM modulation mode, at a coding rate of $\frac{1}{2}$, $\frac{2}{3}$ or $\frac{3}{4}$. 4 pilot subcarriers had a pseudo binary sequence BPSK modulated onto them so as to prevent spectral line generation.

```
pkt_interleaved = wlan_packet_utils.interleaver(pkt_coded , options.regime, N_cbps, N_bpsc)
```

Listing 6.11: Data bit Interleaver (See line 139 of the wlan_tx.py file)

6.5 A Description of the WLAN 802.11 MAC Layer

The 802.11 MAC data is encoded onto the PSDU data section of the PLCP data frame described above. The MAC Frame consists of the *MAC header*, *Frame body* (0-2312 octets) and the *CRC* (4 octets). The MAC header consists of the following field:[82]

6.5.1 MAC Header

The MAC header we created consisted of the following fields:

1. Frame control (2 octets): The Frame Control field is described further in the next subsection.
2. Duration ID (2 octets): The duration id was used to indicate the time in microseconds required to transmit the complete MAC frame.
3. Address 1 (6 octets): If ToDS in the frame control field is set to 1, then this field was used to hold the AP destination address. By default though, it was used to hold the end station destination address.

4. Address 2 (6 octets): If FromDS in the frame control field is set to 1, then this field was used to hold the AP source address. By default though, it was used to hold the mobile station source address.

5. Address 3 (6 octets): If ToDS is set to 1, then this field was used to hold the original source address. If FromDS is set to 1, then this field was used to hold the original destination address.

6. Sequence Control (2 octets): The Sequence Control field consisted of the fragment number and sequence number. If the current MAC frame is not part of the previous MAC frame, then the fragment number is incremented. If the current MAC frame is part of the previous MAC frame, the fragment number remains the same but the sequence number is incremented. For our beacon frame transmission, each beacon frame was assumed to be separate and so the fragment number and sequence number were both set to 0.

7. Address 4 (6 octets): If both ToDS and FromDS are set in the special case where a frame is being transmitted from one AP to another, then address3 contains the original destination address while address 4 contains the original source address. Our beacon frame was not specifically being transmitted to APs and so address 4 was never used.

```
    def get_beacon_info(payload, regime, symboltime):
```

Listing 6.12: Beacon MAC Header Generation (See line 27 of the wlan_utils.py file)

Frame Control

The Frame Control field field was used to transmit the following information:

1. Protocol version (2 bits): The 2 bits are currently fixed to zero by the 802.11 specification.

2. Type (2 bits): These 2 bits were used to indicate the type of frame that is being transmitted. There are 3 main types of frames: Management (00), Control (01) and Data (10) frames. The beacon frame we were transmitting was a management frame and therefore type bits 00 were used.

3. Sub Type (4 bits): Each WLAN MAC frame request or response will have it's own subtype code. The 802.11 WLAN Beacon Frame was being transmitted in our test bed setup since all WLAN chips are able to decode beacon frames without the need to get the chip into monitor mode first (the beacon frame can be received by all chips since it is also not encrypted and is simply a beacon used to inform all WLAN stations in an area of an AP's signal strength, security requirements and capabilities). The sub type 1000 was used since this is the subtype corresponding to a beacon frame.

4. ToDS (1 bit): This bit was set to 0, which is the default. It is only set to 1 if the MAC frame is addressed to an AP for fowarding to the distribution system, which was not true for our case.

5. FromDS (1 bit): This bit was set to 0, which is the default. It is only set to 1 if the MAC frame is coming from outside the distribution system, which was not true for our case.

6. More fragments (1 bit): This bit was not used during our beacon frame transmission. This bit is set to 1 if there are more fragments following the current frame control fragment. Otherwise it is set to 0.

7. Retry (1 bit): This bit was not used during our beacon frame transmission. This indicates that the fragment is a re-transmission of a previously transmitted fragment, which may happen if an acknowledgement packet is lost.

8. Power Management (1 bit): This bit was not used during our beacon frame transmission. This bit indicates whether the WLAN station is going into *power save* (set to 1) or *active* mode (set to 0) after transmission of the current MAC frame.

9. More Data: (1 bit): This bit was not used during our beacon frame transmission. When set to 1, this bit indicates that there's more frame control data following the power management field.

10. WEP (1 bit): This bit was not used during our beacon frame transmission. When set to 1, this bit indicates that the frame body following the MAC header will be encrypted using the WEP algorithm.

11. Order (1 bit): This bit was not used during our beacon frame transmission. When set to 1, this bit indicates that the users can accept a change in the ordering between unicast and multicast frames.

6.5.2 Beacon Frame Body

The Beacon frame we transmitted consisted of the following variable fields:

1. Timestamp (8 octets): The time the beacon frame was generated.

2. Interval (2 octets): The interval in microseconds between consecutive beacon frames being transmitted.

3. Capability Information (2 octets): This field was used to indicate network capabilities such as encryption support, ad hoc support, polling support etc.

4. Beacon frame tags: Each tag consisted of the tag number (1 byte/octet), tag length (1 byte/octet indicating how long the tag data will be), and the tag data itself. For example the *Rhino SSID tag* was tag number 0x00 of tag length 0x05. The tag data was 0x52, 0x68, 0x69, 0x6e, 0x6f, which was the hexadecimal representation of the word Rhino.

```
    # Default 802.11 LAN management beacon frame...
    # timestamp field, 8-bytes
    lan_management_frame = chr(0x78) + chr(0xa1) + chr(0xa5)
4   + chr(0x73) + chr(0xa7) + chr(0x05) + chr(0x00) + chr(0x00)
    # beacon interval field: 0.104448 seconds, 2-bytes
    lan_management_frame += chr(0x66) + chr(0x00)
    # capability information field, 2-bytes
```

```
    lan_management_frame += chr(0x31) + chr(0x04)

    #SSID parameter set
    lan_management_frame += chr(0x00) # Tag number 0
    lan_management_frame += chr(0x05) # Tag length
    # Rhino
    lan_management_frame += chr(0x52) + chr(0x68) + chr(0x69) + chr(0x6e) + chr(0x6f)

    #Supported rates
    lan_management_frame += chr(0x01) # Tag number
    lan_management_frame += chr(0x08) # Tag length
    #Tag interpretation, supported rates: 1, 2, 5.5, 6, 9, 11, 12, 18 Mbit/sec
    lan_management_frame += chr(0x82)+ chr(0x84)+ chr(0x8b)+ chr(0x0c)+ chr(0x12)+
    chr(0x96)+ chr(0x18)+ chr(0x24)

    # DS Parameter set
    lan_management_frame += chr(0x03) #Tag no.
    lan_management_frame += chr(0x01) #Tag length
    lan_management_frame += chr(0x01) #Tag interpretation, current channel is 1

    # Traffic Indication Map (TIM)
    lan_management_frame += chr(0x05) # Tag number
    lan_management_frame += chr(0x04) # TIM length
    lan_management_frame += chr(0x00) + chr(0x01) + chr(0x00) + chr(0x00) # TIM

    # Country information
    lan_management_frame += chr(0x07) # Tag number
    lan_management_frame += chr(0x06) # Tag length
    # Country code, ZA, Any environment
    lan_management_frame += chr(0x5a) + chr(0x41) + chr(0x20)
    # Start channel: 1, Channels: 13, Max TX Power: 20 dBm
    lan_management_frame += chr(0x01) + chr(0x0d) + chr(0x14)

    # ERP Information
    lan_management_frame += chr(0x2a) # Tag number
    lan_management_frame += chr(0x01) # Length
    # no non-ERP STAs, no protection, short/long pre-ambles
    lan_management_frame += chr(0x00)
    # Extended Supported Rates Information
    lan_management_frame += chr(0x32) # Tag number
    lan_management_frame += chr(0x04) # Length
    # TIM
    lan_management_frame += chr(0x30) + chr(0x48) + chr(0x60) + chr(0x6c)
```

Listing 6.13: Beacon Frame Body (See lines 198-240 of the wlan_tx.py file)

6.5.3 Cyclic Redundancy Check Field

The Cyclic Redundancy Check field was a 32 bit field that was used to hold a polynomial code checksum of the transmitted MAC frame. if the CRC value for the received MAC frame did not correspond to the transmitted CRC value, then the WLAN receiver chip would detect that a MAC transmission error had occured and would request that the MAC frame be re-transmitted (Negative Acknowledgement, NAK).

```
    MPDU_with_crc32 = gen_and_append_crc32(MPDU, packet)
```

Listing 6.14: Cyclic Redundancy Check (See line 225 of the wlan_utils.py file)

6.6 OFDM Modulation Implementation

The OFDM symbol transmission implementation is summarized in the figure below:

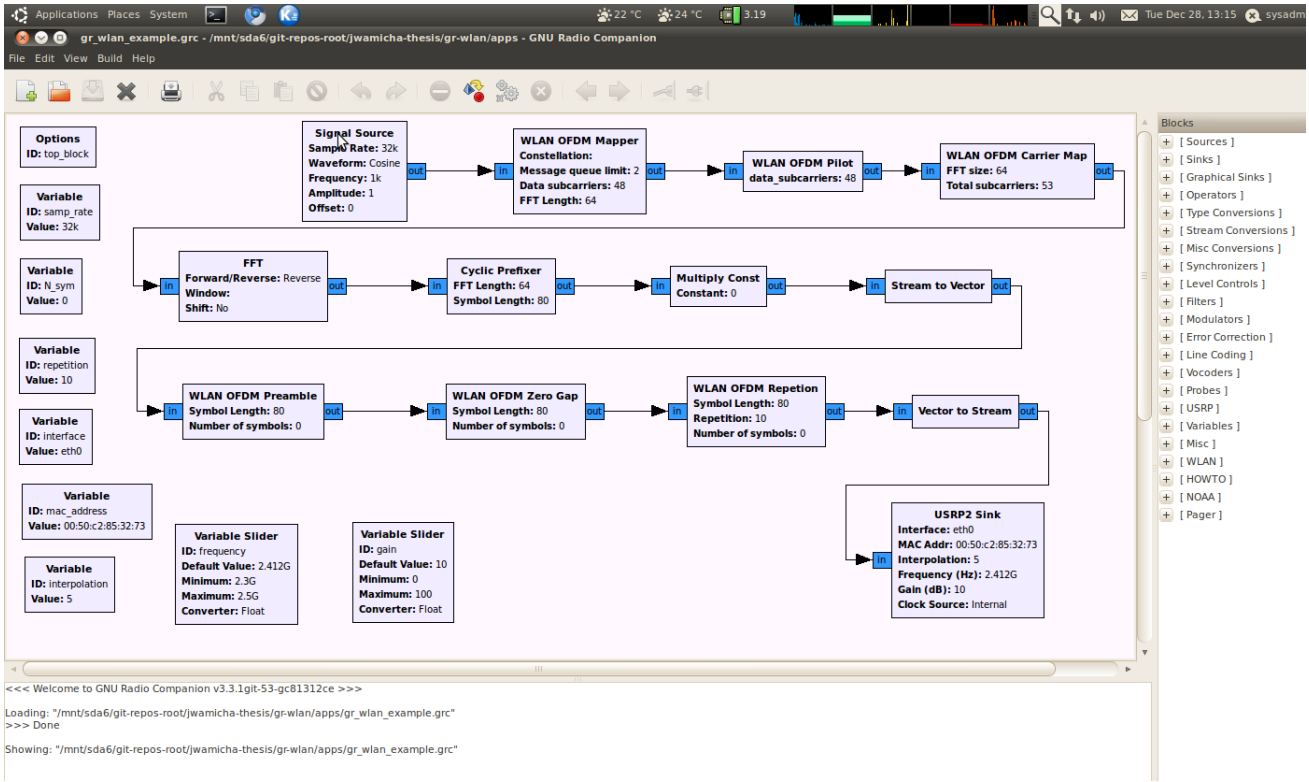


Figure 6.2: 802.11g WLAN Flowgraph

Once the 802.11 PLCP Frame OFDM symbols were generated (See the create_ofdm_symbols function inside WLAN_tx.py, line 120), the symbols were then modulated onto the subcarriers using the wlan_flow_tx class defined inside wlan_ofdm_tx.py. The following steps were followed during the OFDM modulation process:

6.6.1 Mapping of OFDM symbol bits to subcarriers of complex representation:

First, depending on the modulation mode that was selected, the generated OFDM symbol bits were mapped to subcarriers of complex representation.

```
self._pkt_input = gr_wlan.ofdm_mapper(rotated_const, msgq_limit, self.data_subcarriers, self.fft_length)
```

Listing 6.15: Symbol bits mapping (See line 67 of the wlan_ofdm_tx.py file)

6.6.2 Insertion of pilot subcarriers:

4 subcarriers were then inserted into the midst of the 48 data subcarriers:

```
self._pkt_input = gr_wlan.ofdm_mapper(rotated_const, msgq_limit, self.data_subcarriers, self.fft_length)
```

Listing 6.16: Insertion of pilot subcarriers (See line 67 of the wlan_ofdm_tx.py file)

6.6.3 Mapping of subcarriers to the correct spectral location:

The 48 data subcarriers, 4 pilot subcarriers and 12 guard interval subcarriers were mapped to their correct location in the spectrum. A DC component was also inserted in the middle of the subcarriers.

```
    self.cmap = gr_wlan.ofdm_cmap_cc(self.fft_length, self.total_sub_carriers)
```

Listing 6.17: See line 73 of the wlan_ofdm_tx.py file

6.6.4 IFFT of the 64 subcarriers:

The 64 subcarriers were then taken through an IFFT to shift them out of the frequency domain and into the time domain.

```
    self.ifft = gr.fft_vcc(self.fft_length, False, win, False)
```

Listing 6.18: Subcarrier mapping (See line 76 of the wlan_ofdm_tx.py file)

6.6.5 Apppending the cyclic prefix to the PLCP frame:

The cyclic prefix was then appended to the 64 subcarriers containing the PLCP Frame OFDM symbols.

```
self.cp_adder = gr.ofdm_cyclic_prefixer(self.fft_length, self.symbol_length)
```

Listing 6.19: Appending of the cyclic prefix (See line 76 of the wlan_ofdm_tx.py file)

6.6.6 Insertion of the PLCP preamble:

The PLCP Frame OFDM symbols were prefixed with the PLCP preamble.

```
self.preamble = wlan_packet_utils.insert_preamble(self.symbol_length, N_sym)
```

Listing 6.20: PLCP preamble insertion (See line 85 of the wlan_ofdm_tx.py file)

6.6.7 Addition of the zero gap after the cyclic prefix:

After the cyclic prefix, zero gap OFDM symbols were added so as to give the receiver enough time to prepare decoding the next PLCP frame sequence after transmission of the current PPDU frame is complete.

```
self.zerogap = wlan_packet_utils.insert_zerogap(self.symbol_length, N_sym)
```

Listing 6.21: Appending of the zero gap to the PLCP preamble (See line 88 of the wlan_ofdm_tx.py file)

6.6.8 Repeat transmission of the PPDU Frame OFDM symbols:

It was possible to continuously repeat transmission of the PLCP frame depending on value of the repeat option argument. When the repeat option argument to wlan_tx.py was set to 0, the OFDM encoded PLCP frame was transmitted ad infinitum.

```
self.repeat = gr_wlan.repetition(80, options.repetition, N_sym)
```

Listing 6.22: Repeat PLCP frame transmission (See line 91 of the wlan_ofdm_tx.py file)

6.6.9 Generation of the final OFDM modulation flowgraph:

The OFDM modulation steps were then strung together into a final flowgraph using python:

```
self.connect(self.pilot, self.cmap, self.ifft, self.cp_adder, self.scale, \\
self.s2v, self.preamble, self.zerogap, self.repeat, self.v2s)
```

Listing 6.23: Final 802.11g WLAN OFDM flow graph (See line 126 of the wlan_ofdm_tx.py file)

6.6.10 Transmission of OFDM symbols:

The OFDM symbols were then transmitted using the RFX2400 which is connected to the USRP2:

```
    # Start the flow graph.
    print "Start top block..."
    tb.start()
    print "Send 802.11 symbols..."
    # Send 802.11 frame symbols
    tb.flow_tx.send_ofdm_symbols(tb.ofdm_msg , eof=False)
    tb.flow_tx.send_ofdm_symbols('', eof=True)
```

Listing 6.24: Transmission of 802.11g WLAN OFDM symbols (See lines 256-262 of the wlan_tx.py file)

6.7 List of 802.11g Channels

As specified in the 802.11 specification the transmitted OFDM subcarriers were centered around the following 14 center frequencies or channels:

1. **Channel 1**: Transmission was centered around *2.412* GHz.

```
#./wlan_tx.py -e eth0 -f 2.412e9 -i 5 -t beacon -r 0
```

Listing 6.25: Transmission at *2.412* GHz

2. **Channel 2**: Transmission was centered around *2.417* GHz.

```
#./wlan_tx.py -e eth0 -f 2.417e9 -i 5 -t beacon -r 0
```

Listing 6.26: Transmission at *2.417* GHz

3. **Channel 3**: Transmission was centered around *2.422* GHz.

```
#./wlan_tx.py -e eth0 -f 2.422e9 -i 5 -t beacon -r 0
```

Listing 6.27: Transmission at *2.422* GHz

4. **Channel 4**: Transmission was centered around *2.427* GHz.

```
#./wlan_tx.py -e eth0 -f 2.427e9 -i 5 -t beacon -r 0
```

Listing 6.28: Transmission at *2.427* GHz

5. **Channel 5**: Transmission was centered around *2.432* GHz.

```
#./wlan_tx.py -e eth0 -f 2.432e9 -i 5 -t beacon -r 0
```

Listing 6.29: Transmission at *2.432* GHz

6. **Channel 6**: Transmission was centered around *2.437* GHz.

```
#./wlan_tx.py -e eth0 -f 2.437e9 -i 5 -t beacon -r 0
```

Listing 6.30: Transmission at *2.437* GHz

7. **Channel 7**: Transmission was centered around *2.442* GHz.

```
#./wlan_tx.py -e eth0 -f 2.442e9 -i 5 -t beacon -r 0
```

Listing 6.31: Transmission at *2.442* GHz

8. **Channel 8**: Transmission was centered around *2.447* GHz.

```
#./wlan_tx.py -e eth0 -f 2.447e9 -i 5 -t beacon -r 0
```

Listing 6.32: Transmission at *2.447* GHz

9. **Channel 9**: Transmission was centered around *2.452* GHz.

```
#./wlan_tx.py -e eth0 -f 2.452e9 -i 5 -t beacon -r 0
```

Listing 6.33: Transmission at *2.452* GHz

10. **Channel 10**: Transmission was centered around *2.457* GHz.

```
#./wlan_tx.py -e eth0 -f 2.457e9 -i 5 -t beacon -r 0
```

Listing 6.34: Transmission at *2.457* GHz

11. **Channel 11**: Transmission was centered around *2.462* GHz.

```
#./wlan_tx.py -e eth0 -f 2.462e9 -i 5 -t beacon -r 0
```

Listing 6.35: Transmission at *2.462* GHz

12. **Channel 12**: Transmission was centered around *2.467* GHz.

```
#./wlan_tx.py -e eth0 -f 2.467e9 -i 5 -t beacon -r 0
```

Listing 6.36: Transmission at *2.467* GHz

13. **Channel 13**: Transmission was centered around *2.472* GHz.

```
#./wlan_tx.py -e eth0 -f 2.472e9 -i 5 -t beacon -r 0
```

Listing 6.37: Transmission at *2.472* GHz

14. **Channel 14**: Transmission was centered around *2.484* GHz.

```
#./wlan_tx.py -e eth0 -f 2.484e9 -i 5 -t beacon -r 0
```

Listing 6.38: Transmission at *2.484* GHz

6.8 802.11g Transmission speeds

The following transmission speeds were achieved with our 802.11g WLAN OFDM implementation:

1. **6 Mbits/sec**: Transmission was centered around *2.412* GHz.

```
#./wlan_tx.py -e eth0 -f 2.412e9 -i 5 -t beacon -r 0
```

Listing 6.39: Transmission at *2.412* GHz

2. **Channel 2**: Transmission was centered around *2.417* GHz.

```
#./wlan_tx.py -e eth0 -f 2.417e9 -i 5 -t beacon -r 0
```

Listing 6.40: Transmission at *2.417* GHz

3. **Channel 3**: Transmission was centered around *2.422* GHz.

```
#./wlan_tx.py -e eth0 -f 2.422e9 -i 5 -t beacon -r 0
```

Listing 6.41: Transmission at *2.422* GHz

4. **Channel 4**: Transmission was centered around *2.427* GHz.

```
#./wlan_tx.py -e eth0 -f 2.427e9 -i 5 -t beacon -r 0
```

Listing 6.42: Transmission at *2.427* GHz

5. **Channel 5**: Transmission was centered around *2.432* GHz.

```
#./wlan_tx.py -e eth0 -f 2.432e9 -i 5 -t beacon -r 0
```

Listing 6.43: Transmission at *2.432* GHz

6. **Channel 6**: Transmission was centered around *2.437* GHz.

```
#./wlan_tx.py -e eth0 -f 2.437e9 -i 5 -t beacon -r 0
```

Listing 6.44: Transmission at *2.437* GHz

7. **Channel 7**: Transmission was centered around *2.442* GHz.

```
#./wlan_tx.py -e eth0 -f 2.442e9 -i 5 -t beacon -r 0
```

Listing 6.45: Transmission at *2.442* GHz

8. **Channel 8**: Transmission was centered around *2.447* GHz.

```
#./wlan_tx.py -e eth0 -f 2.447e9 -i 5 -t beacon -r 0
```

Listing 6.46: Transmission at *2.447* GHz

9. **Channel 9**: Transmission was centered around *2.452* GHz.

```
#./wlan_tx.py -e eth0 -f 2.452e9 -i 5 -t beacon -r 0
```

Listing 6.47: Transmission at *2.452* GHz

10. **Channel 10**: Transmission was centered around *2.457* GHz.

```
#./wlan_tx.py -e eth0 -f 2.457e9 -i 5 -t beacon -r 0
```

Listing 6.48: Transmission at *2.457* GHz

11. **Channel 11**: Transmission was centered around *2.462* GHz.

```
#./wlan_tx.py -e eth0 -f 2.462e9 -i 5 -t beacon -r 0
```

Listing 6.49: Transmission at *2.462* GHz

12. **Channel 12**: Transmission was centered around *2.467* GHz.

```
#./wlan_tx.py -e eth0 -f 2.467e9 -i 5 -t beacon -r 0
```

Listing 6.50: Transmission at *2.467* GHz

13. **Channel 13**: Transmission was centered around *2.472* GHz.

```
#./wlan_tx.py -e eth0 -f 2.472e9 -i 5 -t beacon -r 0
```

Listing 6.51: Transmission at *2.472* GHz

14. **Channel 14**: Transmission was centered around *2.484* GHz.

```
#./wlan_tx.py -e eth0 -f 2.484e9 -i 5 -t beacon -r 0
```

Listing 6.52: Transmission at *2.484* GHz

Chapter 7

Analysis and Results of the OFDM SDR Implementation using off-the-shelf WLAN chips

"For some reason people were unusually intense in opposition to CDMA technology. Bruce Lusignan, a brilliant professor of electrical engineering at Stanford, said that CDMA, as Qualcomm described it, violates the laws of physics - and this was quoted over and over again. So that laws of physics were involved in this debate! And because it was said to violate the laws of physics, lots of people jumped to the conclusion that Irwin Jacobs and Andrew Viterbi (of the Viterbi algorithm fame) were pushing a technology scam!" - Andrew J. Viterbi, inventor of the Viterbi algorithm and Qualcomm co-founder.

7.1 Overview

In this chapter we present the results of our SDR OFDM implementation by showing how several off the shelf chips were able to detect our Gnuradio *Rhino* SSID beacon frame WLAN OFDM transmission. Even though the results are not presented below, the OFDM transmission was also displayed on the Windows 7 Wireless Network List as well as on WLAN enabled Nokia phones.

7.2 Installing the Gnuradio WLAN Signal Processing Blocks

The Gnuradio gr-wlan code was installed (the gr-wlan code is inside the DVD that accompanies this thesis):

```
#cd gr-wlan
#./bootstrap
#./configure
#make
#make install
```

7.3 Transmitting the WLAN OFDM Symbols using Gnuradio

Once gr-wlan code was installed, to transmit the 802.11g WLAN beacon frame OFDM symbols at 54 Mbps, the following command was used:

```
#./wlan_tx.py --interface=eth1 --freq=2.412e9 --interp=5 --frame-type=beacon --regime=8 --repetition=0
```

7.4 Observing the Custom 802.11g beacon frame on Openmoko FreeRunner's ar6k chip

To view the *Rhino* SSID beacon frame WLAN OFDM transmission coming from the Gnuradio-USRP2-RFX2400 test bed, the Q icon on QtMoko was clicked, after which we navigated to Applications, Terminal and run the following command:

```
#iwlist eth0 scanning
```

The Atheros ar6k off-the-shelf chip was able to successfully decode our Gnuradio *Rhino* SSID beacon frame WLAN OFDM transmission:

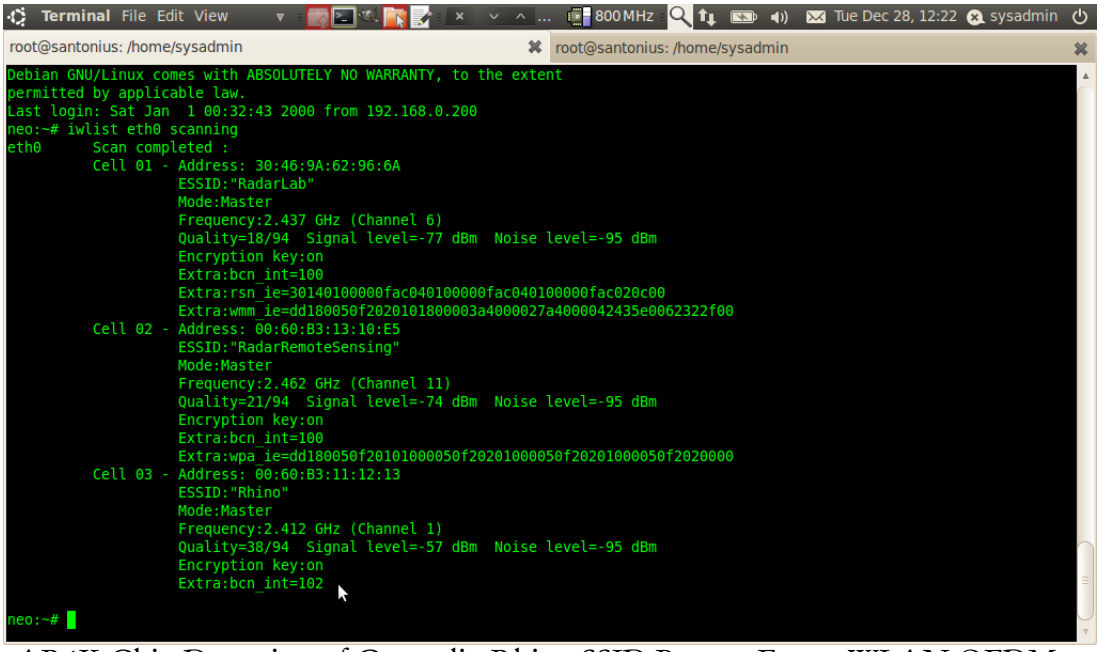


Figure 7.1: AR6K Chip Detection of Gnuradio Rhino SSID Beacon Frame WLAN OFDM transmission using iwlist

The chip transmits the receive power strength to the Linux kernel which then adds it to the radio tap header that gets read by the iwlist application. For the transmission above for example, the incoming receive signal strength was -57dBm.

7.5 Observing the Custom 802.11g beacon frame on the HP Packard Bell Dot Netbook's ar5k chip

To view the *Rhino* SSID beacon frame WLAN OFDM transmission coming from the Gnuradio-USRP2-RFX2400 test bed, on the ar5k chip, the kismet packet sniffer Linux application was used after setting the ar5k chip into monitor mode. By setting the chip into monitor mode, the chip will pass forward all 802.11g PLCP frames that it receives to the Linux kernel, irregardless of the PLCP frame's destination address (though this would only be a major concern if we wanted to sniff out 802.11g data frames. 802.11g beacon frames are ideally sent to all chips irregardless of whether the WLAN chip belongs to the network or not).

The name of the chip was obtained using the following command:

```
#root@santonius:/home/sysadmin# lspci | grep -i wireless
03:00.0 Ethernet controller: Atheros Communications Inc. AR5001 Wireless Network Adapter (rev 01)
```

The ar5k was set into monitor mode by using the command below:

```
#airmon-ng start wlan0
```

The kismet application was started using the following command:

```
#kismet
```

Kismet shows the Atheros ar5k off-the-shelf chip was able to successfully decode our Gnuradio *Rhino* SSID beacon frame WLAN OFDM transmission:

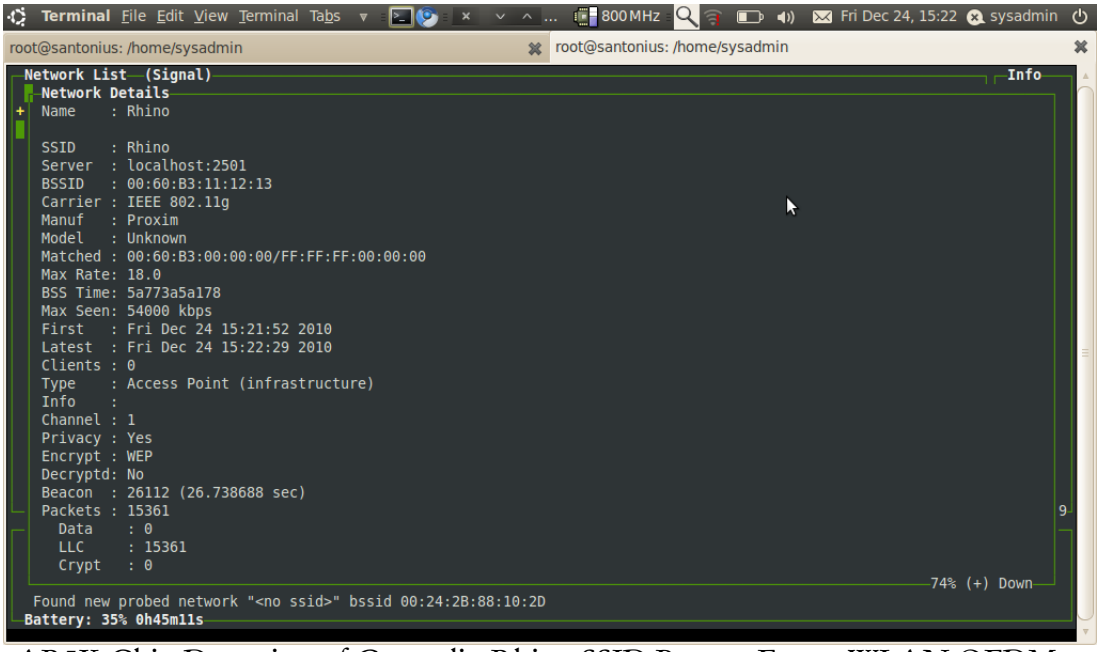


Figure 7.2: AR5K Chip Detection of Gnuradio Rhino SSID Beacon Frame WLAN OFDM transmission using Kismet

The airodump-ng Linux application was also used to monitor our Gnuradio *Rhino* SSID beacon frame WLAN OFDM transmission by using the following command:

```
#airodump-ng -c 1 -d 00:60:b3:11:12:13
```

The option argument *-c 1* refers to the 802.11g channel 1 which is centered at 2.412GHz. The option argument *-d 00:60:b3:11:12:13* is used to ask airodump-ng to monitor only packets coming from the BSSID source address 00:60:b3:11:12:13. This was the same source address we specified as the MAC address 2 field when building our beacon frame MAC header.

```
# some source mac-address
mac_address2  = chr(0x00) + chr(0x60) + chr(0xb3) + chr(0x11) + chr(0x12) + chr(0x13)
```

Listing 7.1: Source MAC Address 2 (See line 32 of the wlan_utils.py file)

The airodump-ng results are shown below:

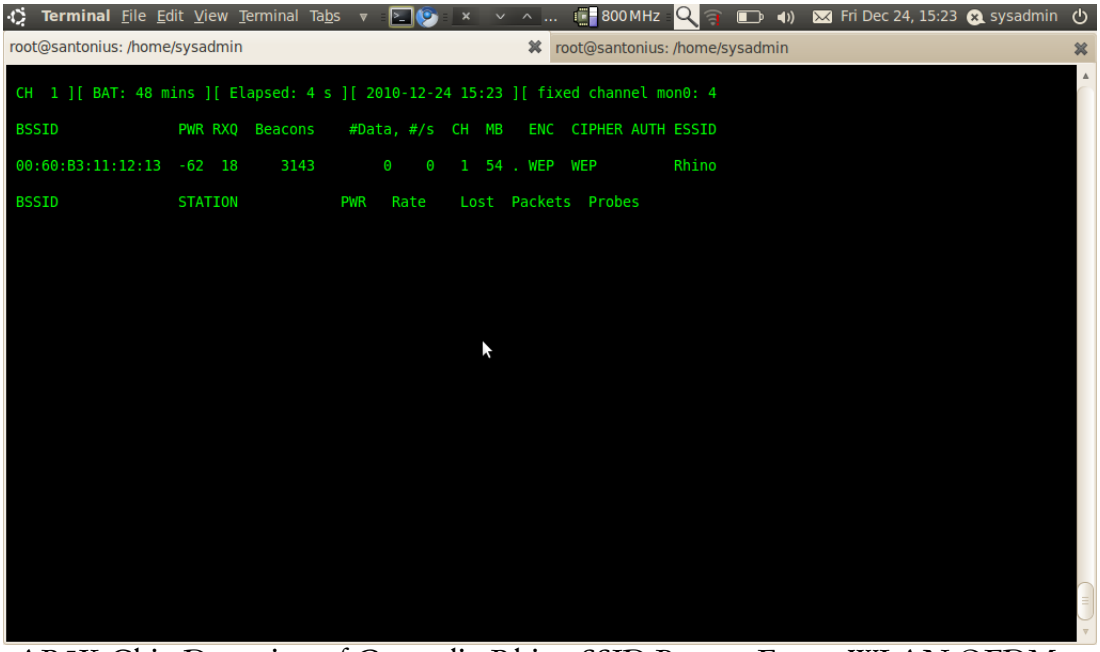


Figure 7.3: AR5K Chip Detection of Gnuradio Rhino SSID Beacon Frame WLAN OFDM transmission using airodump-ng

The chip transmits the receive power strength to the Linux kernel which then adds it to the radio tap header that gets read by the airodump-ng application. For the transmission above for example, the incoming receive signal strength was -62dBm.

The *Rhino* SSID beacon frame WLAN OFDM transmission could also be observed from Ubuntu's Network Manager top panel desktop applet:

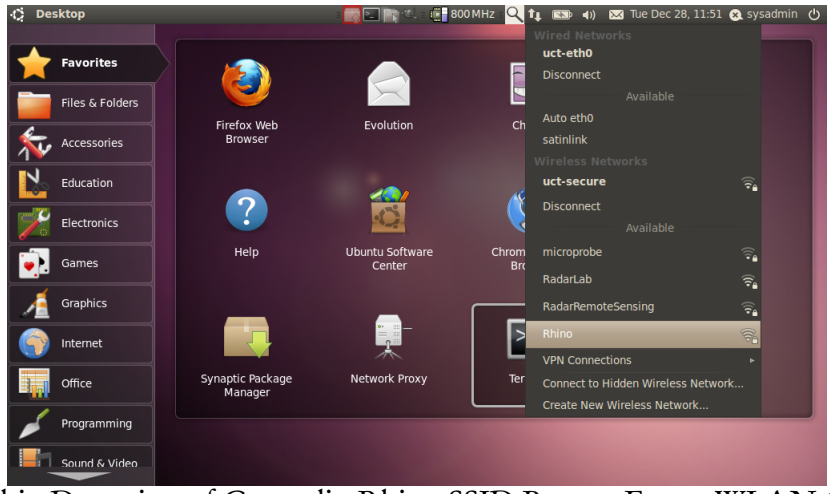


Figure 7.4: AR5K Chip Detection of Gnuradio Rhino SSID Beacon Frame WLAN OFDM transmission using Ubuntu Network Manager

7.6 Observing the Custom 802.11g beacon frame using the Broadcom BCM4312 WLAN chip

To view the *Rhino* SSID beacon frame WLAN OFDM transmission coming from the Gnuradio-USRP2-RFX2400 was also observed using the Broadcom BCM4312 WLAN chip running on our test notebook laptop. The chip information was obtained using the following command:

```
#root@astrodon:/mnt/sda3# lspci | grep Network
04:00.0 Network controller: Broadcom Corporation BCM4312 802.11b/g LP-PHY (rev 01)
```

The *Rhino* SSID beacon frame WLAN OFDM transmission could be observed from Ubuntu's Network Manager top panel desktop applet:

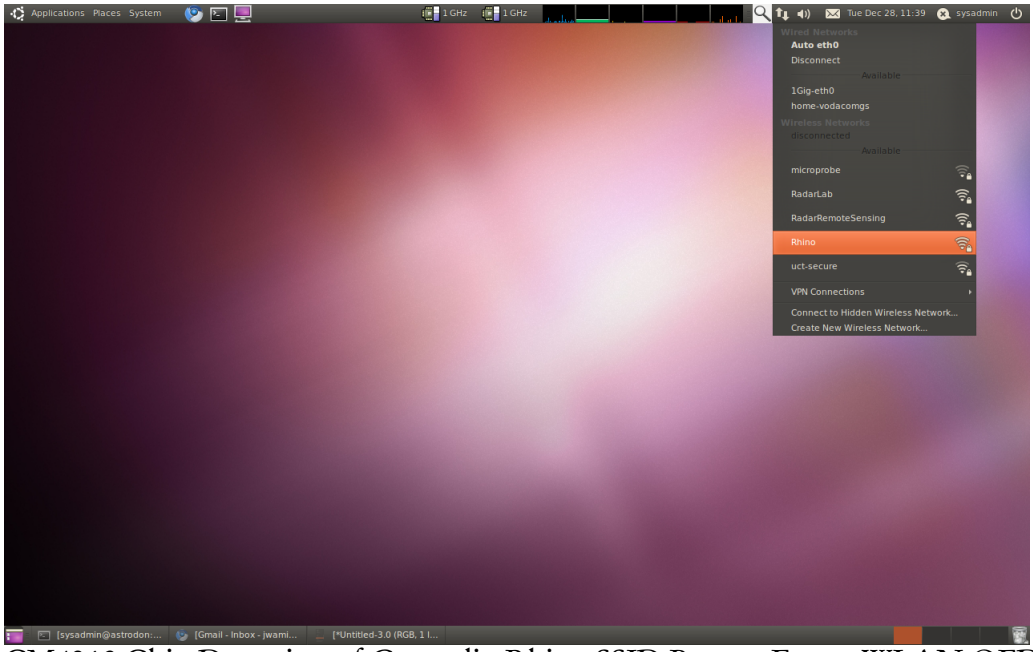


Figure 7.5: BCM4312 Chip Detection of Gnuradio Rhino SSID Beacon Frame WLAN OFDM transmission using Ubuntu Network Manager

Figure 7.6: Field tests of the WLAN OFDM transmission using 14 dBi antenna

Chapter 8

Rhino Expansion Board Design

"Outside our consciousness there lies the cold and alien world of actual things. Between the two stretches the narrow borderland of the senses. No communication between the two worlds is possible excepting across the narrow strip. For a proper understanding of ourselves and of the world, it is of the highest importance that this borderland should be thoroughly explored." - Heinrich Hertz, the inventor of the theory of light and electromagnetic waves.

8.1 Overview

8.2 Background: Gnuradio and the USRP2

The USRP2 is the current defacto hardware board used for SDR research. The USRP2 features a Xilinx Spartan 3-2000 FPGA which runs an inbuilt aeMB microblaze processor used to control all board functions. The FPGA is also used for DDC and DUC functions. This leaves about 30 multipliers, 3 Block RAMS and and about 40% of the logic slices available for developers to deploy their algorithms on. The USRP2 also features PMC connectors, a 14-bit 100 MSamples/sec ADC (LT2284), a 16-bit, 400 MSamples/sec DAC (AD9777) and a 1GigE ethernet port. The PMC connectors are used for attaching RF acquisition and transmission daughter boards to the USRP2. The 1GigE ethernet port is used to send and receive RF data to and from the host PC. The host PC will normally be installed with Gnuradio, which is used to generate the RF data to be sent out to the USRP2 for transmission, as well as process the incoming RF data from the USRP2. The USRP2 was designed between 2006 and 2008, with the first production boards becoming available in May of 2009.

8.3 The Rhino Board is born

It was becoming increasingly difficult to implement extreme DSP algorithms for Radar and Telecommunications experiments on the USRP2 since the USRP2 was originally designed between 2006-2008, using technology that was by then already more than 4 years old. In addition, by the first quarter of 2010 more powerful hardware was available and at a much cheaper price. Dr. Alan Langman therefore came up with the idea for the Rhino board, a board would still as affordable as the USRP2, but far more versatile and powerful than the USRP2. The Rhino FPGA board would feature adapter boards (such as the Rhino expansion board) which would enable USRP2 users to still be able to make use of their existing RF aquisition and transmission daughter boards. The Gnuradio host PC toolset would also be ported to Rhino so as to save USRP2 users code development time. So thus the Rhino board idea was born.

Rhino is an acronym for (R)econfigurable (H)ardware (I)nterface for computi(N)g and radi(O), a name coined by Simon Scott, the main developer of the Rhino board. Since the Rhino board was to be an open, reconfigurable and scalable FPGA board, it would form an excellent tool for reconfigurable computing research in the Radar, Telecommunications and Bioinformatics fields. It was to have the following features:

- Spartan 6 FPGA
- Texas Instruments AM3517 ARM Processor and Micro-controller
- IEEE 1588 Precision Time Protocol chip (this precision time protocol chip will prove very useful for scaling the Rhino boards as it will allow several Rhino boards to by synchronized together eg for MIMO experiments) using 10GigE.
- 2 SMA Clock Inputs and Outputs.
- 1 GPIO Connector
- 2 10GigE CX4 Connectors
- 2 FMC (LPC) Connectors
- 1 GigE Ethernet Port
- Power Supply Unit
- DDR2, DDR3 and Flash Memory
- Serial Connector
- USB Connector
- Audio In/Output

VHDL or Verilog HDL code running on the Xilinx Spartan 6 FPGA will allow for highly reconfigurable and re-usable software defined radio functions such as:

- Digital Down Conversion.
- OFDM Modulation/Demodulation code.
- Filters.
- Channel Estimation.
- Error Correction and Cancellation.
- RF Gain Control.
- Frequency Hopping
- RF Input/Output Tuning.

The Rhino board was to have the following advantages over the USRP2:

- Since the FPGA would no longer have an in-built processor like the USRP2, it would have a lot more logic cells, mulitipliers and block RAMs available for DSP algorithm experimentation.
- A dedicated, faster 600 MHz ARM microprocessor would now be available (the dedicated ARM microprocessor would replace the less powerful aeMB processor running from inside the USRP2 FPGA and which is normally clocked at 100 MHz).
- When using a 100 MHz clock, each FMC connector on Rhino would have a top data transfer speed of upto 1750 MBytes/sec. When using a 100 MHz clock, the maximum bit transfer speed across the PMC connectors is 800 MBytes/sec. Thus, by using FMC connectors the Rhino FPGA board will have the ability to communicate much faster with peripheral devices than the USRP2.
- 2 10GigE ethernet ports would be available to synchronize a whole stack of Rhino boards, while still ensuring that data is still efficiently transferred between them.
- By using the the FMC connectors, it would now be possible to acquire or transmit RF data through more channels. This would provide enhanced resolution for Radar and Telecommunications experiments. The FMC connectors would also allow Rhino to be used with several already existing ADC-DAC boards from manufacturers such as 4DSP.

8.4 The Rhino Expansion Board Design

The Rhino expansion board was to be designed to allow software defined radio researchers to interface any of their existing Gnuradio daughter boards to the new, powerful, open source, low cost Rhino FPGA

board. The Rhino expansion board would essentially consist of an ADC, DAC, Synchronization Clock and Voltage Controlled Temperature Compensated Crystal Oscillator. On the Rhino FPGA Board, an ARM AM3517 Processor Chip would be used as the micro-controller, while the Xilinx Spartan 6 FPGA would be used for signal processing functions.

The Rhino expansion board would also have to ensure that all RF functions on the Gnuradio daughter boards such as Gain Control, Power Output, RF Input Tuning, RF Outputs and IF Inputs could still be controlled from the Rhino FPGA board just like the USRP2. The Rhino board would act as the hardware bridge between the Rhino FPGA board and the RF daughter boards from the Gnuradio/USRP2 project.

8.5 Rhino Expansion Board Iteration 1

The first iteration of the Rhino expansion board consisted of the following main components:

- A low power, dual channel ADC from Linear Technologies: LT2285
- A low power, dual channel DAC from Analog Devices: AD9777
- 2 PMC connectors were placed onto the Rhino expansion board. The first PMC connector would be used for the transmit chain, while the second PMC connector would be used for the receive chain.
- 1 FMC connector was to be used for bridging the Gnuradio daughter cards to the Rhino FPGA Board. Both the transmit and receive chains were to be routed through this single FMC connector.
- Clock Input SMA connector.
- Clock Output SMA connector.
- Voltage Controlled Temperature Compensated Crystal Oscillator was used to generate the clock.

8.5.1 Rhino Expansion Board Receive Chain

The Receive Chain on the first iteration of the Rhino expansion board is shown below:

8.5.2 The Rhino Expansion Board Receive Chain Components

The first iteration of the Rhino expansion board receive chain consisted of the following main components:

- **PMC Connector**: An I-signal pair and Q-signal pair would be received from a single PMC Connector and routed to the ADC input.
- **Linear Technology's LT2285**: This is a 14-bit, LVDS, dual channel, 150 MSamples/sec ADC. It has a 3.3V supply voltage, wide analog input bandwidth of 640 MHz and low power dissipation of 790 mW. It would take in I and Q signals from the PMC connector. It would be pin compatible with the USRP2's LT2284 ADC, but have a higher sample rate (The USRP2's ADC has a sampling rate of 100 MSamples/sec).
- **An FMC-LPC Connector**: This would have LVDS data and control lines running from the ADC sitting on the Rhino expansion board to the Rhino FPGA board. To enable all Rhino expansion board receive and transmit data and control lines to fit onto 1 FMC-LPC connector, the LT2285 ADC would have to share it's data lines with the AD9777 DAC. The SPI control lines would be used to control data transfer between the LTC2285 and the Rhino FPGA board, without interfering with the AD9777 DAC.

8.5.3 Rhino Expansion Board Transmit Chain

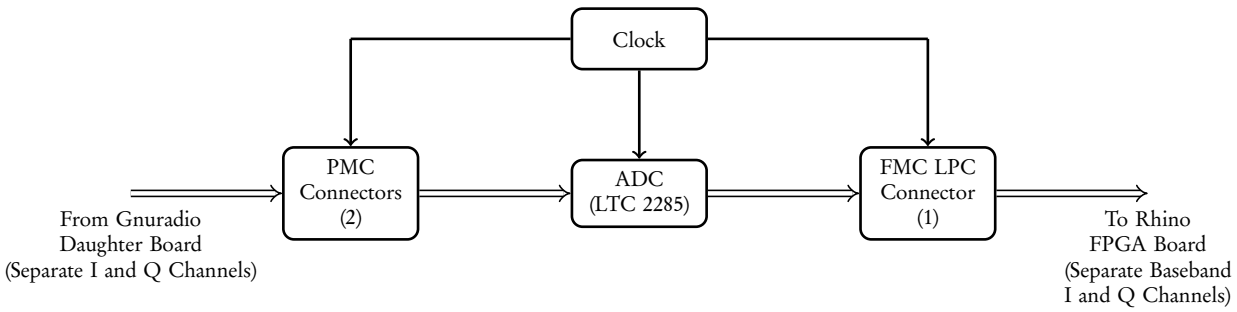
The first iteration of the Rhino expansion board transmit chain is shown below:

8.5.4 The Rhino Expansion Board Transmit Chain Components

The first iteration of the Rhino expansion board transmit chain consisted of the following main components:

- **Analog Device's AD9777**: This is a 16-bit, dual channel, 400 MS/sec DAC. It has a 3.3V supply voltage, 2x/4x/8 interpolating filter, 70+ MHz direct IF transmission/output. It outputs I and Q signals to the RFX2400 Daughter Board via the PMC Connector. This was the same DAC that was used on the USRP2.

Figure 8.1: Rhino Expansion Board Receive Chain: Iteration 1



- **PMC Connector**: An I-signal pair and Q-signal pair are sent to the RFX2400 Daughter Board from the Rhino expansion board via a single PMC Connector.
- **An FMC-LPC Connector**: This would have LVDS data lines running from Rhino FPGA board to the DAC that would be sitting on the Rhino expansion board. To enable all Rhino expansion board transmit and receive data and control lines to fit onto 1 FMC-LPC connector, the AD9777 DAC would have to share it's data lines with the LT2285 ADC. The SPI control lines would be used to control data transfer between the AD9777 and the Rhino FPGA board, without interfering with the LT2285 ADC.

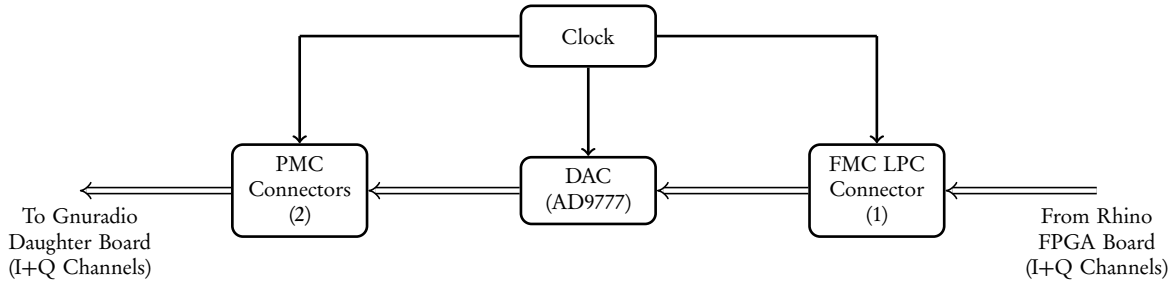
8.6 Rhino Expansion Board Iteration 2

The second iteration of the Rhino expansion board consisted of the following main components:

- A low power, 14-bit, 250 MSamples/sec, dual channel ADC from Texas Instruments: ADS62P49
- A low power, 16-bit, 8-bit input, 800 MSamples/sec, dual channel DAC from Texas Instruments: DAC3283
- 2 PMC connectors were placed onto the Rhino expansion board. The first PMC connector would be used for the transmit chain, while the second PMC connector would be used for the receive chain.
- 1 FMC-LPC connector was to be used for bridging the Gnuradio daughter cards to the Rhino FPGA Board. Both the transmit and receive chains were to be routed through this single FMC connector.
- A Voltage Controlled Temperature Compensated Crystal Oscillator was used to generate the clock.

The second iteration of the Rhino expansion board design was necessary since it was discovered that the SPI clock for the AD9777 had to be run at a recommended maximum clock rate of 15 MHz. Though it was still theoretically possible to have the ADC and DAC share the same data lines by say reading off

Figure 8.2: Rhino Expansion Board Transmit Chain: Iteration 1



the ADC during the rising edge of a 200 MHz clock and reading off the DAC during the clock's falling edge, it would unfortunately have been a complicated, non-standard solution. It is for this reason that the LT2285 ADC and the AD9777 DAC were replaced with the ADS62P49 ADC and DAC3283 DAC respectively.

The ADS62P49 was selected since it was a 14-bit, dual channel ADC with a sampling rate of 250 MSamples/sec, which was almost double the sampling rate of the LT2285 ADC. This would help with much better resolution for Radar and Telecommunications experiments. The DAC3283 DAC was selected because half the number of LVDS data lines coming in from the Rhino FPGA board (via the FMC LPC connector) could be used as compared to the AD9777 DAC. This is because a single input digital line was capable of receiving 2 bits at a time instead of the normal 1. This allowed us to fit all the data and clock lines onto a single FMC-LPC connector, without needing to use SPI, since this would have had a negative impact on the speed and performance of the Rhino expansion board. The DAC3283 had a sampling rate of 800 MSamples/sec, which was double the sampling rate of the AD9777 DAC.

The second iteration of the Rhino expansion board would also have the SMA clock inputs and outputs removed, hence greatly reducing the number of components required to fabricate the board. The expansion board also had the low power, dual channel, 16 bit AD5663 DAC (used to send analog waveforms to a secondary RF peripheral that may be connected to the daughter board via a serial port connector on the daughter board) and the low power dual channel AD7922 ADC (used to receive analog waveforms from a secondary RF peripheral that may be connected to the daughter board via a serial port connector on the daughter board), both from Analog Devices, removed. This was because we could not think of any reason why we would want to use the extra serial port found on some daugther boards, and also because we could find no literature about any Gnuradio research project making use of the extra serial port. As a result of removing all the above components in the second iteration of the Rhino expansion board, the cost to academic institutions of from the USRP2 to the Rhino FPGA board will be reduced. This is a very important factor to consider for financially constrained academic institutions that could potentially use Rhino.

8.6.1 Rhino Expansion Board Receive Chain

The Receive Chain on the second iteration of the Rhino expansion board is shown below:

8.6.2 The Rhino Expansion Board Receive Chain Components

The second iteration of the Rhino expansion board receive chain consisted of the following main components:

- **PMC Connector**: An I-signal pair and Q-signal pair would be received from the daughter board

using a single PMC connector and routed to the ADC input.

- **Texas Instrument's ADS62P49**: This is a 14-bit, LVDS, dual channel, 250 MSamples/sec ADC. It would take in I and Q signals from the PMC connector.
- **An FMC Connector**: This would have LVDS data and control lines running from the ADC sitting on the Rhino expansion board to the Rhino FPGA board.

8.6.3 Rhino Expansion Board Transmit Chain

The second iteration of the Rhino expansion board transmit chain is shown below:

8.6.4 The Rhino Expansion Board Receive Chain Components

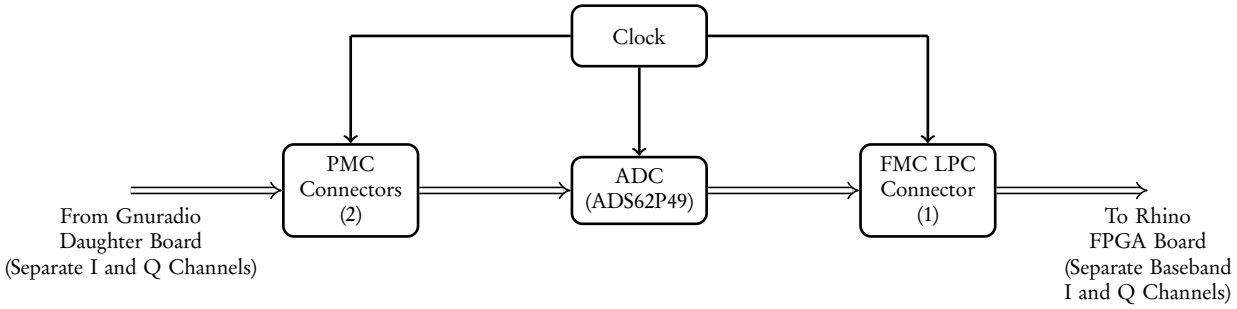
The second iteration of the Rhino expansion board receive chain consisted of the following main components:

- **PMC Connector**: An I-signal pair and Q-signal pair would be received from the daughter board on a single PMC connector and routed to the ADC input.
- **Texas Instrument's ADS62P49**: This is a 14-bit, LVDS, dual channel, 250 MSamples/sec ADC. It would take in I and Q signals from the PMC connector.
- **An FMC Connector**: This would have LVDS data and control lines running from the ADC sitting on the Rhino expansion board to the Rhino FPGA board.

8.6.5 Rhino Expansion Board Transmit Chain

The second iteration of the Rhino expansion board transmit chain is shown below:

Figure 8.3: Rhino Expansion Board Receive Chain: Iteration 2



8.6.6 The Rhino Expansion Board Transmit Chain Components

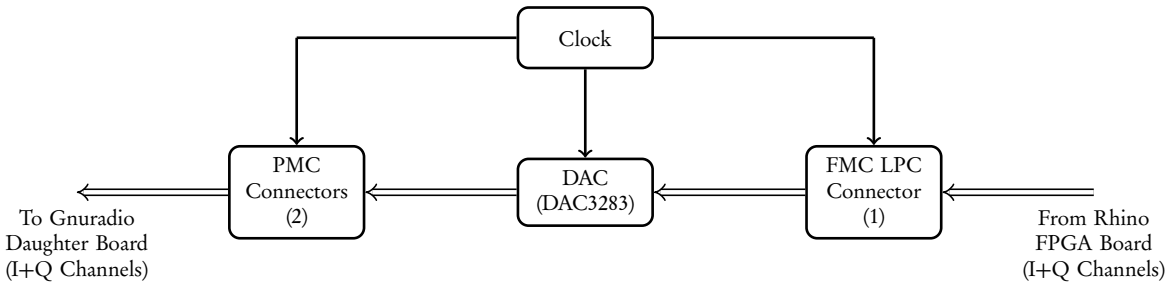
The second iteration of the Rhino expansion board transmit chain consisted of the following main components:

- **Texas Instrument's DAC3283**: This is a 16-bit, dual channel, 8-bit input, 800 MS/sec DAC. It has a 3.3V supply voltage, 2x/4x interpolating filter and 128 MHz direct IF transmission/output. It outputs I and Q signals to the RFX2400 Daughter Board via the PMC connector.
- **PMC Connector**: An I-signal pair and Q-signal pair are sent to the RFX2400 Daughter Board from the Rhino expansion board via a single PMC connector.
- **An FMC Connector**: This would have LVDS data lines running from Rhino FPGA board to the DAC that would be sitting on the Rhino expansion board.

8.7 Open Source EDA Software used to Design the Rhino Expansion Board

During our research into the Rhino expansion board design, we also looked at what open source EDA CAD applications we could use to design the schematics and gerber files required to fabricate the Rhino expansion board. This proved to be a challenging task, as often the documentation was nearly no where as good as the documentation that comes with commercial EDAs. The two most popular open source EDA CAD applications were gEDA and KiCAD. Both gEDA and KiCAD had no limits on the number of layers the PCB board could have, but in the end we settled for gEDA since it was relatively better documented. In the case of gEDA, there was no integrated environment through which one could design one's schematics, execute the design rule checker (DRC), generate the netlists, generate the bill of materials, do the spice simulations and generate PCB gerber files. Each of this tasks had to be done using a separate application which formed a part of the gEDA suite. The output of one application would be used as the input going to the next one. Both gEDA and KiCAD had no limit on the number of layers one could use for a board.

Figure 8.4: Rhino Expansion Board Transmit Chain: Iteration 2



8.8 gEDA Tools

The gEDA tools had to be used in the following order:

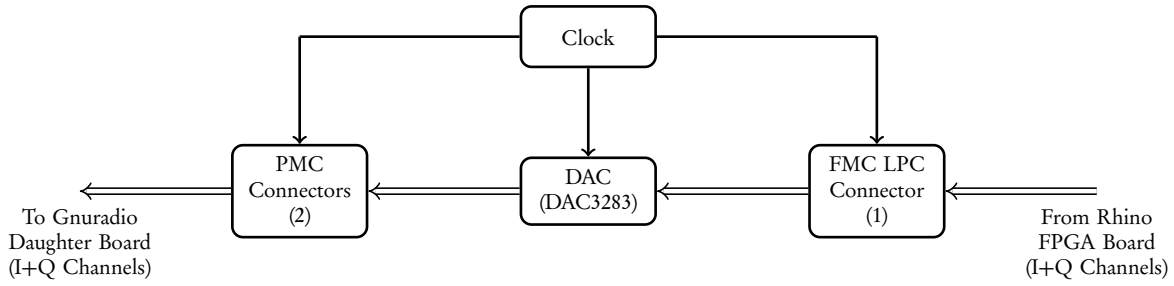
- gschem: for schematic design.
- gattrib: for assigning of net and symbol attributes or assignment of SPICE/Verilog/VHDL attributes.
- refdes_renum: for refdes assignment (each schematic component/symbol is assigned a unique name).
- gnetlist: for generating netlist files off your gschem schematics (schematic capture).
- gspiceui: for SPICE modelling once you have successfully generated your netlists using gnetlist.
- gschem2pcb: once your schematic design is complete, export your gschem *.sch files to PCB layout tool's *.PCB files.
- PCB: do your PCB layout. Export final PCB layout to gerber file. After using using PCB, the exported gerber files would be taken to the PCB Manufacturer for PCB trace generation and soldering of components onto the PCB board. Though the Rhino expansion board design to ensure compatibility with Gnuradio daughter boards was done, due to time constraints, the PCB layout could not be completed.
- gerberv: gerber file editor.

Since no single integrated environment existed for gEDA, a makefile was used to aid with the schematics generation process. To open all the schematics for example, a simple *make open* command could be used (which makes a call to the gschem application underneath).

```
$cd rhinoexpansion-v2
$make open
```

Listing 8.1: A command

Figure 8.5: Rhino Expansion Board Transmit Chain: Iteration 2



Once the schematics were complete, to generate the (reference designators), the *make renum* command could be used (which makes a call to the refdes_renum command underneath).

```
$make renum
```

Listing 8.2: A command

To generate a bill of materials csv file, the the *make bom2* command could be used (which makes a call to the 'gnetlist -g bom2' command underneath).

```
$make bom2
```

Listing 8.3: A command

To generate the spice files to be used for simulations of the analog components of the circuit, the *make net-spice-sdb* command could be used (which makes a call to the 'gnetlist -g spice-sdb' command underneath).

```
$make net-spice-sdb
```

Listing 8.4: A command

To generate the VHDL files that could be used for simulations of the digital components of the circuit, the *make net-VHDL* command could be used (which makes a call to the 'gnetlist -g VHDL' command underneath).

```
$make net-\gls{vhdl}
```

Listing 8.5: A command

Several other commands exist to help with EDA design while using gEDA. To get the rest of the commands, please consult the gEDA Makefile in the Appendix section (?? 13.4).

Chapter 9

Rhino Expansion Board, Schematics, Netlist and Bill of Materials

“I didn’t think; I experimented." - Wilhelm Conrad RÃűntgen, the inventor of the X-ray.

9.1 Overview

The Rhino expansion board schematics and bill of materials are presented and explained in this chapter.

9.2 Rhino Expansion Board Schematics: Iteration 1

Page 1 of the Rhino expansion board schematics shows the LT2285 ADC and AD9777 DAC nets:

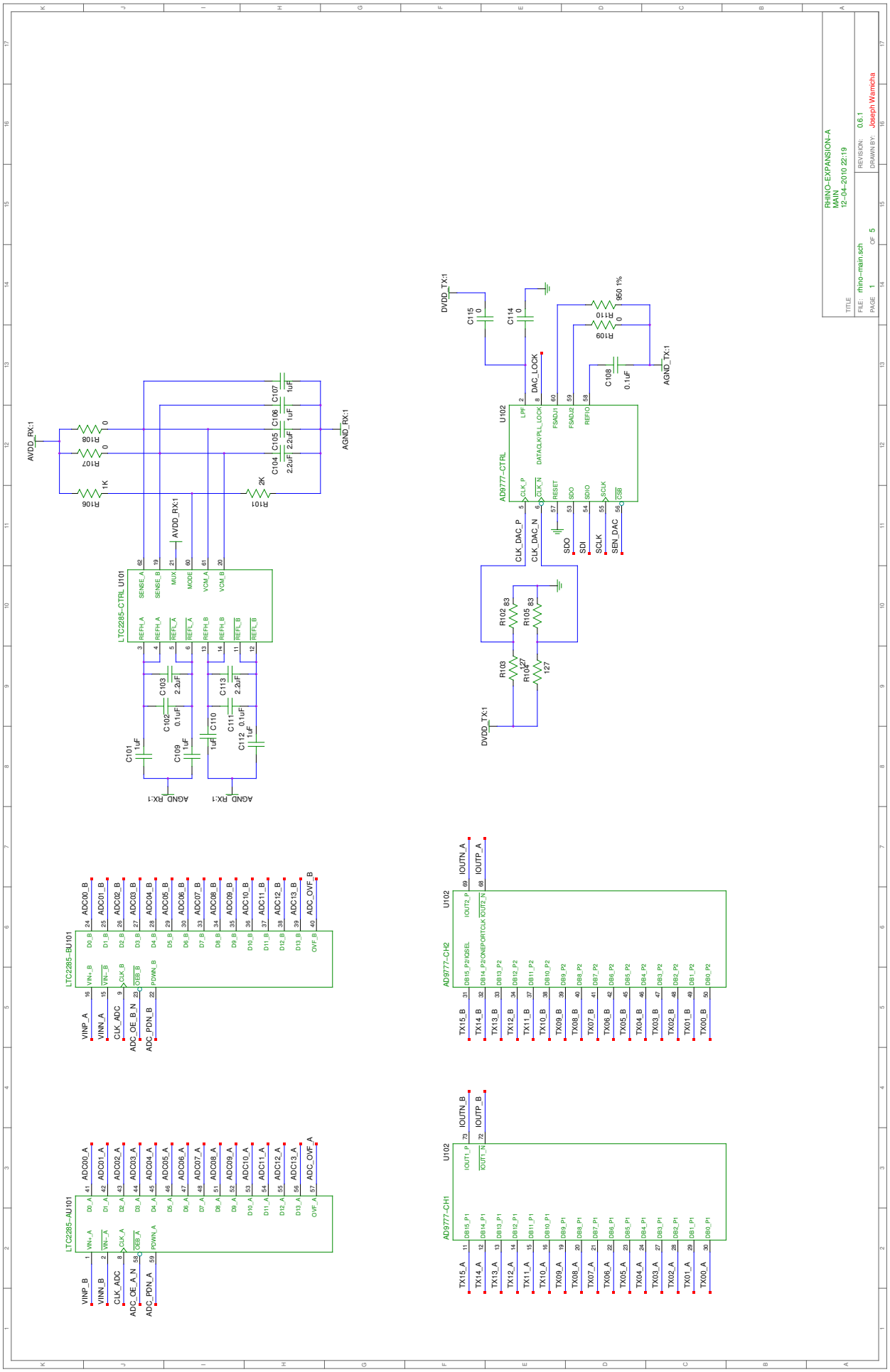


Figure 9.1: Rhino Expansion Board V1: Main

Page 2 of the Rhino expansion board schematics shows the FMC connector nets:

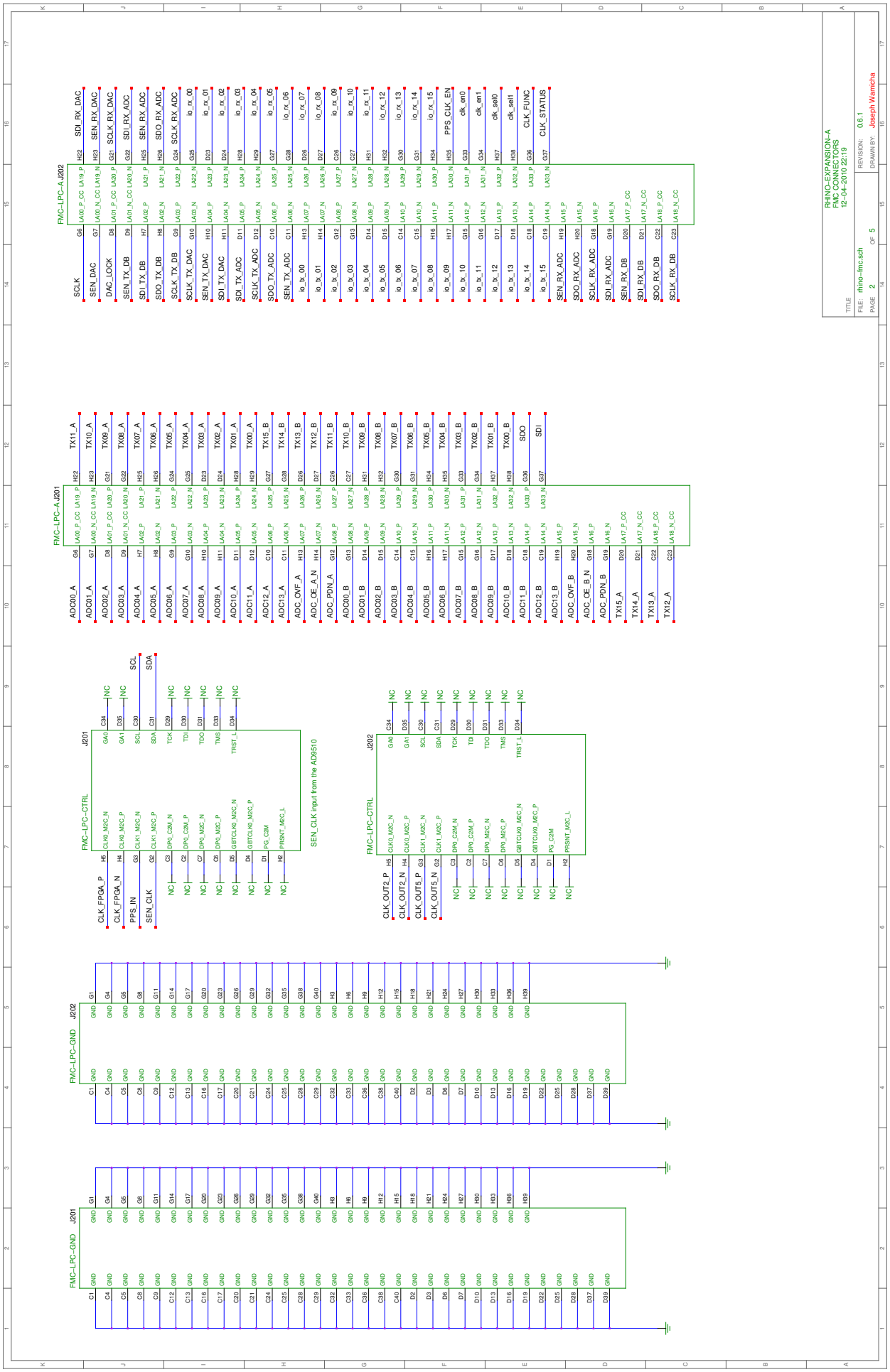


Figure 9.2: Rhino Expansion Board V1: FMC

Page 3 of the Rhino expansion board schematics shows the nets for the 2 PMC connectors:

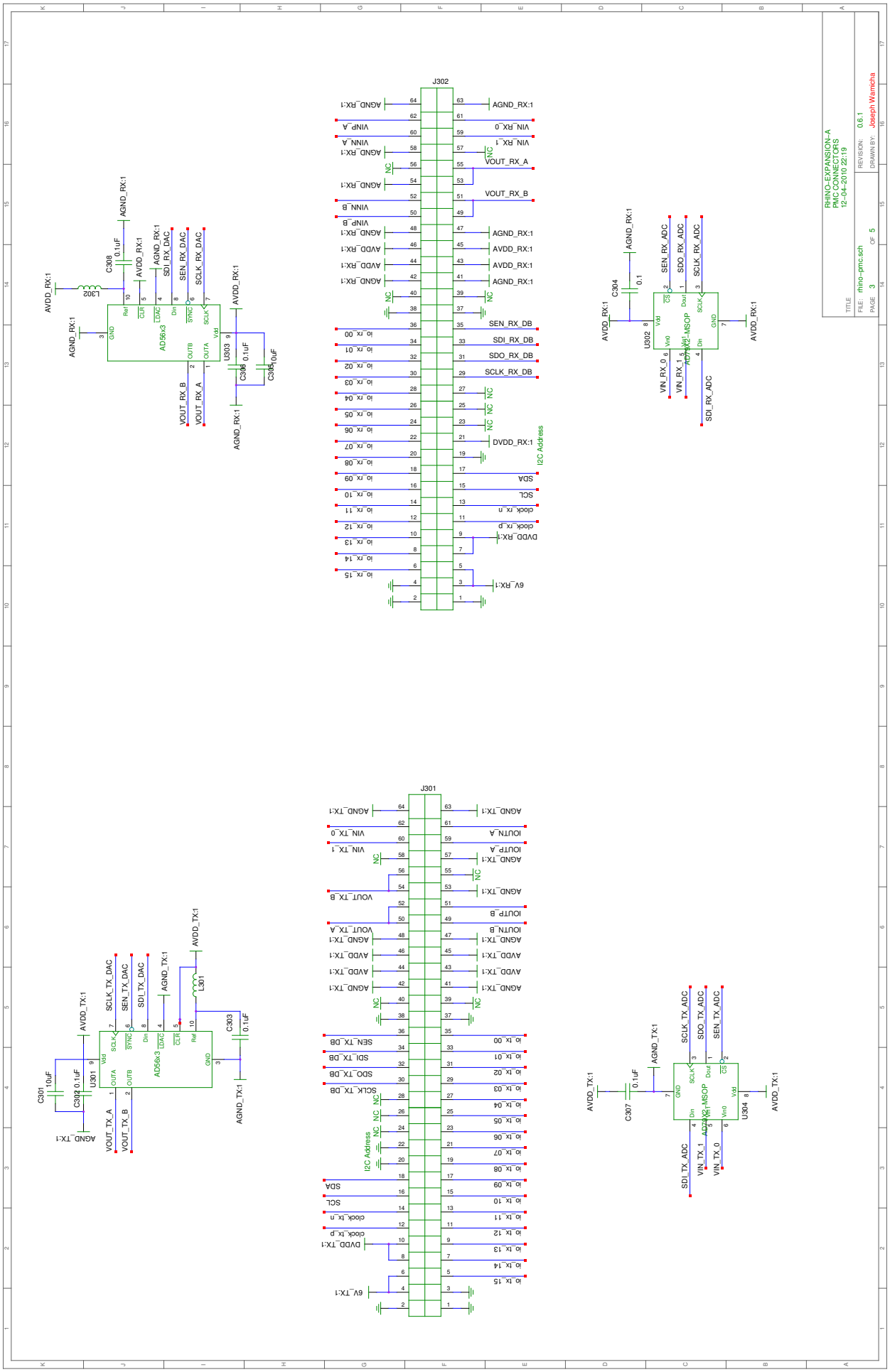


Figure 9.3: Rhino Expansion Board V1: PMC

Page 4 of the Rhino expansion board schematics shows the power nets:

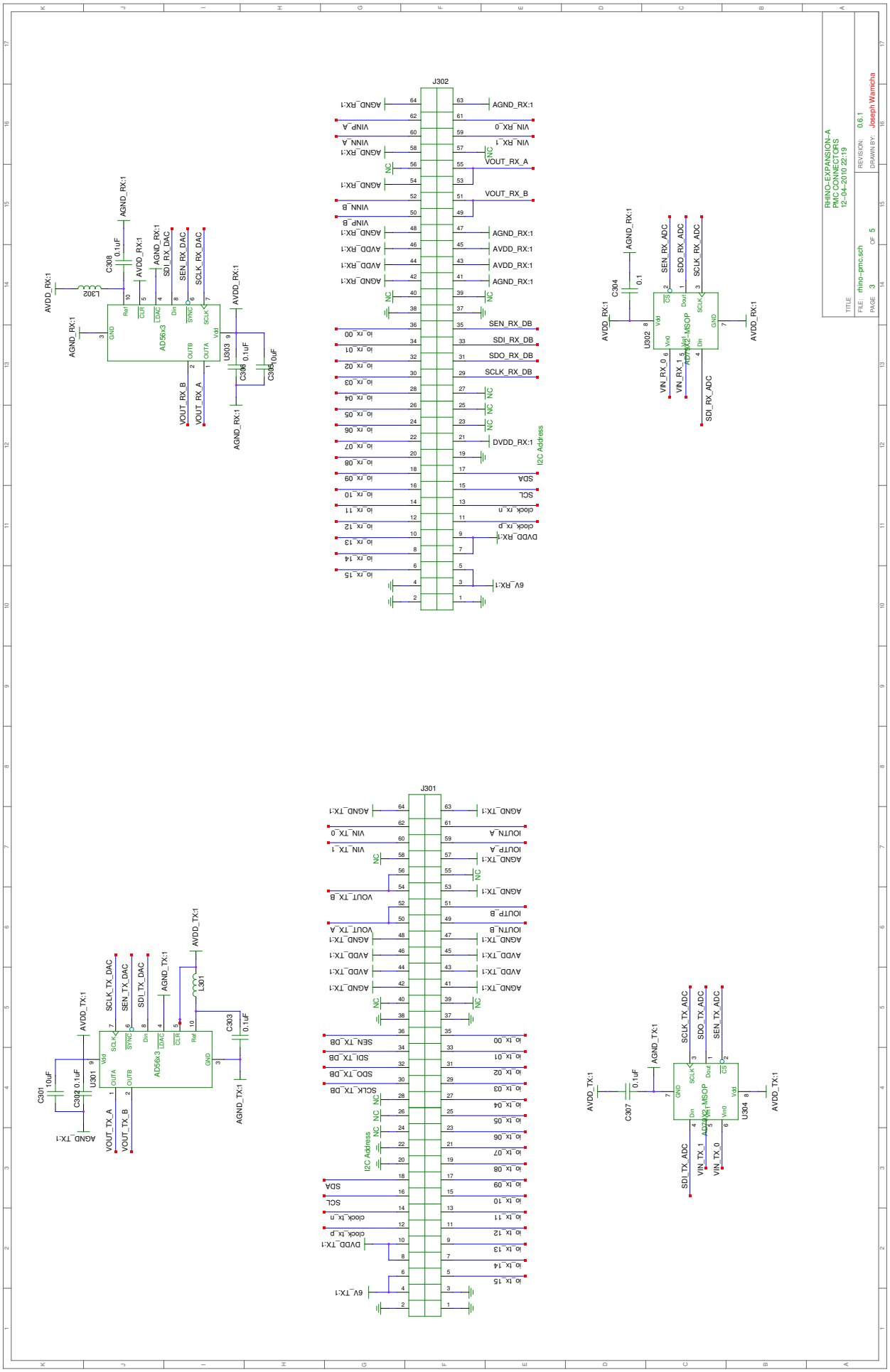


Figure 9.4: Rhino Expansion Board V1: Power

Page 5 of the Rhino expansion board schematics shows the clock nets:

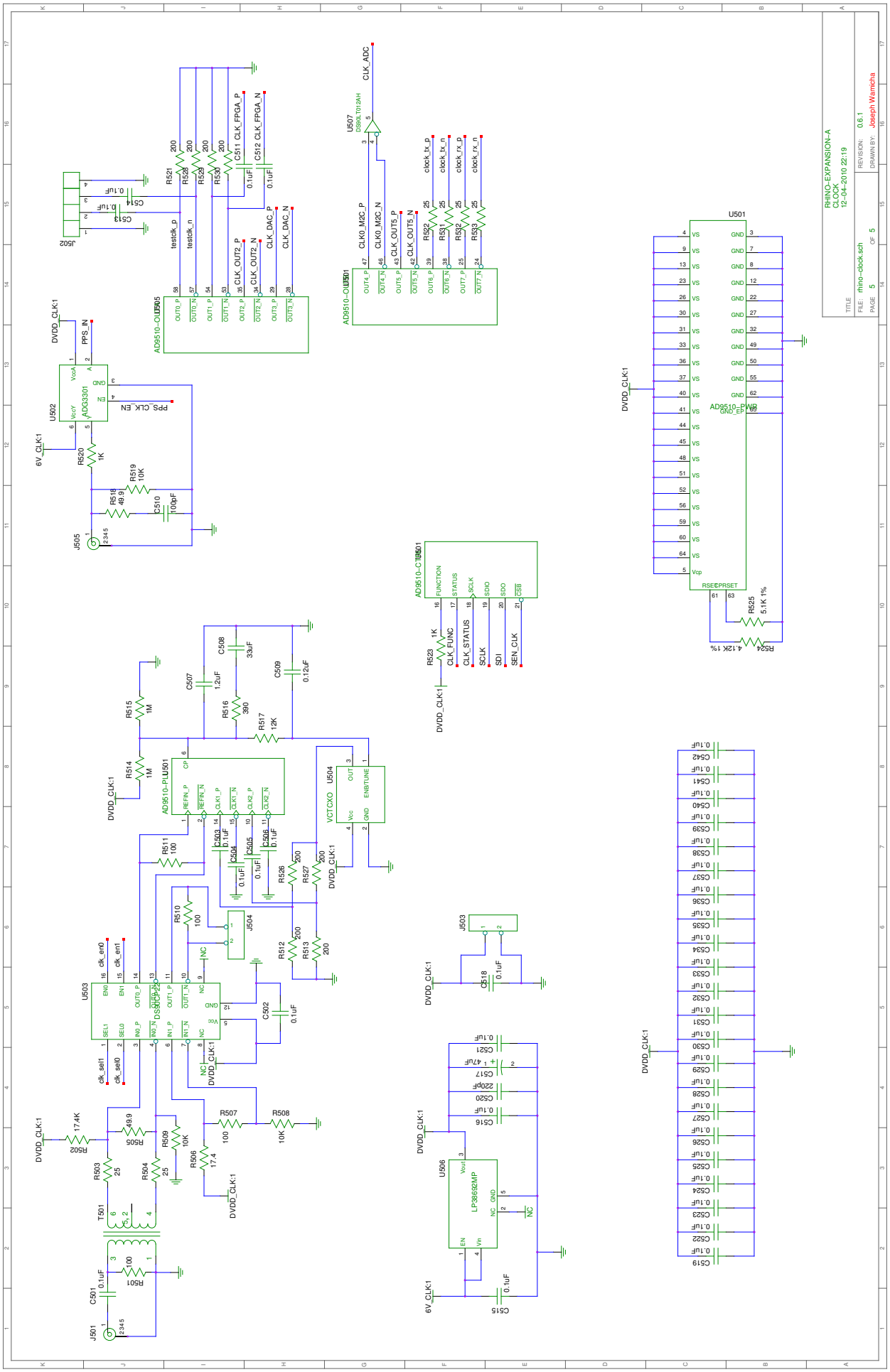


Figure 9.5: Rhino Expansion Board V1: Power

9.2.1 Bill of Materials

The Bill of Materials for the first iteration of the Rhino expansion board is shown in the next page:

Table 9.1: Rhino Expansion Board BOM: Iteration 1

refdes	footprint	value	description	qty
U101	QFN64	unknown	Dual 14-bit 125 MSps ADC	1
R101	603	2K	unknown	1
U102	TQFP80	unknown	400 MS/s 16-bit DAC	1
C103,C104,C105,C113	1206	2.2uF	capacitor	4
R103,R104	603	127	resistor	2
R102,R105	603	83	resistor	2
R107,R108,R109	402	0	resistor	3
R110	603	950 1%	resistor	1
C114,C115	603	0	capacitor	2
J201	fmc	unknown	FMC Connector	2
J301,J302	pmc	unknown	PMC Connectors	2
C304	603	0.1	capacitor	1
U301,U303	MSOP10	unknown	AD5623/AD5643/AD5663 (+R) Dual 12/14/16-bit serial DAC	2
C301,C305	1206	10uF	capacitor	2
U302,U304	MSOP8	unknown	AD7912/7922 Analog Devices Dual 10/12-bit serial ADC	2
L301,L302,L401,L402,L403,L404,L405,L406,L407	1206	unknown	inductor	9
C101,C106,C107,C109,C110,C112,C407,C426,C449	1206	1uF	capacitor	9
R401,R402	603	6	resistor	2
U502	SC70-6	unknown	Level translator	1
T501	603	unknown	unknown	1
J502	SIP4	unknown	unknown	1
J503,J504	SIP2N	unknown	generic connector	2
R502	1206	17.4K	unknown	1
U503	TSSOP	unknown	2x2 LVDS Crosspoint Switch, in SO16 or TSSOP16	1
R506	603	17.4	resistor	1
R501,R507,R510,R511	603	100	resistor	4
U504	vctcxo	unknown	VCTCXO	1
R514,R515	603	1M	resistor	2
C507	1206	1.2uF	capacitor	1
C508	1206	33uF	capacitor	1
R516	603	390	resistor	1
R517	603	12K	resistor	1
C509	603	0.12uF	capacitor	1
J501,J505	SMA_VERT	unknown	unknown	2
R505,R518	603	49.9	resistor	2
R508,R509,R519	603	10K	resistor	3
C510	603	100pF	capacitor	1
U501,U505	LFCSP64	unknown	Clock Divider and PLL	2
U506	SOT223-5	unknown	National Fixed LDO, 1A	1
C406,C408,C413,C418,C423,C427,C450,C517	1206	47uF	polarized capacitor	8
U507	SOT23-5	unknown	LVDS Receiver	1
R106,R520,R523	603	1K	resistor	3
R524	603	4.12K 1%	resistor	1
R525	603	5.1K 1%	resistor	1
R512,R513,R521,R526,R527,R528,R529,R530	603	200	resistor	8
R503,R504,R522,R531,R532,R533	603	25	resistor	6
C403,C405,C410,C412...C541,C542	603	0.1uF	capacitor	76

9.3 Rhino Expansion Board Schematics: Iteration 2

Page 1 of the Rhino expansion board schematics shows the ADS62P49 ADC and DAC3283 DAC nets:

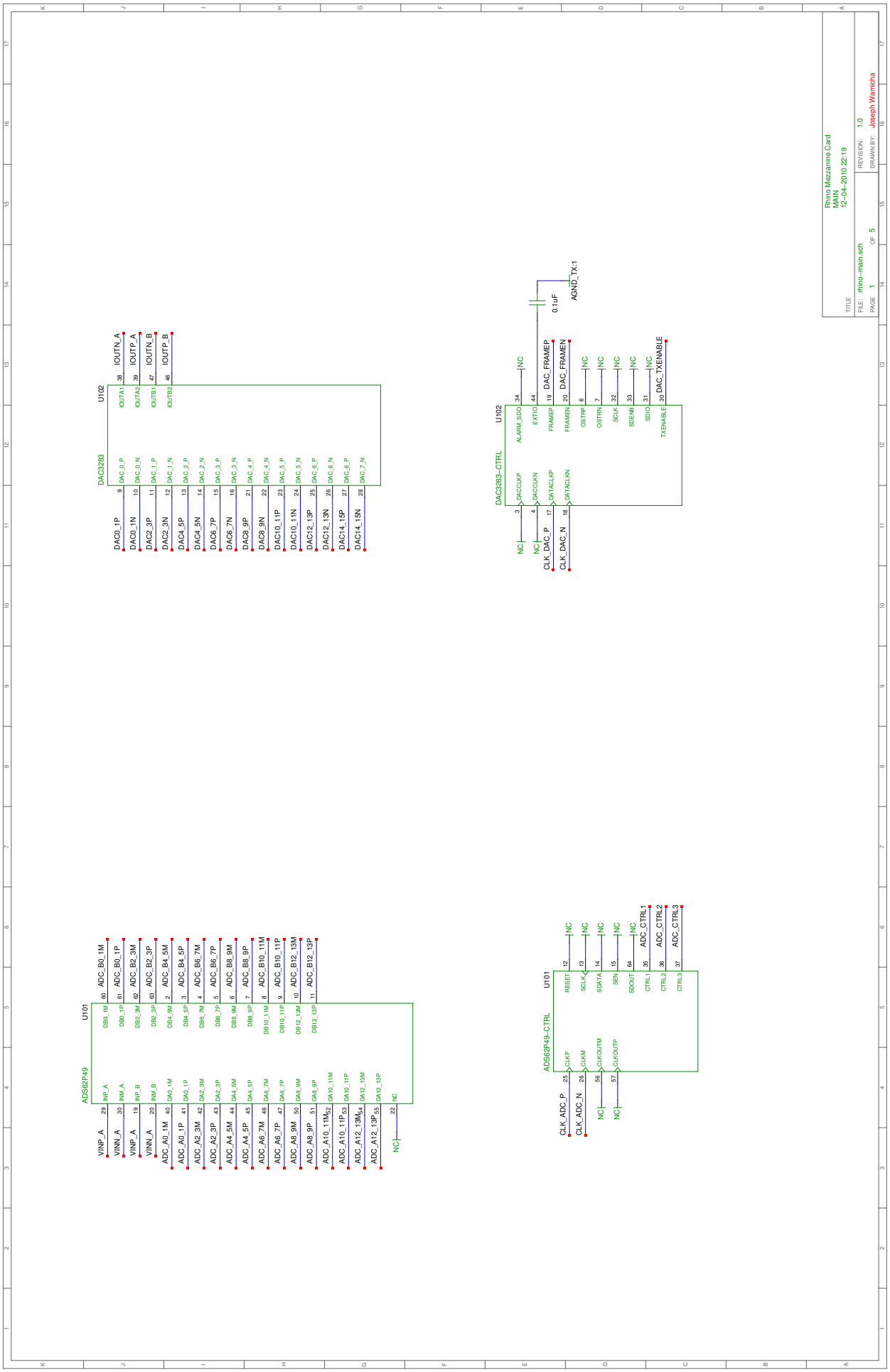


Figure 9.6: Rhino Expansion Board V2: Main

Page 2 of the Rhino expansion board schematics shows the FMC connector nets:

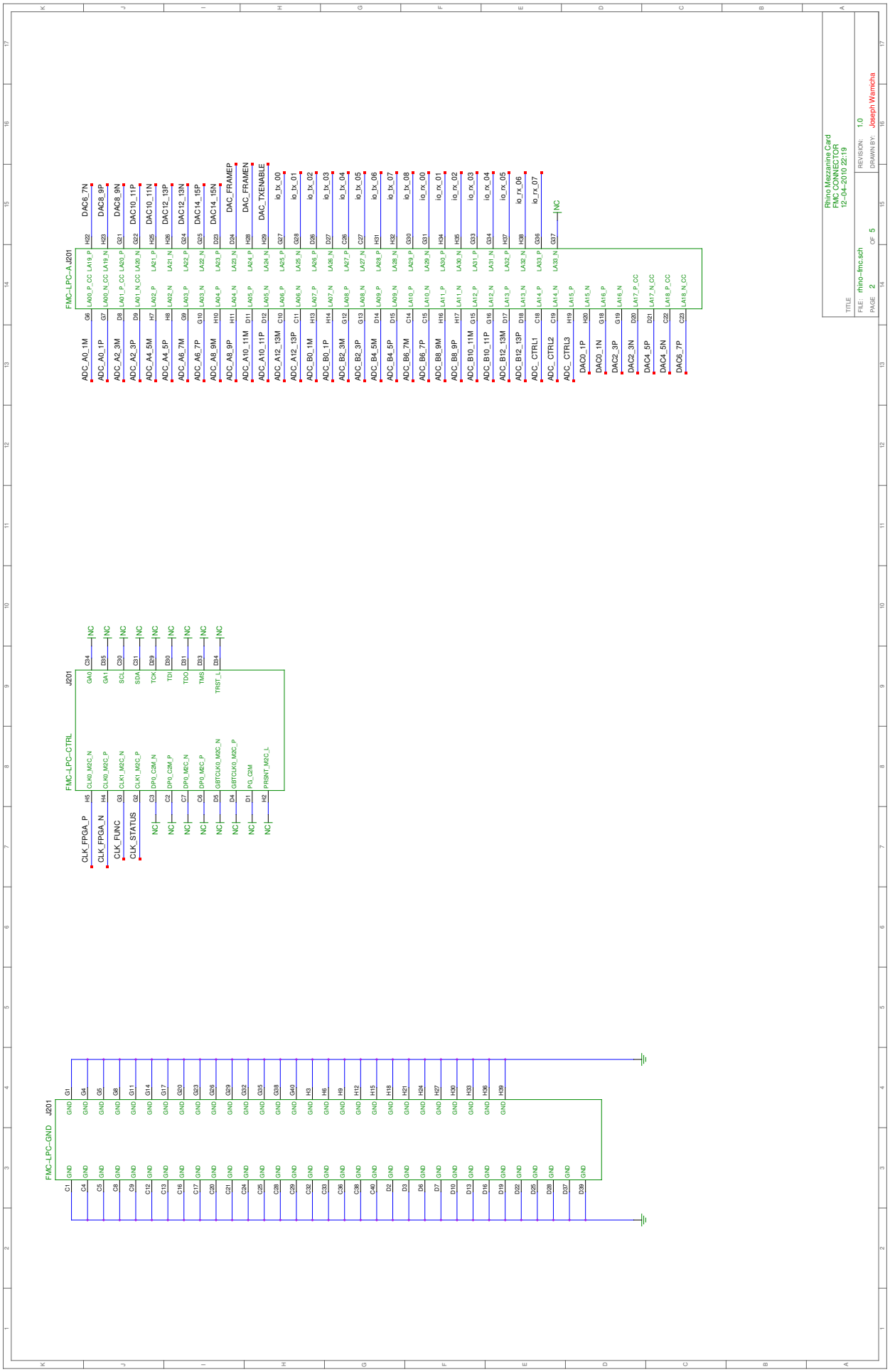


Figure 9.7: Rhino Expansion Board V2: FMC

Page 3 of the Rhino expansion board schematics shows the nets for the 2 PMC connectors:

Figure 9.8: Rhino Expansion Board V2: PMC

Page 2 of the Rhino expansion board schematics shows the power nets:

Figure 9.9: Rhino Expansion Board V2: Power

Page 2 of the Rhino expansion board schematics shows the clock nets:

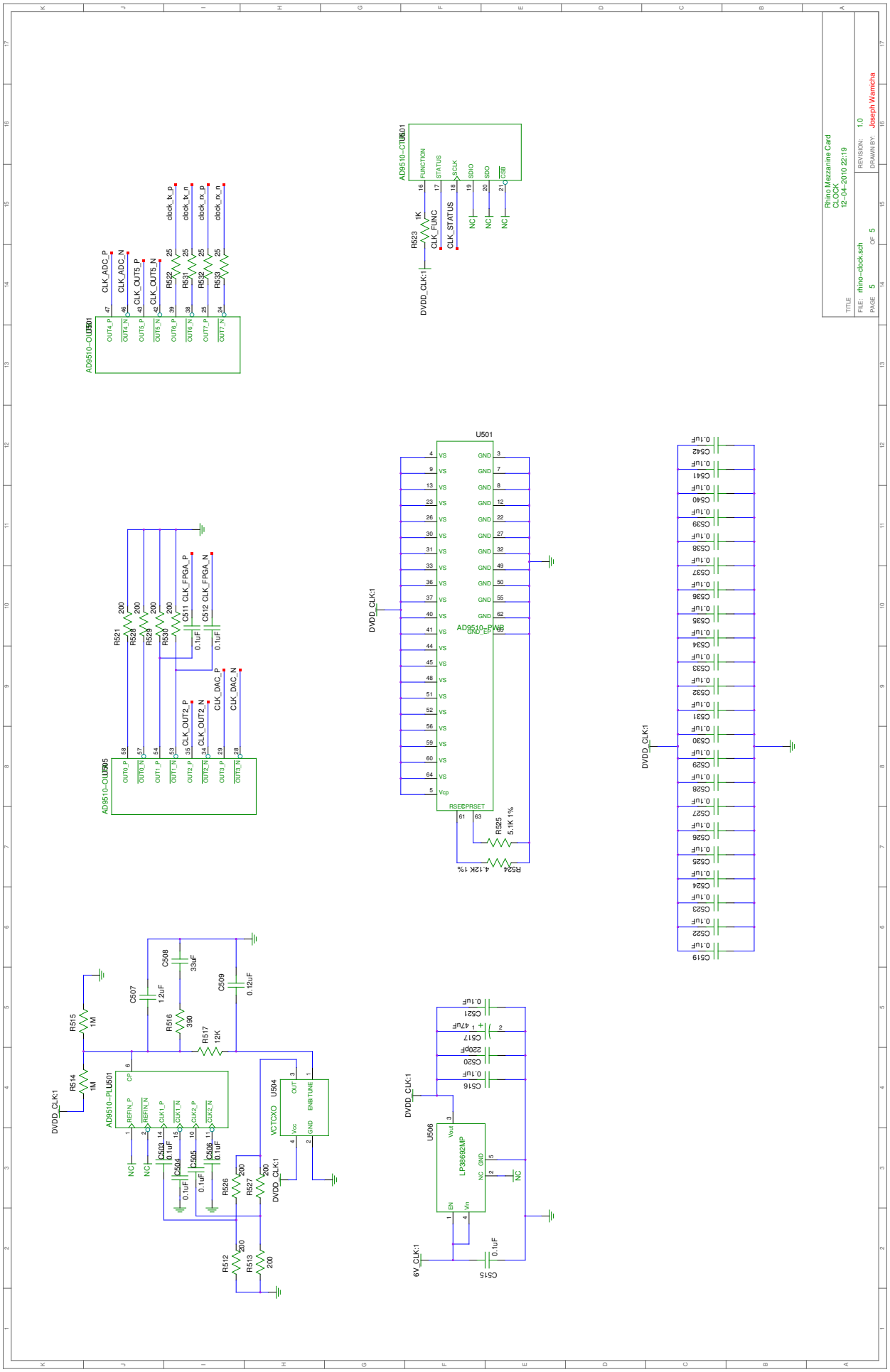


Figure 9.10: Rhino Expansion Board V2: Power

9.3.1 Bill of Materials

The Bill of Materials for the first iteration of the Rhino expansion board is shown in the next page:

Table 9.2: Rhino Expansion Board BOM: Iteration 2

R516	603	390	resistor	1
R517	603	12K	resistor	1
C509	603	0.12uF	capacitor	1
U501,U505	LFCSP64	unknown	Clock Divider and PLL	2
U506	SOT223-5	unknown	National Fixed LDO, 1A	1
C406,C408,C413,C418,C423,C427,C450,C517	1206	47uF	polarized capacitor	8
R523	603	1K	resistor	1
R524	603	4.12K 1%	resistor	1
R525	603	5.1K 1%	resistor	1
R512,R513,R521,R526,R527,R528,R529,R530	603	200	resistor	8
R522,R531,R532,R533	603	25	resistor	4
C403,C405,C410...C539,C540,C541,C542	603	0.1uF	capacitor	45

Chapter 10

Edge Fed EBG Microstrip Patch Antenna Generator Script

"None of the global warming discussions mention the word âĂIJnanotechnology.âĂİ Yet nanotechnology will eliminate the need for fossil fuels within 20 years. If we captured 1% of 1% of the sunlight (1 part in 10,000) we could meet 100% of our energy needs without ANY fossil fuels. We canâĂŹt do that today because the solar panels are too heavy, expensive, and inefficient" - Ray Kurzweil, the prophetic author of the book: "The Singularity is Near".

10.1 Overview

Due to their small form factors and conformability, microstrip patch antennas have become extremely popular for use inside embedded devices. Though they inherently have a high quality factor (Q), their bandwidth is very narrow, they experience high side lobe levels and their radiation patterns are susceptible to spurious surface wave interference from the surrounding substrate.[1, Pg 3]

A patch antenna generator script was written to help with rapid prototyping of edge fed EBG microstrip antennas, so that their characteristics can be studied more closely, and conclusions drawn. The generator script, mpatchebgcalc.py, spits out a gerber file by default which is used to fabricate the antenna using a photoplotter. The script is also able to generate a feko .pre script file that can be used to model the antenna characteristics and determine the optimum dimensions. Once the optimum dimensions have been determined using OptFeko, these dimensions can then be fed back into the generator script to regenerate the antenna gerber file. The generator scripts generates the antenna using the following arguments: desired resonant frequency, dielectric constant, substrate height, tangent loss, metallic patch conductivity, desired EBG columns, spacing between the EBG structures, name of the output gerber or feko file and the output file type (gerber or feko file).

Definitions:
$\bar{E}$ = The electric field intensity in volts per meter (V/m).
$\bar{B}$ = The magnetic flux density in webers per square meter (Wb/m^2).
$\bar{H}$ = The magnetic field intensity in volts per meter (A/m).
$\bar{D}$ = The electric flux density in coulombs per square meter (C/m^2).
$\bar{J}$ = The electric charge density in coulombs per cubic meter (C/m^3).
$\nabla\times$ = The curl operator.
$\nabla.$ = The divergence operator.
$\bar{D} = \epsilon_0\bar{E}$ in free space
$\bar{B} = \mu_0\bar{H}$ in free space
ϵ_0 = Permittivity of free space = $8.854 \times 10^{-12} F/m$
ϵ_r = Relative permittivity of substrate
ϵ_{eff} = Effective permittivity of substrate
μ_0 = Permeability of free space = $4\pi \times 10^{-7} H/m$
ρ = Charge density
L = Patch Length
L_{eff} = Effective Patch Length
ΔL = Patch Length extension as a result of the fringe fields at the edge of the patch, along it's length.
W = Patch Width
h = Height of dielectric substrate
c = Speed of light in free space
f_0 = Resonant frequency
λ_0 = Wavelength at resonant frequency

10.2 Maxwell's equations

Maxwell's equations for propagating EM fields are given as follows:

$$\begin{aligned} \nabla \times \bar{E} &= -j\omega\bar{B} \\ \nabla \times \bar{H} &= j\omega\bar{D} + \bar{J} \\ \nabla.\bar{D} &= \rho \\ \nabla \times \bar{B} &= 0 \end{aligned} \tag{10.1}$$

The rate of change of charge with time is related to the current flow by the following equation:

$$\nabla.\bar{J} = \frac{\delta\rho}{\delta t} \tag{10.2}$$

Some of the characteristics we consider during the design of a microstrip patch antenna are its radiation patterns, beamwidth, directivity, gain, polarization, input impedance, efficiency and bandwidth.[1, Pg 9]

10.3 Rectangular Microstrip Antenna Dimensions

[68, Page 30] Since they are very easy to fabricate, Rectangular Microstrip Antennas (RMSAs) are often used in an antenna design. An RMSA will have a length that would be slightly less than half of the desired resonant frequency wavelength. The width of the antenna is maximized as much as possible so as to enhance the RMSA's bandwidth. However, to prevent higher order TM modes from coming close to the fundamental resonant frequency TM_{10} mode, the width of the RMSA must obey the following rule:

$$L < W < 2L$$

So as to take into account the fringe fields at the edge of the patch, the *effective permittivity* was used for our RMSA calculations. The effective permittivity of an RMSA is given by the following equation:

$$\epsilon_{eff} = \frac{\epsilon_r + 1}{2} + \frac{\epsilon_r - 1}{2}(1 + \frac{10h}{W})^{-\frac{1}{2}} \tag{10.3}$$

eq. (10.3) was implemented using the following code:

```
    def get_eps_eff(self, eps_rel, h_sub, Width):
        int1 = (eps_rel + 1.0) / 2.0
        int2 = (eps_rel - 1.0) / 2.0   (math.pow((1 + (10.0   h_sub / Width)), -0.5))
        self.eps_eff = int1 + int2
        return self.eps_eff
```

Listing 10.1: Effective Permittivity (See lines 75-79 of the mpatchebgcalc.py file)

ΔL resulting from the fringe fields is given as [71, Page 4]:

$$\Delta L = h \times 0.412 \times (\frac{\epsilon_{eff} + 0.3}{\epsilon_{eff} - 0.258}) \times (\frac{\frac{W}{h} + 0.264}{\frac{W}{h} + 0.813}) \tag{10.4}$$

eq. (10.4) was implemented using the following code:

```
    def get_dL(self, h_sub, eps_eff, Width):
        dL_1 = 0.412   h_sub   ((eps_eff + 0.3)/(eps_eff - 0.258))
        dL_2 = (Width/h_sub + 0.264) / (Width/h_sub + 0.813)
        self.dL = dL_1   dL_2
        return self.dL
```

Listing 10.2: dL (See lines 97-101 in the mpatchebgcalc.py file)

The actual length of the patch is given as:

$$L = L_{eff} - (2 \times \Delta L) \tag{10.5}$$

eq. (10.5) was implemented using the following code:

```
    def get_Len(self, L_eff, dL):
        self.Len = L_eff - (2    dL)
        return self.Len
```

Listing 10.3: Metallic Patch Length (See lines 103-105 of the mpatchebgcalc.py file)

The effective length of the patch is given as:

$$\begin{aligned} L_{eff} &= L + (2 \times \Delta L) \\ &= \frac{\lambda_0}{2\sqrt{\epsilon_{eff}}} \\ &= \frac{c}{2f_0\sqrt{\epsilon_{eff}}} \end{aligned} \tag{10.6}$$

eq. (10.6) was implemented using the following code:

```
    def get_Leff(self, f0, eps_eff):
2       self.Leff = spconst.c / (2    f0    math.sqrt(eps_eff))
        return self.Leff
```

Listing 10.4: Effective Patch Length (See lines 103-105 of the mpatchebgcalc.py file)

Thus, the resonant frequency while using the patch antenna is given by the following equation:

$$f_0 = \frac{c}{2\sqrt{\epsilon_{eff}}} \frac{1}{L_{eff}} \tag{10.7}$$

10.4 EBG Dimensions

EBG structures are artificial, well ordered, periodic or non-periodic structures made out of either dielectric or metallic materials. When the ordered EBG structures are laid onto a material, a new artifical meta-material is formed, which acquires new properties that are not normally found in nature. EBG structures ordered on the substrate of a patch for example, will prevent propagation of unwanted surface waves on the patch at the stop gap frequency. The new meta materials may exhibit new properties such as negative refractive index and near perfect magnetic conduction[112, Pg 2].

Definitions:
c = speed of light in free space

$m = TM_{m0}$ resonant mode
$n = TM_{0n}$ resonant mode
$a =$ length of the waveguide
$b =$ width of the waveguide
$\epsilon_r =$ relative permittivity of substrate
$\mu_r =$ relative permeability of substrate $= 1$
$N =$ total number of EBG structures
$ebg_{len} =$ EBG Length
$ebg_{width} = ebg_{len} =$ EBG Width
$ebg_{gap} =$ gap between each of the EBG structures
$ebg_{totalgap} =$ sum of all the gaps between the EBG structures
$feed_{len} =$ inset feed length $feed_{width} =$ inset feed width

For our example patch antennas, the EBG structures were designed to stop surface wave propagation within a certain stop frequency band. In order to determine the dimensions of the EBG structures, assume that the EBG structures are spaced so closely together that they form a waveguide capable of propagating an EM wave. For a standard rectangular plate waveguide, the waveguide's resonance frequency modes are given by the following equation:

$$f_{mn} \approx \frac{c}{2\pi\sqrt{\mu_r\epsilon_r}}\sqrt{(\frac{m\pi}{a})^2+(\frac{n\pi}{b})^2} \tag{10.8}$$

If only TM_{m0} waves are only propagated along the length of the waveguide, then b is equal to zero and the lower frequency of the EBG stop band gap would be given by the following equation:

$$\begin{aligned} f_{low} &= \frac{c}{2\pi\sqrt{\mu_r\epsilon_r}}\sqrt{(\frac{m\pi}{a})^2} \\ &= \frac{c}{2\pi\sqrt{\mu_r\epsilon_r}}(\frac{m\pi}{a}) \\ &= \frac{c}{2\sqrt{\epsilon_r}}(\frac{m}{a}) \\ &= \frac{c}{2\sqrt{\epsilon_{eff}}}(\frac{m}{a}) \end{aligned} \tag{10.9}$$

ϵ_{eff} is given by eq. (10.3) and is used to ensure more accurate frequency calculation results, since it takes into consideration the effect of fringe fields induced at the edge of a metallic patch. If the waveguide only had a single order propagation mode along its length, TM_{10}, then eq. (10.9) becomes:

$$f_{low} = f_{(1,0)} \quad = \frac{c}{2\sqrt{\epsilon_{eff}}}(\frac{1}{a}) \tag{10.10}$$

The equation eq. (10.10) is the same as the equation used to determine the patch length, eq. (10.7). If such

a waveguide were to be fabricated onto the substrate, it would also be capable of transmitting at the same frequency as the metallic patch. To ensure that this does not happen, we create discontinuities on the waveguide so that rather than have a singular long waveguide, we have several periodic metallic patches forming the *virtual* waveguide. These periodic patches form our EBG structures and are arranged along the length of the patch. The EBG structures are ordered around the resonant frequency of the patch, but have discontinuities introduced between them. As a result of the discontinuities between the patches, the resistance to TM wave propagation at patch resonant frequency becomes so high (due to the impedance mismatch between the EBG structures), that the transverse magnetic waves are unable to induce surface waves on the substrate, hence contributing to the patch antenna's improved directivity and gain at the resonant frequency. The directivity and gain is improved since no less power is now lost to induced surface waves on the dielectric substrate after adding the EBG structures.

If the waveguide were composed of N contiguous metallic patches, then it would be capable of transmitting upto N order modes. Thus, if our patch antenna was surrounded by N EBG patches along its length, the low frequency threshold equation in eq. (10.10) would become:

$$\begin{aligned} f_{low} &= f_{(N-1,0)} \\ &= \frac{c}{2\sqrt{\epsilon_{eff}}}\left(\frac{N-1}{L_{eff}}\right) \end{aligned} \tag{10.11}$$

f_{low} in eq. (10.11) is the frequency above which, all $TM_{(N-1,0)}$ waves modes are stopped from propagating along the surface of the substrate. Our desired lower stop gap frequency threshold would be the Nth propagation mode along the length of this waveguide, where N is the total number of EBG structures used. The EBG patch length would thus be given by the following equation:

$$ebg_{len} = \frac{L_{eff} - (ebg_{gap} \times (N+1))}{N} \tag{10.12}$$

eq. (10.12) was implemented using the following code:

```
    def get_ebg_len(self, Len, Leff, ebg_N):
        ebg_total_gap = (Leff - Len)
        #((ebg_N + 1)   discontinuity) + (ebg_N   ebg_len) = Leff
        # ebg_total_gap + (ebg_N   ebg_len) = Leff
        if ebg_N == 0:
            return 0

        self.ebg_len = (Leff - ebg_total_gap) / ebg_N
        self.ebg_width = self.ebg_len
        return self.ebg_len
```

Listing 10.5: EBG Length (See lines 217-228 of the mpatchebgcalc.py file)

The N^{th} order virtual waveguide propation mode for the desired low frequency threshold stop gap will

therefore depend on the number of EBG structures used. The ebg_{gap} is given by the following equation:

$$ebg_{gap} = \frac{ebg_{totalgap}}{N+1} = \frac{2 \times \Delta L}{N+1} \tag{10.13}$$

eq. (10.13) was implemented using the following code:

```
    def get_ebg_gap(self, Len, Leff, ebg_N):
        ebg_total_gap = (Leff - Len)
        self.ebg_gap = (ebg_total_gap / (ebg_N + 1))
        return self.ebg_gap
```

Listing 10.6: EBG Gap (See lines 103-105 of the mpatchebgcalc.py file)

The higher frequency stop gap threshold is given by the following equation:

$$\begin{aligned} f_{high} &= f_{(0,n)} \\ &= \frac{c}{2\pi\sqrt{\mu_r \epsilon_r}} \sqrt{(\frac{n\pi}{a})^2} \end{aligned} \tag{10.14}$$

However, since we are only interested in limiting $TM_{N-1,0}$ modes propagating on the substrate surface along the length of the patch, the high frequency stop gap is not taken into consideration when fabricating our single band patch antenna.

10.5 Inset Feed Dimensions for a 50 Ohm Matched Input Impedance

Where $2 < \epsilon_r < 10$, the inset feed length to the edge fed microstrip patch antenna is given by the following equation [88]:

$$feed_{len} = 10^{-4}\{0.001699\epsilon_r^7 + 0.13761\epsilon_r^6 - 6.1783\epsilon_r^5 + 93.187\epsilon_r^4 - 682.69\epsilon_r^3 + 2561.9\epsilon_r^2 - 4043\epsilon_r + 6697\} \times \frac{L}{2} \tag{10.15}$$

eq. (10.15) was implemented using the following code:

```
1       def get_inset_feed_length(self, eps_rel, Len):
            fl1 = (0.001699    math.pow(eps_rel, 7)) + (0.13761    math.pow(eps_rel, 6)) + \\
                  (6.1783    math.pow(eps_rel, 5)    -1) + (93.187    math.pow(eps_rel, 4)) + \\
                  (682.69    math.pow(eps_rel, 3)    -1) + (2561.9    math.pow(eps_rel, 2)) + \\
                  (4043    eps_rel    -1) + 6697
6           self.fl = fl1    math.pow(10, -4)    Len/2
            return self.fl
```

Listing 10.7: Inset Feed Length (See lines 194-197 of the mpatchebgcalc.py file)

Any feed notch width can be used for the inset. However, the feed notch width should preferrably not be less than 1mm so as to provide enough space on which to solder an SMA edge connector.

Chapter 11

Simulation, Results and Analysis of the Fabricated Microstrip Patch Antennas

"And that because the moving parts are a million times smaller than the ones we're familiar with, they move a million times faster, just as a smaller tuning fork produces a higher pitch than a large one." - Eric K. Drexler, the prophetic author of the book: "Engines of Creation 2.0: The Coming Era of Nanotechnology".

11.1 Overview

This chapter presents the directivity and bandwidth results of our edge fed microstrip patch antennas with EBG structures. From the results, it was observed that using EBG structures improved directivity by between 2 and 3 dB. In addition, wider spaced EBG structures seemed to improve directivity even better. The directivity measurements were taken using the Agilent 5071B Network Analyzer. The substrate material used to fabricate the patch antennas was the R04003C substrate from Rogers, which had the following properties:

- Dielectric constant of 3.38
- Substrate height of 32 mils (0.813mm)
- Tangential loss of 2.7×10^{-3}
- The patch conductivity of the copper patch was 5.69×10^{7} S/m

11.2 Gerber file generation and results

11.2.1 Patch 1: 0 EBGs

The first edge fed microstrip patch antenna to be fabricated was generated with the following mpatchebgcalc.py command:

```
$./mpatchebgcalc.py -f 2.4e9 -d 3.38 --substrate-height 8.13e-3 --tangent-loss 2.7e-3 --patch-conductivity \\
5.69e7 --\gls{ebg}-cols 0 --minimum-discontinuity 1e-3 --out-file mspebg --filetype gerber
```

The resulting patch antenna can be seen in the diagram below:

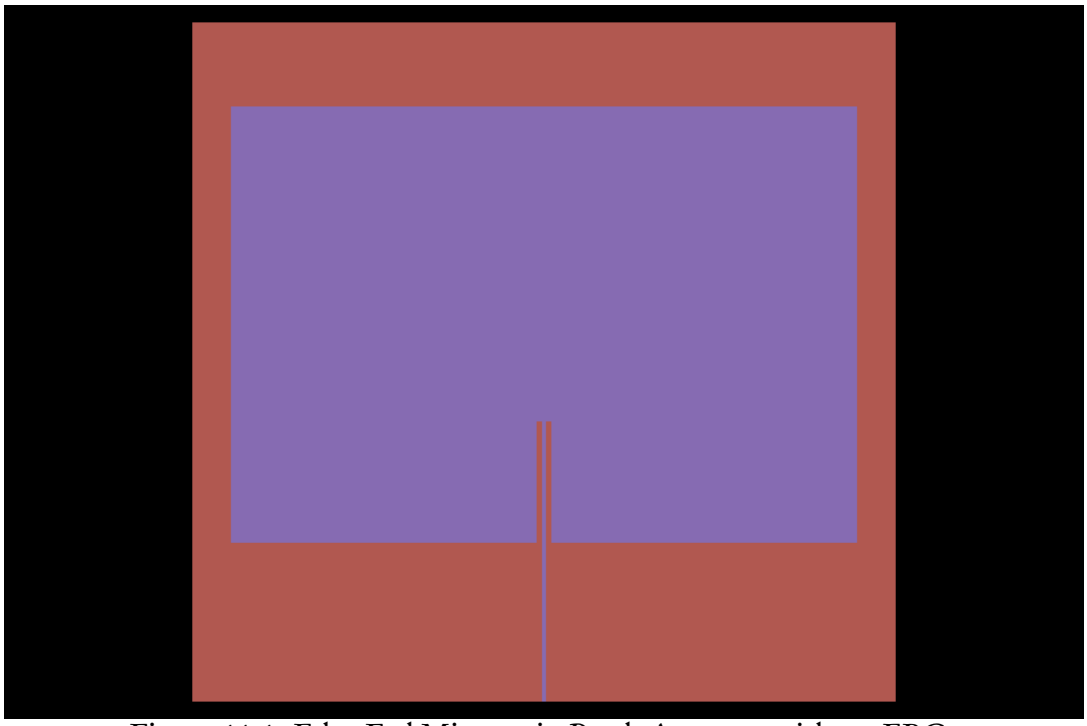


Figure 11.1: Edge Fed Microstrip Patch Antenna with no EBGs

The directivity and bandwidth results are shown in the diagram below:

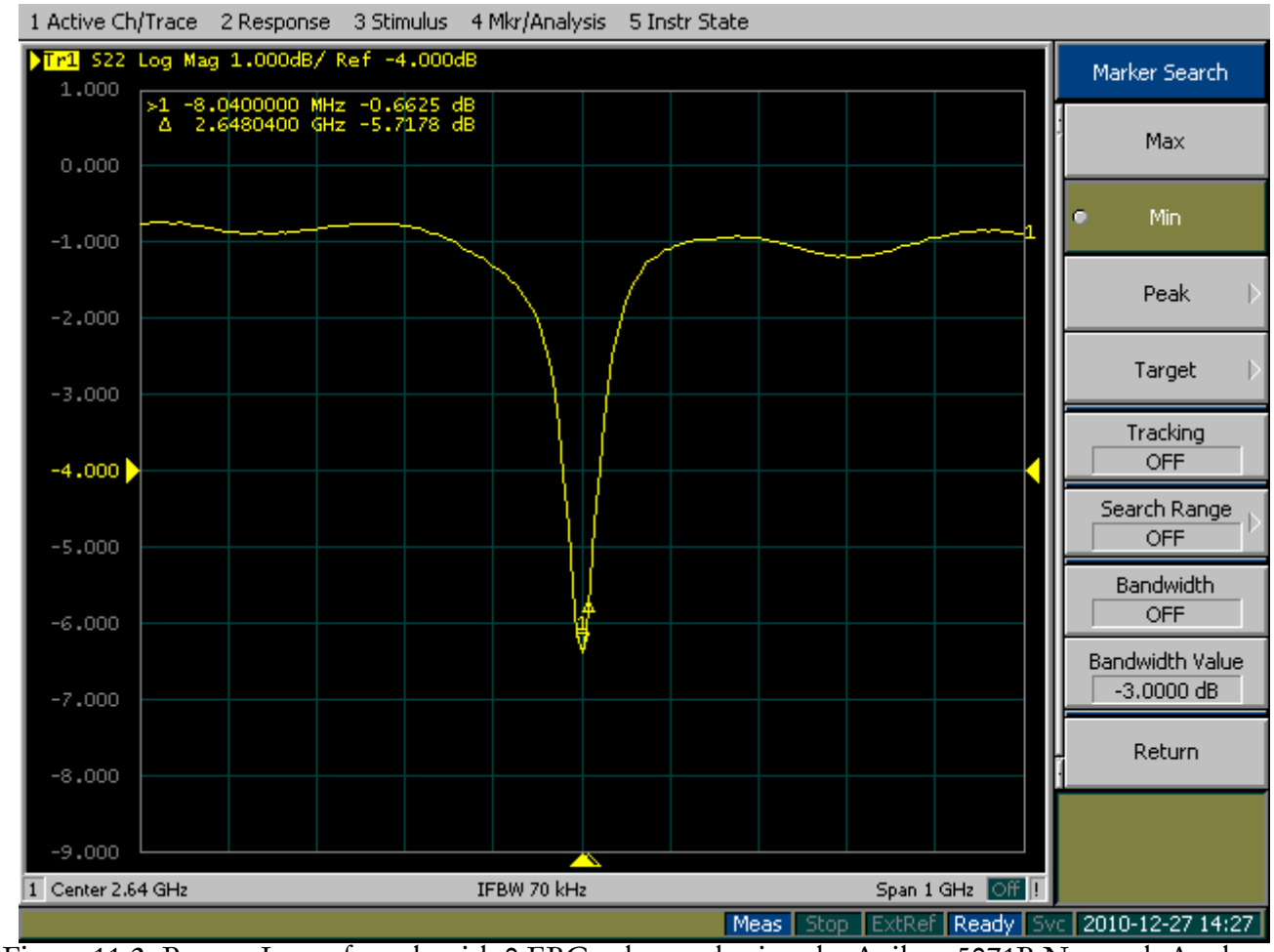


Figure 11.2: Return Loss of patch with 0 EBGs observed using the Agilent 5071B Network Analyzer

The directivity and bandwidth results are shown in the plot below:

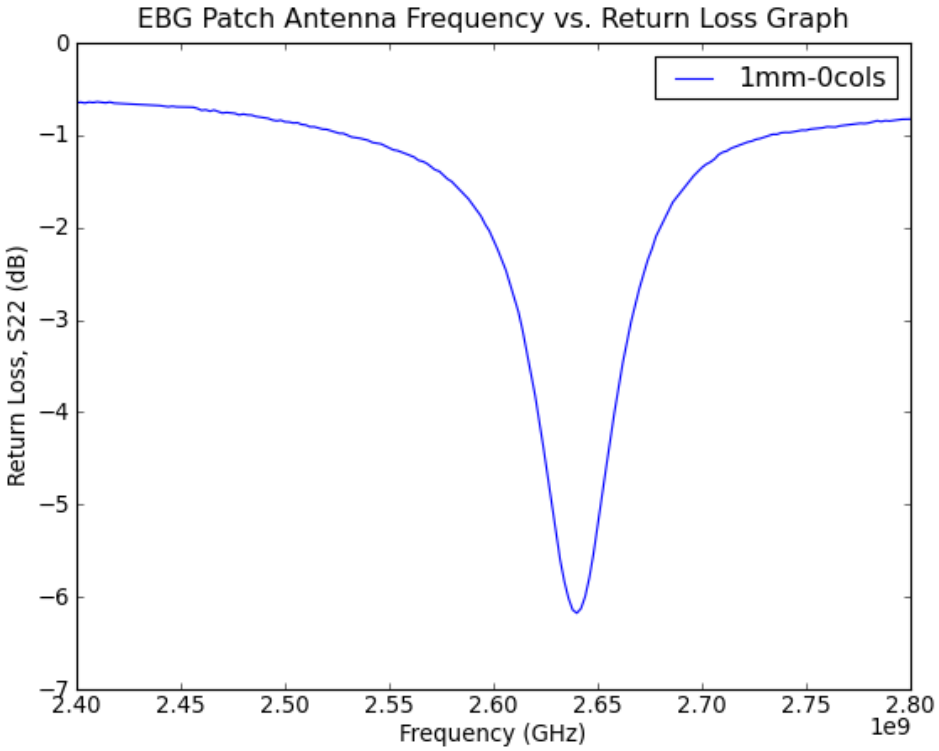


Figure 11.3: Return Loss Plot of patch with 0 EBGs

11.2.2 Patch 2: 2 EBG columns with 1mm spacing

The second edge fed microstrip patch antenna to be fabricated was generated with the following mpatchebg-calc.py command:

```
$./mpatchebgcalc.py -f 2.4e9 -d 3.38 --substrate-height 8.13e-3 --tangent-loss 2.7e-3 --patch-conductivity \\
5.69e7 --\gls{ebg}-cols 2 --minimum-discontinuity 1e-3 --out-file mspebg --filetype gerber
```

The resulting patch antenna can be seen in the diagram below:

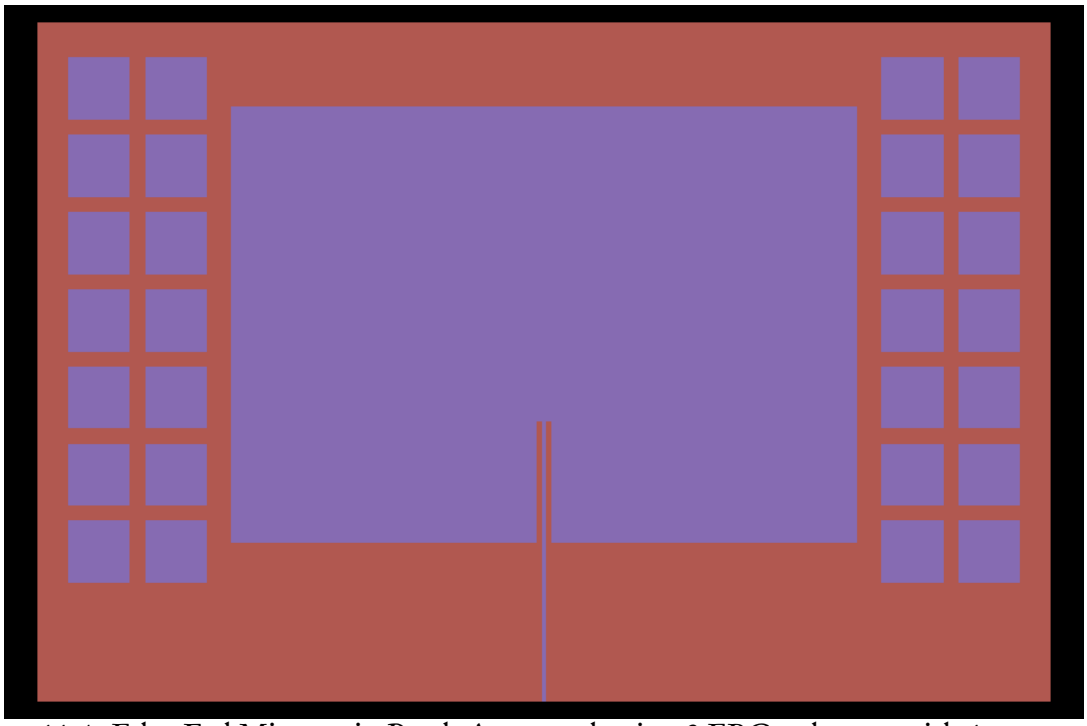


Figure 11.4: Edge Fed Microstrip Patch Antenna having 2 EBG columns with 1mm spacing

The directivity and bandwidth results are shown in the diagram below:

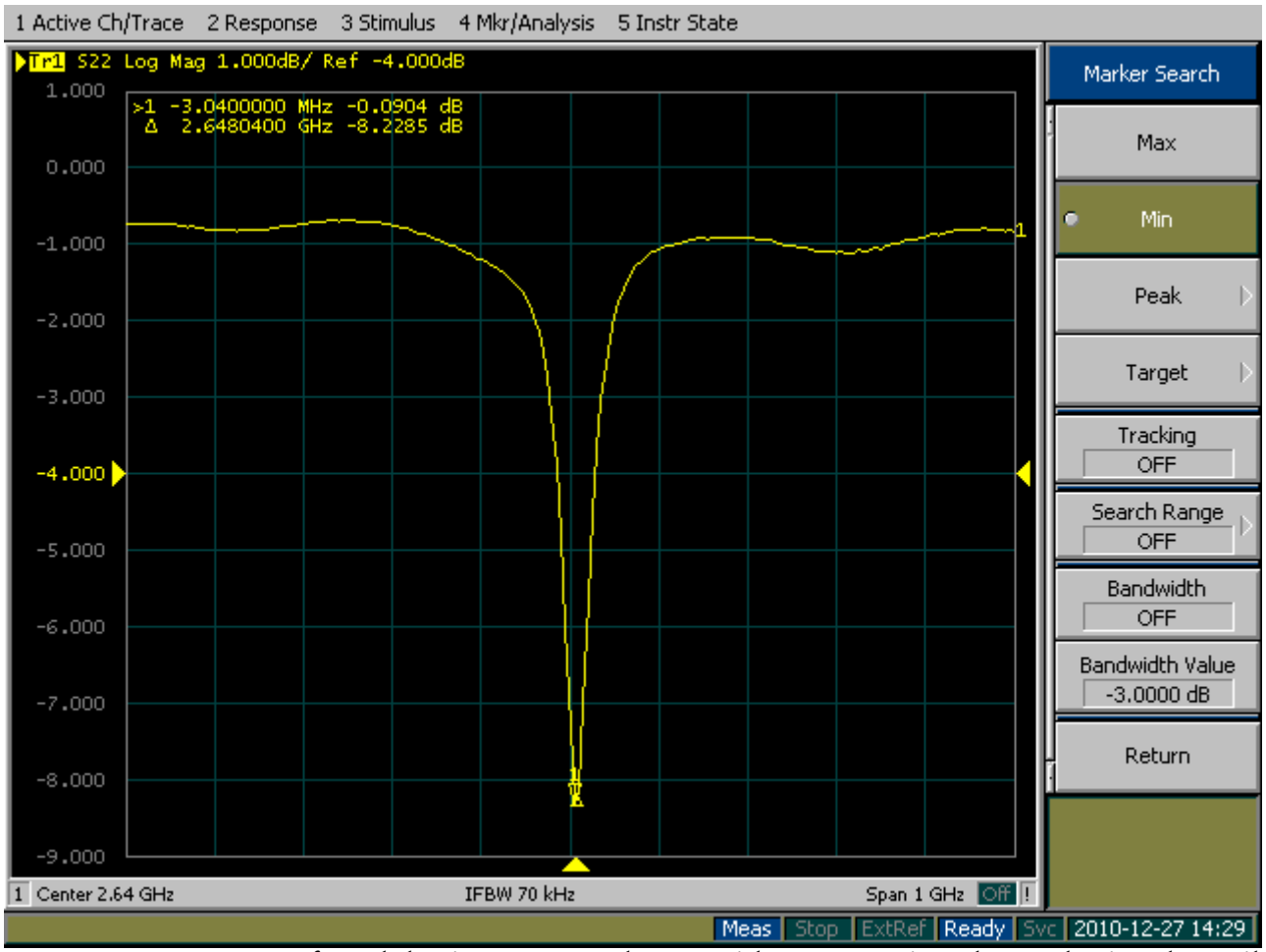


Figure 11.5: Return Loss of patch having 2 EBG columns with 1mm spacing observed using the Agilent 5071B Network Analyzer

The directivity and bandwidth results are shown in the plot below:

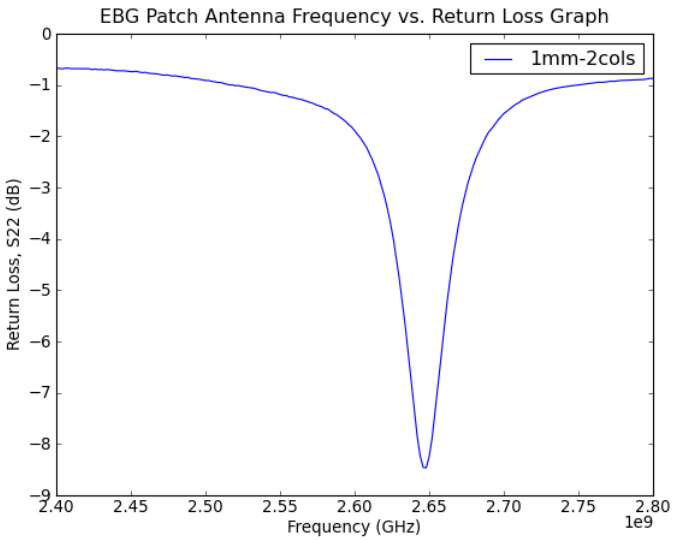


Figure 11.6: Return Loss Plot of patch having 2 EBG columns with 1mm spacing

11.2.3 Patch 3: 4 EBG columns with 1mm spacing

The third edge fed microstrip patch antenna to be fabricated was generated with the following mpatchebg-calc.py command:

```
$./mpatchebgcalc.py -f 2.4e9 -d 3.38 --substrate-height 8.13e-3 --tangent-loss 2.7e-3 --patch-conductivity \\
5.69e7 --\gls{ebg}-cols 4 --minimum-discontinuity 1e-3 --out-file mspebg --filetype gerber
```

The resulting patch antenna can be seen in the diagram below:

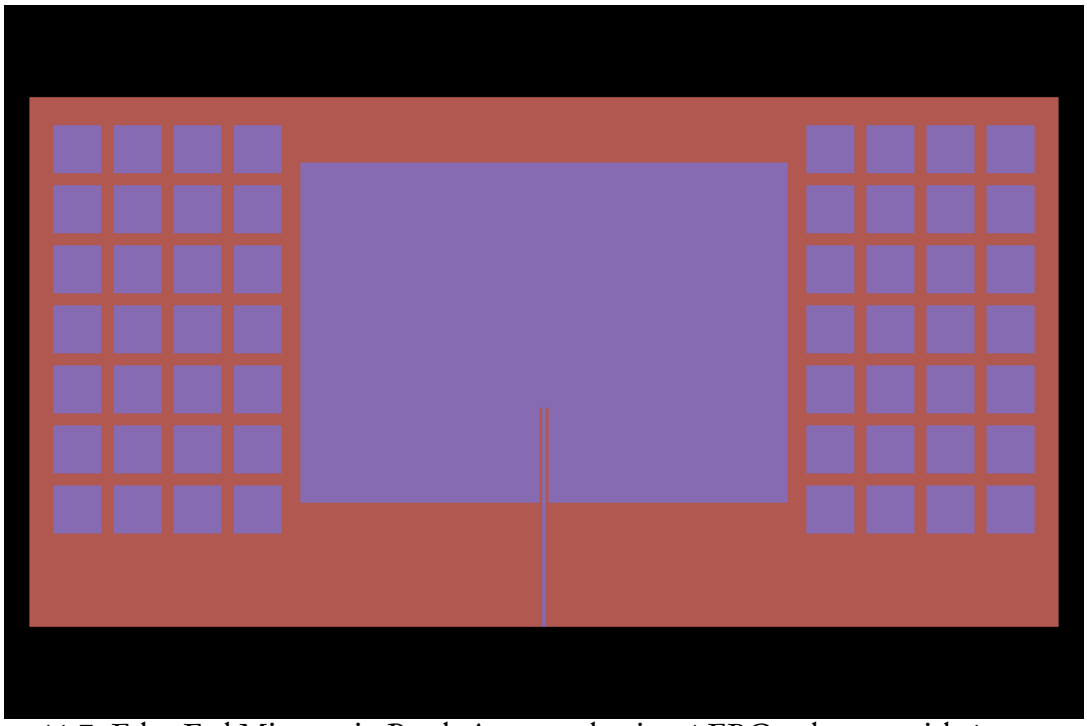


Figure 11.7: Edge Fed Microstrip Patch Antenna having 4 EBG columns with 1mm spacing

The directivity and bandwidth results are shown in the diagram below:

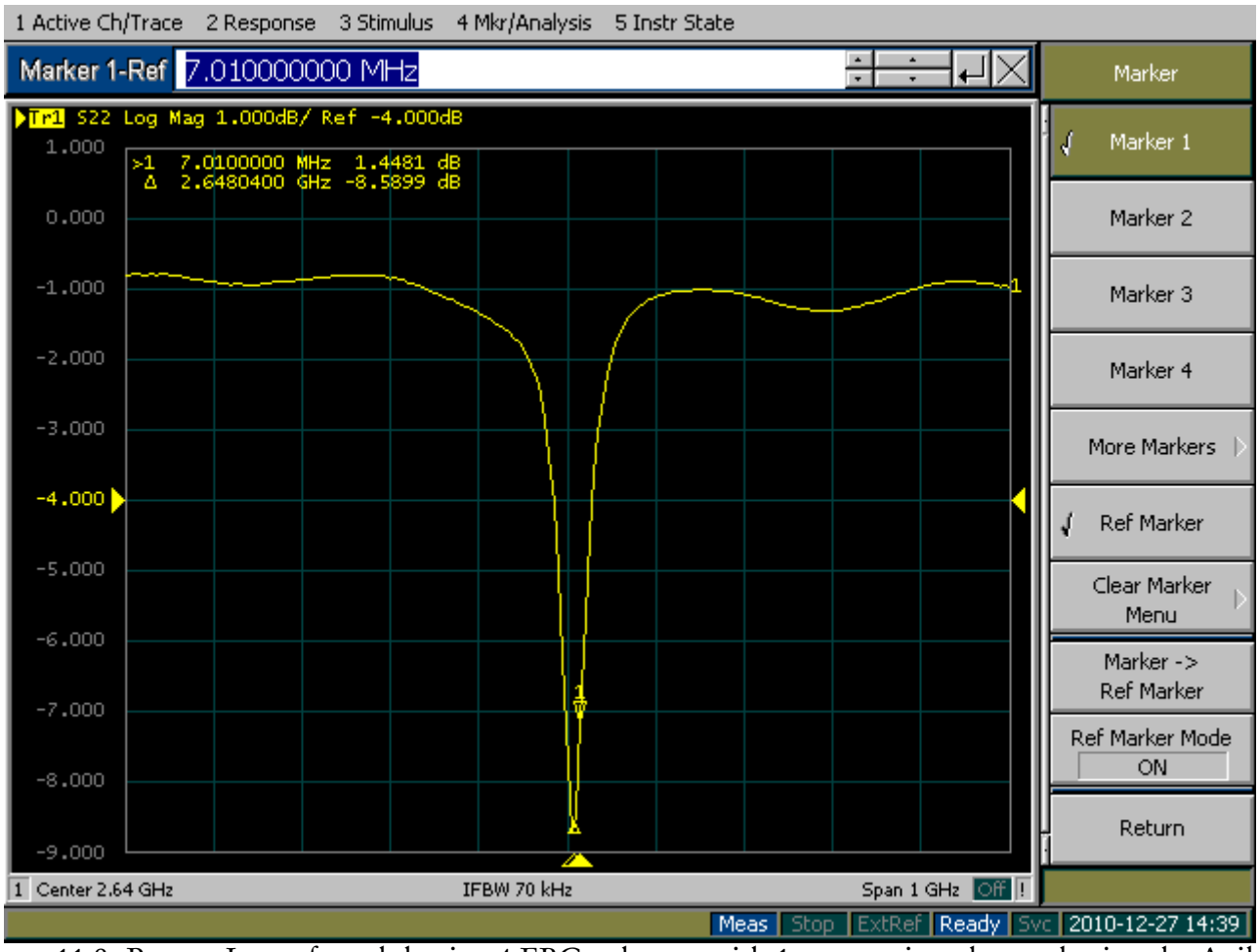


Figure 11.8: Return Loss of patch having 4 EBG columns with 1mm spacing observed using the Agilent 5071B Network Analyzer

The directivity and bandwidth results are shown in the plot below:

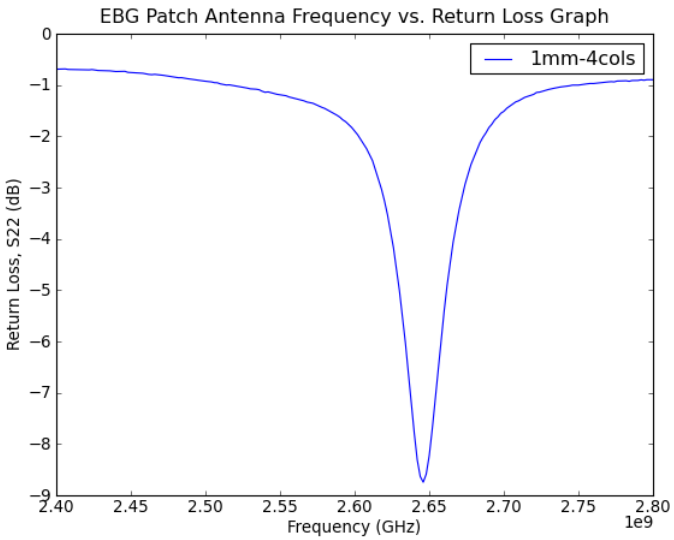


Figure 11.9: Return Loss Plot of patch having 4 EBG columns with 1mm spacing

11.2.4 Patch 4: 4 EBG columns with 0.5mm spacing

The fourth edge fed microstrip patch antenna to be fabricated was generated with the following mpatchebg-calc.py command:

```
$./mpatchebgcalc.py -f 2.4e9 -d 3.38 --substrate-height 8.13e-3 --tangent-loss 2.7e-3 --patch-conductivity \\
5.69e7 --\gls{ebg}-cols 4 --minimum-discontinuity 0.5e-3 --out-file mspebg --filetype gerber
```

The resulting patch antenna can be seen in the diagram below:

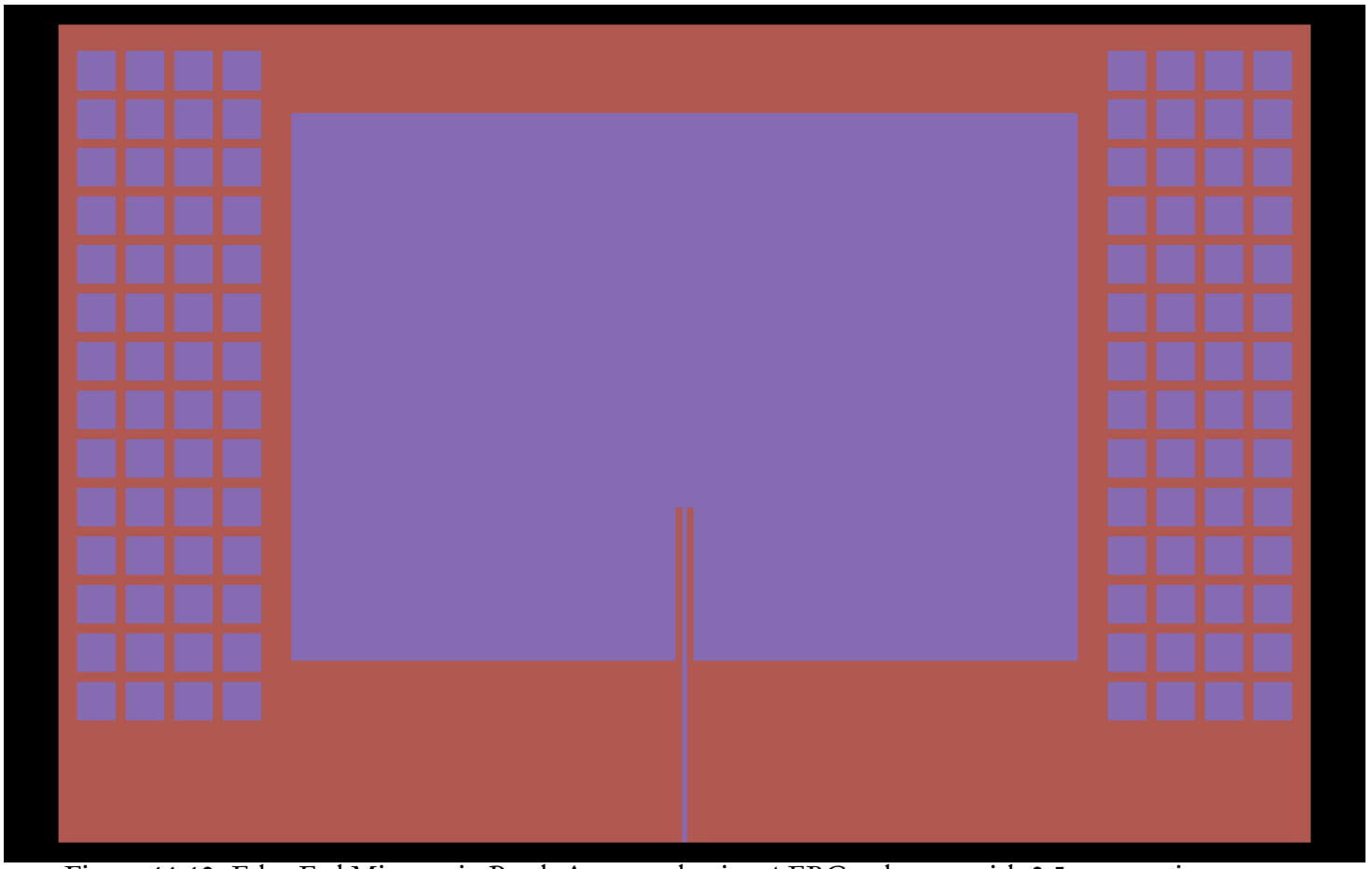


Figure 11.10: Edge Fed Microstrip Patch Antenna having 4 EBG columns with 0.5mm spacing

The directivity and bandwidth results are shown in the diagram below:

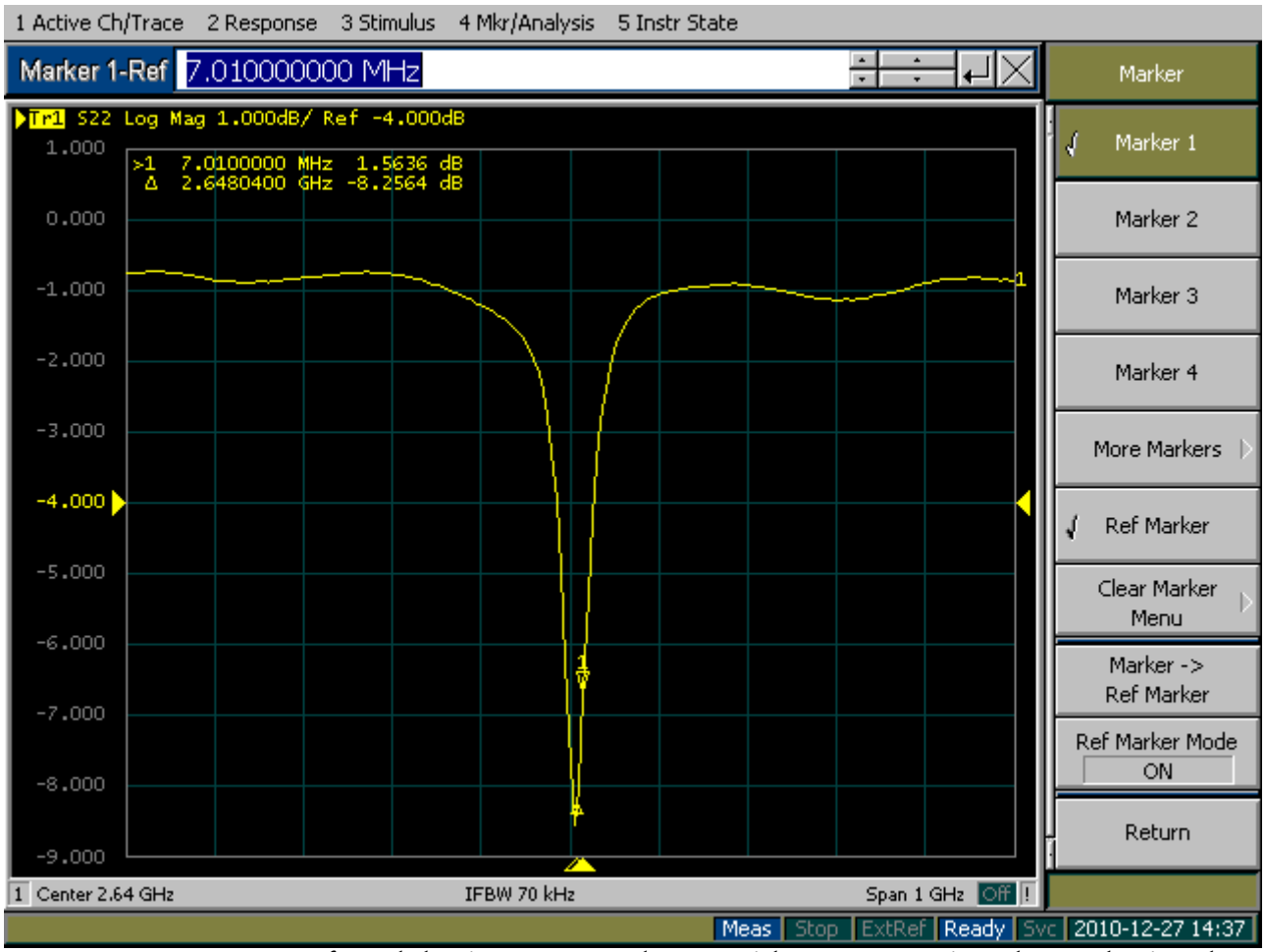


Figure 11.11: Return Loss of patch having 4 EBG columns with 0.5mm spacing observed using the Agilent 5071B Network Analyzer

The directivity and bandwidth results are shown in the plot below:

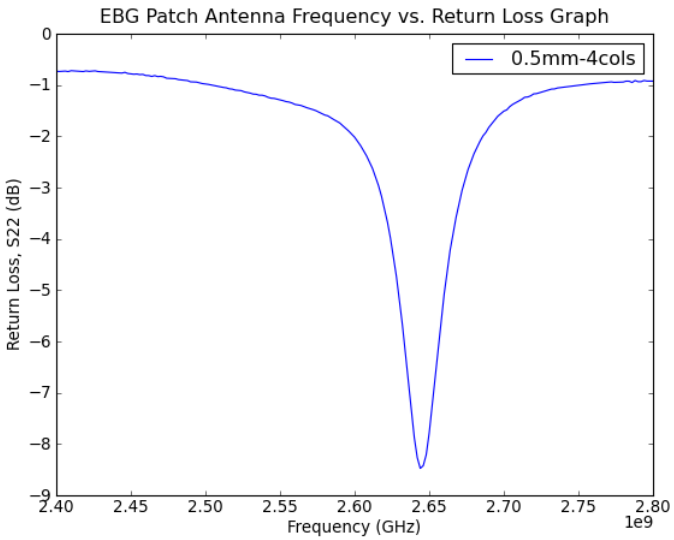


Figure 11.12: Return Loss Plot of patch having 4 EBG columns with 0.5mm spacing

11.3 Feko file generation and results

A feko *.pre file that was used to model the antenna was generated using the following command:

```
$./mpatchebgcalc.py -f 2.4e9 -d 3.38 --substrate-height 8.13e-3 --tangent-loss 2.7e-3 --patch-conductivity \\
5.69e7 --\gls{ebg}-cols 0 --minimum-discontinuity 1e-3 --out-file mspebg --filetype feko
```

This outputs the file *mspebg.pre* which can then be opened from within editfeko using the following command:

```
$editfeko mspebg.pre
```

This will open the editfeko GUI editor. From within the editor navigate to the *Run* menu and click on *PREFEKO* to generate a meshing solution for our patch antenna. Once the mesh solution has been successfully generated, navigate to the *Run* menu and click on *FEKO* to do perform impedance and far field simulation calculations of our antenna. Once *FEKO* has successfully run, an mspebg.fek file will be generated. Run *POSTFEKO* to see the fa field and impedance calculations as a result of our simulation.

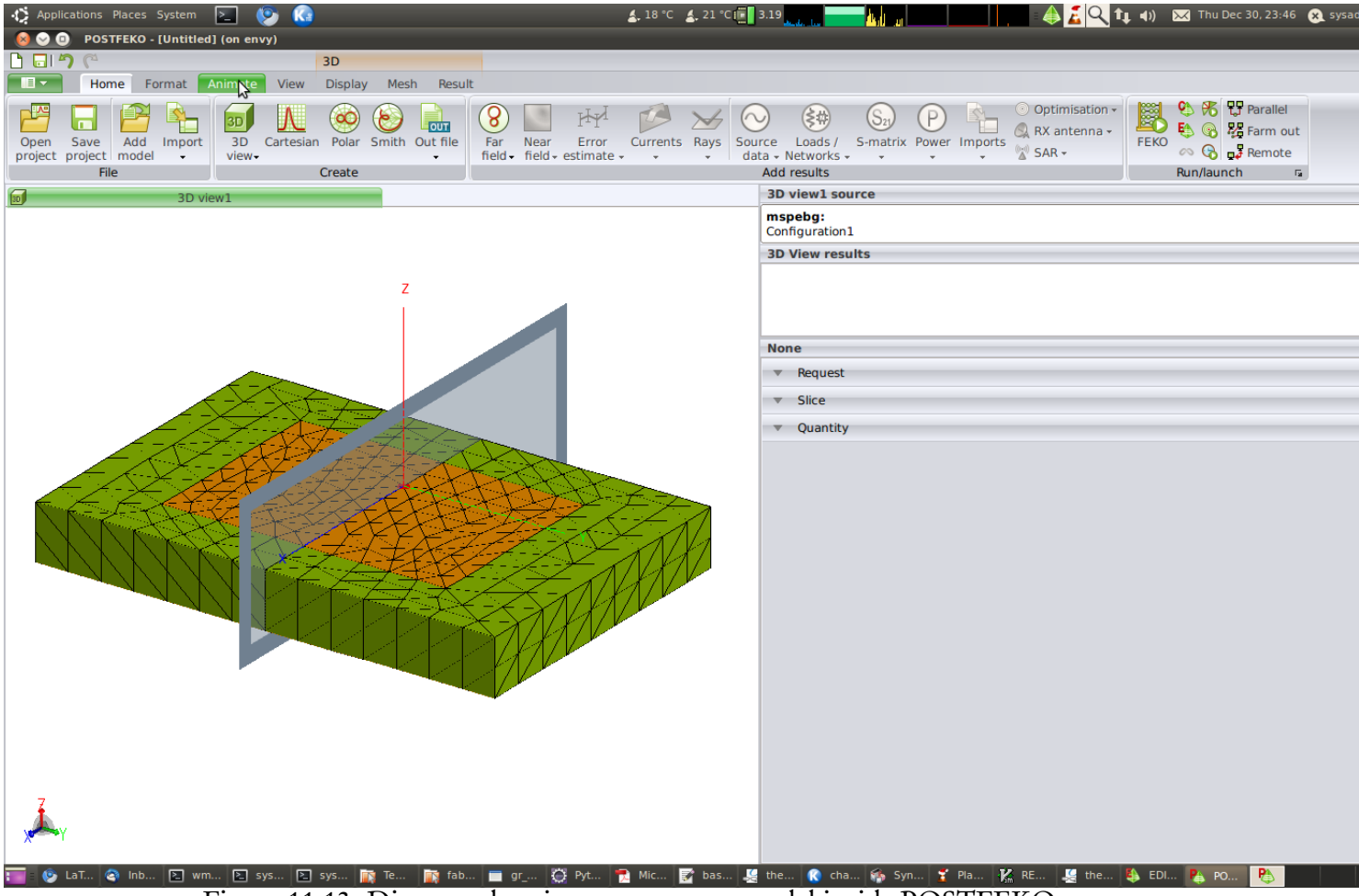


Figure 11.13: Diagram showing our antenna model inside POSTFEKO

The mspebg.fek file can be imported into cadfeko navigating to File, Import, Mesh, FEKO model (*.fek) as shown in the diagram below.

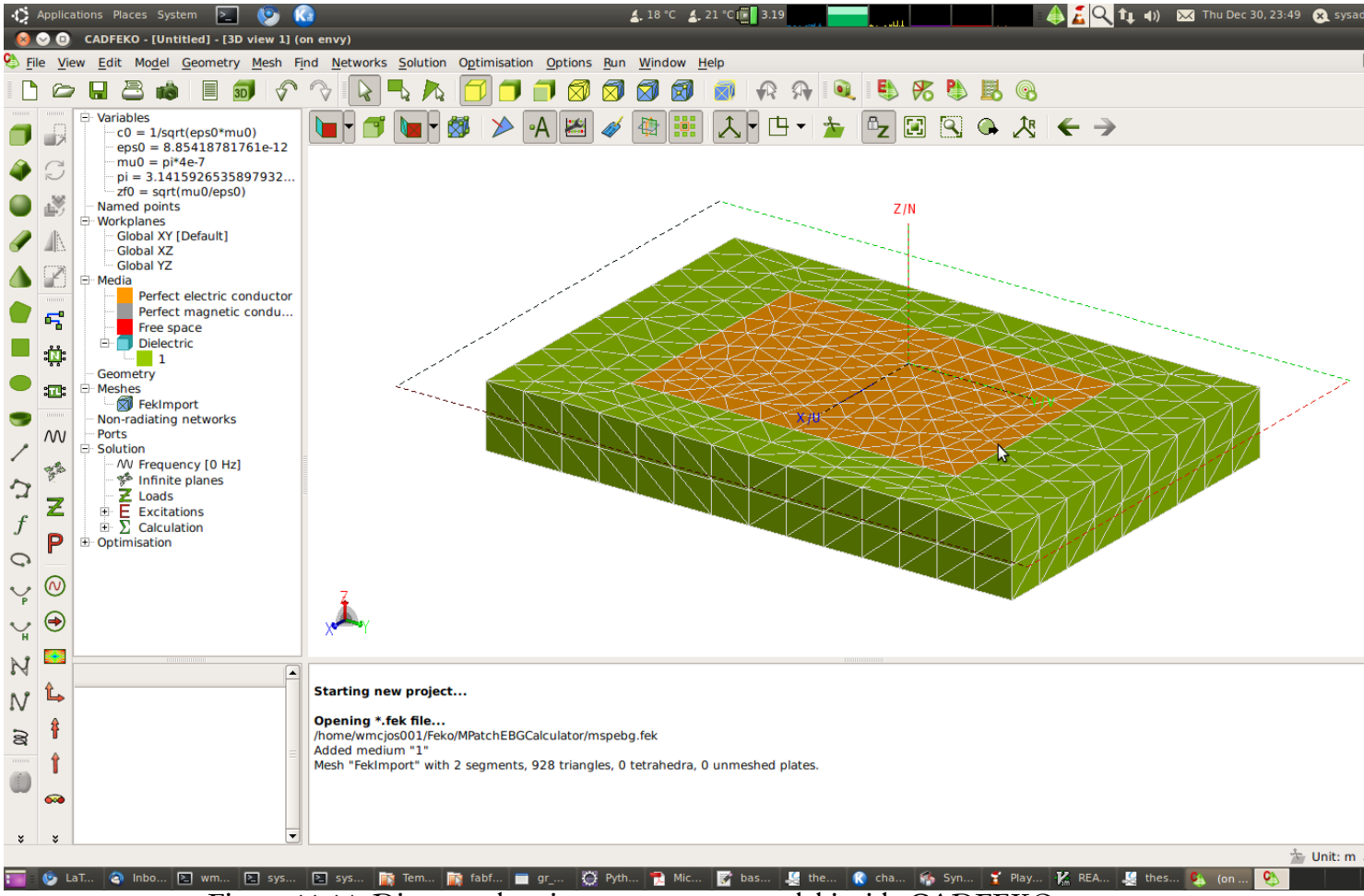


Figure 11.14: Diagram showing our antenna model inside CADFEKO

11.3.1 Feko shortcomings

During our attempt to model the EBG antenna, we discovered that the DP geometry card was unable to accept more than 26 geometry points (which represent the letters of the alphabet). Due to this limitation, we were therefore unable to include the EBG structures in the autogenerated mesh model. The EBG structures can only be modelled by drawing them manually, and can not be fed into feko using their scripting language.

We also discovered that though a Feko .fek file can be imported into the CADFEKO environment, it does not import the antenna variables along with it so that the patch antenna dimensions can not be optimised using *OPTFEKO*.

11.3.2 Summary of Results

Even though we specified that our patch antennas should resonate at 2.4 GHz, the actual resonant frequency was 2.648 GHz, which was 148 MHz off our desired resonant frequency. The mpatchebgcalc.py script will need to be refactored so as to allow better convergence onto the desired frequency. The final input impedance was 30Ω instead of the desired 50Ω. The mpatchebgcalc.py script will therefore need to be refactored so as to allow better convergence onto the desired input impedance value.

However, the main objective of our experiment was achieved by fabricating EBG structures that prevent surface wave propagation at the target frequency and hence improve the directivity of the patch antenna, without losing any bandwidth. The results are summarized in the table below:

Table 11.1: Fabricated EBG Patch Antennas: Directivity and Bandwidth Results

Patch	EBG Gap	EBG Len	EBG Columns	Frequency	Return Loss	BW
1	0	0	0	2.648 GHz	-5.7 dB	27 MHz
2	1mm	4.198mm	2	2.648 GHz	-8.24 dB	18 MHz
3	1mm	4.198mm	4	2.648 GHz	-8.52 dB	17 MHz
4	0.5mm	2.099mm	4	2.648 GHz	-8.35 dB	18 MHz

The results for all the fabricated patches are summarized in the graph plots below:

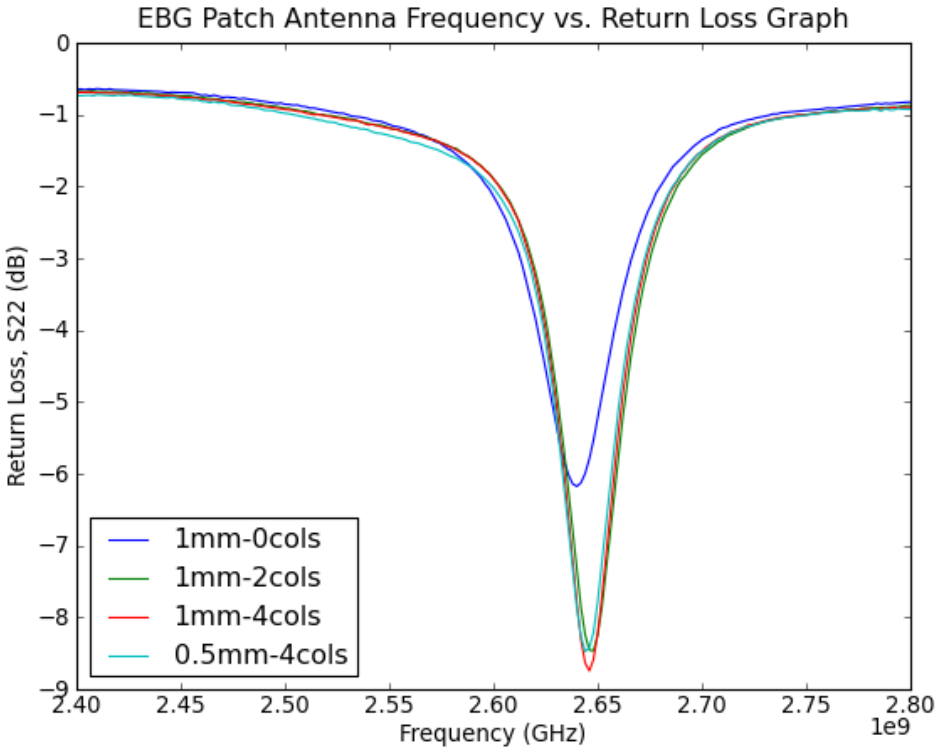


Figure 11.15: Return Loss Plots of all the Antennas

The fabricated edge fed patch antennas are shown in the photo below:

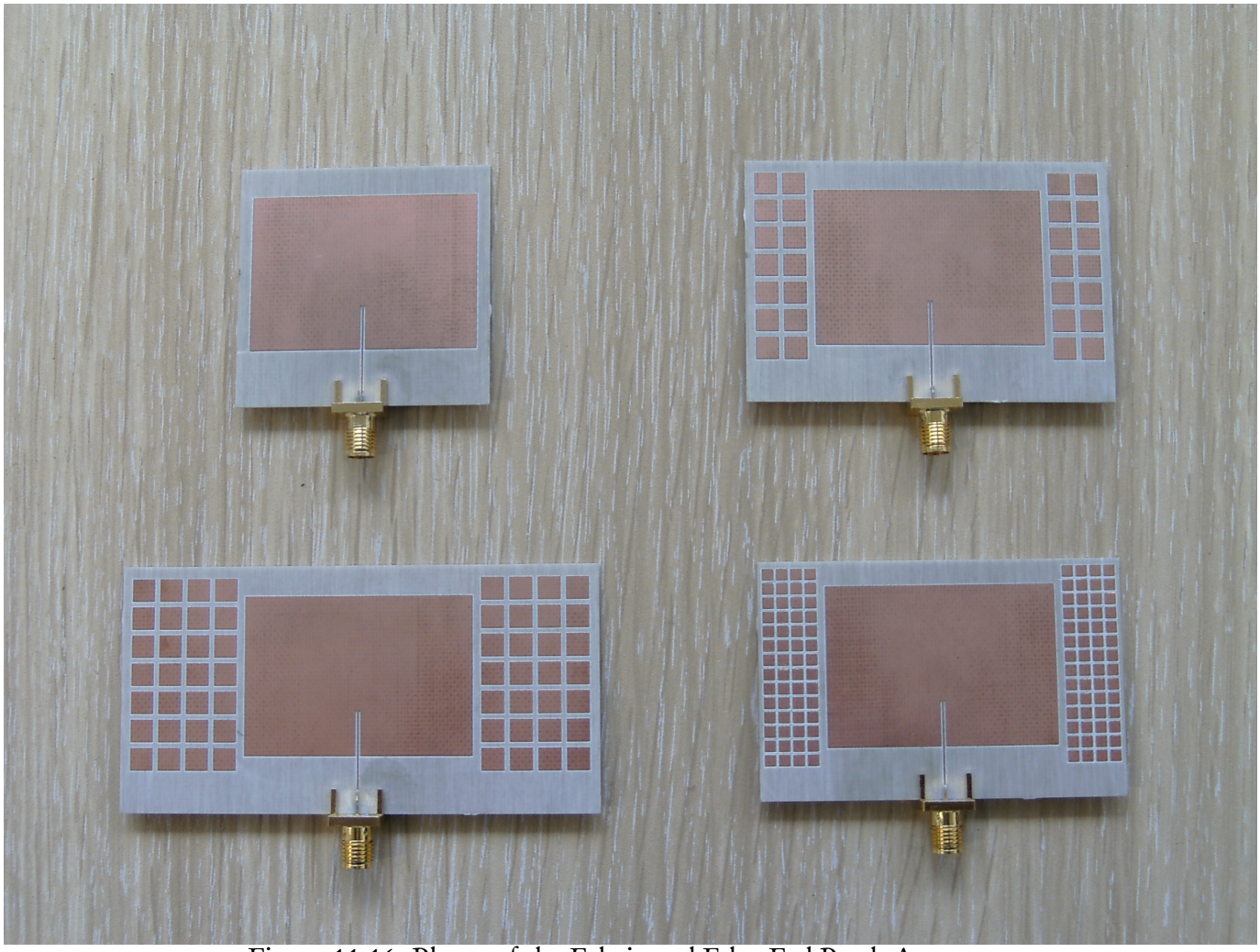


Figure 11.16: Photo of the Fabricated Edge Fed Patch Antennas

Chapter 12

Conclusions and Recommendations

"Now, the name of this talk is "There is Plenty of Room at the Bottom"—not just "There is Room at the Bottom." What I have demonstrated is that there is room—that you can decrease the size of things in a practical way. I now want to show that there is plenty of room. I will not now discuss how we are going to do it, but only what is possible in principle—in other words, what is possible according to the laws of physics. I am not inventing anti-gravity, which is possible someday only if the laws are not what we think. I am telling you what could be done if the laws are what we think; we are not doing it simply because we haven't yet gotten around to it." - Richard Feynman, whose revolutionary vision launched the global nanotechnology race.

12.1 Overview

This chapter presents the conclusions of our research and makes some recommendations for future research.

12.2 Conclusions

As can be seen from the results in chapter 7, we succeeded at our SDR OFDM implementation objective where transmitted 802.11g WLAN OFDM symbols could be correctly detected by off the shelf WLAN OFDM chips, and power of the received signal measured. It will therefore be possible to design controlled experiments to test the passive radar applications of OFDM. The SDR base station can also be used to dynamically perform OFDM optimisations.

From the results in chapter 11, we also succeeded in our objective to prove that designing a patch antenna with EBG structures would result in improved directivity. We also discovered directivity can be improved without losing any bandwidth. We however did not succeed in our objective to model the EBG patch antenna using Feko, due to limitations of the Feko *DP* geometry card.

We succeeded in one of our final objectives, which was to design a mixed signal circuit that would allow USRP2 RF daughter boards to be used with the new and more powerful Rhino board. We however did not succed in fabricating the Rhino expansion board due to time constraints.

Finally, the 4 year old video streaming puzzle was finally answered! This is because by doing an OFDM SDR implementation, we were able to understand exactly how mobile wireless protocols work (all modern wireless protocols from DVB and WLAN, to WiMAX and LTE, currently use the OFDM modulation method at their core). The OFDM implementation gave us immense control over our data transmission speeds for example (section 6.8, section 6.2). We also show how using new custom meta materials, a new and better designed antenna with improved directivity can be fabricated (section 11.3.2). Having an antenna with better directivity and hence receive sensitivity would allow us to either increase our coding rate for existing standard modulation modes or alternatively it would allow us to increase the modulation mode order (by for example using 256 QAM instead of 64 QAM, [100]), since less errors are likely to occur. Increasing our code rate would improve our data rate so that high density 3D TV video streams could be efficiently transmitted wirelessly in future. Data rates can also be improved by using more advanced FEC codes such as Turbo codes and Viterbi codes with longer constraint lengths.

12.3 Recommendations

Based on our conclusions above, the following recommendations can be made for future work:

- The Gnuradio 802.11 WLAN OFDM Transmitter can potentially be used to perform controlled Radar OFDM experiments.
- The feko file generated by the mpatchebgcalc.py script could be imported into cadfeko so as to perform optimisation simulations used to help ascertain what the best EBG and patch antenna dimensions would be. This would however entail a bug fix and enhancements to the Feko Antenna Modelling software. The generated feko file should also seek to model an inset feed rather than a pin fed patch antenna so as to potentially improve the accuracy of the simulation model.
- The mpatchebgcalc.py script will need to be refactored so as to allow better convergence onto the desired frequency. For the 2.4 GHz resonant frequency patch antenna we fabricated for example, the actual resonant frequency was 2.648 GHz.
- The mpatchebgcalc.py script will need to be refactored so as to allow better convergence onto the desired input impedance value. For our fabricated antennas for example, the final input impedance was 30Ω instead of the desired 50Ω.
- The mpatchebgcalc.py could be extended to include dimension calculations for multiband antennas

in which case one could look at how different rows and columns of EBGs could be used to limit multiple $TM(0, n)$ modes for the multiple resonant frequency bands.

- Custom materials could be designed at the nano level so as fabricate antennas with improved directivity and bandwidth.
- A faster OFDM implementation could be implemented by making use of the nano antenna which exhibits improved directivity for example.
- The Rhino expansion board spice simulations will need to be done. The board layout will also need to be done using possibly using the PCB application from the gEDA software suite. Once the board is fabricated, it will need to be tested with the Rhino FPGA board and the USRP2s RF daughter boards. The Rhino board tool platform is still under development as of this date.

Chapter 13

Appendix

13.1 SDR OFDM Code

This appendix presents the gnuradio python flowgraph code used to transmit the beacon frames.

```
#!/usr/bin/env python
#
# Copyright 2005, 2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
#
# Built on the incredible work of FTW, Credits: https://www.cgran.org/wiki/ftw80211ofdmtx
# List of Contributors: Andrea Costantini,
#                       Paul Fuxjaeger, (fuxjaeger@ftw.at)
#                       Danilo Valerio, (valerio@ftw.at)
#                       Paolo Castiglione, (castiglione@ftw.at)
#                       Giammarco Zacheo, (zacheo@ftw.at)
#                       Joseph Wamicha
#

import time, struct, sys
from gnuradio import gr, blks2, usrp2, eng_notation
from gnuradio.eng_option import eng_option
from optparse import OptionParser
from wlan_ofdm_tx import wlan_flow_tx
from binascii import unhexlify
import wlan_utils as wlan_packet_utils

"""
./wlan_tx.py --interface=eth1 --freq=2.412e9 --interp=5 --frame-type=beacon -r 0
"""

class wlan_tx(gr.top_block):
    def __init__(self, options, payload):
        gr.top_block.__init__(self)
        ################################################
        # Variables
        ################################################
        self.interface = options.interface                # Network Interface
```

```
        self.usrp2_mac_addr = options.usrp2_mac_addr        # USRP2 MAC address
        self.tx_freq = options.freq                         # Center Frequency
        self.interpolation = options.interp                 # Interpolation Factor
        self.gain = options.gain                            # Tx Gain
        self.amplitude = 0.1
        self.samp_rate = samp_rate = 32000
        self.frame = options.frame                          # 802.11g management frame
        self.info = ""                                      # Holds the 802.11 frame info
        self.ofdm_msg = ""                                  # OFDM message to be transmitted

        self.fft_length = 64
        self.cp_length = 16
        self.symbol_time = options.interp   (self.fft_length + self.cp_length)   (1/100)

        # create the ofdm symbol message.
        self.create_ofdm_symbols(payload, options)
        #self.connect(self.ofdm_msg, gr.file_sink(gr.sizeof_gr_complex, "802.11-ofdm-packet.dat"))

        # Number of OFDM data symbols which is needed to generate the preamble and zerogap
        N_sym = self.info["N_sym"]

        # Setup sink and connect output of wlan_flow_tx to it.
        self.setup_usrp_sink()
        self.flow_tx = wlan_flow_tx(options, N_sym)
        # Static value to make sure we do not exceed +-1 for the floats being sent to the sink.
        self.norm = options.norm
        self.amp = gr.multiply_const_cc(self.norm)

        # setup basic connections
        self.connect((self.flow_tx, 0), (self.amp, 0))

        # Connect 802.11 OFDM source to USRP2 sink...
        self.connect((self.amp, 0), (self.u, 0))

        # If logging is enabled, write final baseband signal to disk.
        if options.log:
            self.connect(self.flow_tx, gr.file_sink(gr.sizeof_gr_complex, "final.dat"))


    def setup_usrp_sink(self):
        """
        Creates a USRP sink, determines the settings for best bitrate,
        and attaches to the transmitter's subdevice.
        """
        self.u = usrp2.sink_32fc(self.interface, self.usrp2_mac_addr)
        print "USRP2 mac address:", self.u.mac_addr()
        self.u.set_interp(self.interpolation)
        print "Max gain: ", self.u.gain_max()
        self.u.set_gain(self.gain)
        self.set_freq(self.tx_freq)

    def set_freq(self, target_freq):
        """
        Set the center frequency we're interested in.

        @param target_freq: frequency in Hz
        @rtype: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.
        """
        tr = self.u.set_center_freq(target_freq)
        if tr == None:
            sys.stderr.write('Failed to set center frequency.\n')
            raise SystemExit, 1
        else:
            sys.stdout.write('Center frequency successfully set.\n')

    def create_ofdm_symbols(self, payload, options):
        """
        Create the ofdm symbols to be sent.

        @param payload: data to send
        @type payload: string
        """

        if options.frame_type == "probe":
            self.info = wlan_packet_utils.get_probe_info(payload, options.regime, self.symbol_time)
        else:
            self.info = wlan_packet_utils.get_beacon_info(payload, options.regime, self.symbol_time)
```

```
        N_cbps = self.info["N_cbps"]
        N_bpsc = self.info["N_bpsc"]
        N_rate = self.info["rate"]
        N_sym = self.info["N_sym"]

        (pkt, Length) = wlan_packet_utils.gr_wlan_make(payload, options.regime, self.symbol_time, options.frame_type)
        (pkt_scrambled, Length) = wlan_packet_utils.scrambler(pkt, Length)
        pkt_coded = wlan_packet_utils.conv_encoder(pkt_scrambled, Length, options.regime, N_cbps, N_bpsc, N_sym, N_rate)
        pkt_interleaved = wlan_packet_utils.interleaver(pkt_coded , options.regime, N_cbps, N_bpsc)
        self.ofdm_msg = gr.message_from_string(pkt_interleaved)

        return self.ofdm_msg

# ////////////////////////////////////////////////////////////////////////////
#                                   main
# ////////////////////////////////////////////////////////////////////////////

def main():

    parser = OptionParser(option_class=eng_option, conflict_handler="resolve")

    parser.add_option("-e", "--interface", type="string", default="eth0", help="set ethernet interface, [default=%default]")

    parser.add_option("-m", "--usrp2-mac-addr", type="string", default="", help="set USRP2 MAC address, [default=auto-select]")

    parser.add_option("-b", "--bssid-mac-addr", type="string", default="",
help="set BSSID MAC address for the 802.11 frame, [default=auto-select]")

    parser.add_option("-s", "--essid", type="string", default="", help="set ESSID for the 802.11 frame, [default=auto-select]")

    parser.add_option("-d", "--destination-mac-addr", type="string", default="",
help="set Destination MAC address for the 802.11 frame, [default=auto-select]")

    parser.add_option("-f", "--freq", type="eng_float", default=2.412e9, help="set USRP2 carrier frequency, [default=%default]")

    parser.add_option("-i", "--interp", type="intx", default=5,
                      help="set USRP2 interpolation factor, [default=%default]\
                      5 -> 802.11a/g, OFDM-symbolduration=4us, \
                      10 -> 802.11p, OFDM-symbolduration=8us")

    parser.add_option("-g", "--gain", type="int", default=10 , help="set USRP2 Tx GAIN in [dB] [default=%default]")

    parser.add_option("", "--regime", type="string", default="1",
                      help="set OFDM coderegime:    [default=%default]\
                      1 -> 6 (3) Mbit/s (BPSK r=0.5), \
                      2 -> 9 (4.5) Mbit/s (BPSK r=0.75), \
                      3 -> 12 (6) Mbit/s (QPSK r=0.5), \
                      4 -> 18 (9) Mbit/s (QPSK r=0.75), \
                      5 -> 24 (12) Mbit/s (QAM16 r=0.5), \
                      6 -> 36 (18) Mbit/s (QAM16 r=0.75), \
                      7 -> 48 (24) Mbit/s (QAM64 r=0.66), \
                      8 -> 54 (27) Mbit/s (QAM64 r=0.75)")

    parser.add_option("-n", "--norm", type="eng_float", default=0.3 , help="set gain factor for complex baseband floats [default=%default]")

    parser.add_option("-s", "--swapIQ", action="store_true", default=False,
help="swap IQ components before sending to USRP2 sink [default=%default]")

    parser.add_option("-r", "--repetition", type="int", default=1 , help="set number of frame-copies to send, 0=infinite [default=%default] ")

    parser.add_option("-l", "--log", action="store_true", default=False, help="write debug-output of individual blocks to disk")

    parser.add_option("-a", "--frame", type="string", default="",
help="a default 802.11g frame will be sent instead if this option is left empty, [default=%default]")

    parser.add_option("-t", "--frame-type", type="string", default="",
help="a default 802.11g beacon frame will be transmitted if no frame type is specified. [default=%default]")

    (options, args) = parser.parse_args ()

    # Default 802.11 LAN management beacon frame...
    # The Timing Synchronisation Function (TSF/Timestamp) still needs to be implemented and Point coordination function (PCF)...
    lan_management_frame = chr(0x78) + chr(0xa1) + chr(0xa5) + chr(0x73) + chr(0xa7) + chr(0x05) + chr(0x00) + chr(0x00) # timestamp field
    lan_management_frame += chr(0x66) + chr(0x00) # beacon interval field: 0.104448 seconds, 2-bytes
    lan_management_frame += chr(0x31) + chr(0x04) # capability information field, 2-bytes

    #SSID parameter set
    lan_management_frame += chr(0x00) # Tag number 0
    #lan_management_frame += chr(0x05) # Tag length
    #lan_management_frame += chr(0x53) + chr(0x61) + chr(0x6e) + chr(0x74) + chr(0x69) # Santi
    lan_management_frame += chr(0x05) # Tag length
    lan_management_frame += chr(0x52) + chr(0x68) + chr(0x69) + chr(0x6e) + chr(0x6f) # Rhino
```

```
    #Supported rates
    lan_management_frame += chr(0x01) # Tag number
    lan_management_frame += chr(0x08) # Tag length
    #Tag interpretation, supported rates: 1, 2, 5.5, 6, 9, 11, 12, 18 Mbit/sec
    lan_management_frame += chr(0x82)+ chr(0x84)+ chr(0x8b)+ chr(0x0c)+ chr(0x12)+ chr(0x96)+ chr(0x18)+ chr(0x24)

    # DS Parameter set
    lan_management_frame += chr(0x03) #Tag no.
    lan_management_frame += chr(0x01) #Tag length
    lan_management_frame += chr(0x01) #Tag interpretation, current channel is 1

    # Traffic Indication Map (TIM)
    lan_management_frame += chr(0x05) # Tag number
    lan_management_frame += chr(0x04) # TIM length
    lan_management_frame += chr(0x00) + chr(0x01) + chr(0x00) + chr(0x00) # TIM

    # Country information
    lan_management_frame += chr(0x07) # Tag number
    lan_management_frame += chr(0x06) # Tag length
    lan_management_frame += chr(0x5a) + chr(0x41) + chr(0x20)  # Country code, ZA, Any environment
    lan_management_frame += chr(0x01) + chr(0x0d) + chr(0x14) # Start channel: 1, Channels: 13, Max TX Power: 20 dBm

    # ERP Information
    lan_management_frame += chr(0x2a) # Tag number
    lan_management_frame += chr(0x01) # Length
    lan_management_frame += chr(0x00) # no non-ERP STAs, no protection, short/long pre-ambles

    # Extended Supported Rates Information
    lan_management_frame += chr(0x32) # Tag number
    lan_management_frame += chr(0x04) # Length
    lan_management_frame += chr(0x30) + chr(0x48) + chr(0x60) + chr(0x6c) # TIM

    management_data = None
    frame = options.frame
    if frame == "":
        management_data = lan_management_frame
    else:
        management_data = unhexlify(frame)

    print "Building the WLAN OFDM Flow Graph..."
    tb = wlan_tx(options, management_data)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: failed to enable realtime gr scheduling."

    # Start the flow graph.
    print "Start top block..."
    tb.start()
    print "Send 802.11 symbols..."
    # Send 802.11 frame symbols
    tb.flow_tx.send_ofdm_symbols(tb.ofdm_msg , eof=False)
    tb.flow_tx.send_ofdm_symbols('', eof=True)

    # wait for it to finish
    tb.wait()
    print "Signal Flow Graph Run complete."

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Listing 13.1: The wlan_tx.py Code File

```
#!/usr/bin/env python
#
# Copyright 2005, 2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
```

```
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.

import copy, math, sys
from gnuradio import gr, gru, usrp2, eng_notation
import gnuradio.gr.gr_threading as _threading
from gnuradio.blks2impl import psk, qam
import wlan_utils as wlan_packet_utils
import gr_wlan

# wlan_flow_tx class sets up the transmit path.
class wlan_flow_tx(gr.hier_block2):
    def __init__(self, options, N_sym):
        gr.hier_block2.__init__(self, "wlan_ofdm_tx",
                                gr.io_signature(0, 0, 0), # Input signature
                                gr.io_signature(1, 1, gr.sizeof_gr_complex)) # Output signature

        # OFDM MODULATION TO ENCODE 802.11 PACKETS BEGINS: #
        msgq_limit=2
        pad_for_usrp=False

        self.fft_length = 64
        self.total_sub_carriers = 53
        self.data_subcarriers = 48
        self.cp_length = 16
        self.regime = options.regime
        self.symbol_length = self.fft_length + self.cp_length

        # assuming we have 100Ms/s going to the USRP2 and 80 (FFT + GI / 64 subcarriers + 16 subcarriers) samples per symbol
        # we can calculate the OFDM symbol time (in microseconds)
        # depending on the interpolation factor
        self.symbol_time = options.interp    (self.fft_length + self.cp_length)    (1/100)

        win = []

        if(self.regime == "1" or self.regime == "2"):
            rotated_const = wlan_packet_utils.bpsk(self)

        elif (self.regime == "3" or self.regime == "4"):
            rotated_const = wlan_packet_utils.qpsk(self)

        elif(self.regime == "5" or self.regime == "6"):
            rotated_const = wlan_packet_utils.qam16(self)

        elif(self.regime == "7" or self.regime == "8"):
            rotated_const = wlan_packet_utils.qam64(self)

        # map groups of bits to complex symbols
        self._pkt_input = gr_wlan.ofdm_mapper(rotated_const, msgq_limit, self.data_subcarriers, self.fft_length)

        # insert pilot symbols
        self.pilot = gr_wlan.ofdm_pilot_cc(self.data_subcarriers)

        # move subcarriers to their designated place and insert DC
        self.cmap = gr_wlan.ofdm_cmap_cc(self.fft_length, self.total_sub_carriers)

        # inverse fast fourier transform
        self.ifft = gr.fft_vcc(self.fft_length, False, win, False)

        # add cyclic prefix
        self.cp_adder = gr.ofdm_cyclic_prefixer(self.fft_length, self.symbol_length)

        # scale accordingly
        self.scale = gr.multiply_const_cc(1.0 / math.sqrt(self.fft_length))

        # add training sequence
        self.preamble = wlan_packet_utils.insert_preamble(self.symbol_length, N_sym)

        # append zero samples at the end (receiver needs that to decode)
        self.zerogap = wlan_packet_utils.insert_zerogap(self.symbol_length, N_sym)

        # repeat the frame a number of times
        self.repeat = gr_wlan.repetition(80, options.repetition, N_sym)

        self.s2v = gr.stream_to_vector(gr.sizeof_gr_complex , self.symbol_length)
        self.v2s = gr.vector_to_stream(gr.sizeof_gr_complex , self.symbol_length)

        # swap real and immaginary component before sending (GNURadio/USRP2 bug!)
        if options.swapIQ == True:
```

```
            self.gr_complex_to_imag_0 = gr.complex_to_imag(1)
            self.gr_complex_to_real_0 = gr.complex_to_real(1)
            self.gr_float_to_complex_0 = gr.float_to_complex(1)
            self.connect((self.v2s, 0), (self.gr_complex_to_imag_0, 0))
            self.connect((self.v2s, 0), (self.gr_complex_to_real_0, 0))
            self.connect((self.gr_complex_to_real_0, 0), (self.gr_float_to_complex_0, 1))
            self.connect((self.gr_complex_to_imag_0, 0), (self.gr_float_to_complex_0, 0))
            self.connect((self.gr_float_to_complex_0, 0), (self))
        elif options.swapIQ == False:
            self.gr_complex_to_imag_0 = gr.complex_to_imag(1)
            self.gr_complex_to_real_0 = gr.complex_to_real(1)
            self.gr_float_to_complex_0 = gr.float_to_complex(1)
            self.connect((self.v2s, 0), (self.gr_complex_to_imag_0, 0))
            self.connect((self.v2s, 0), (self.gr_complex_to_real_0, 0))
            self.connect((self.gr_complex_to_imag_0, 0), (self.gr_float_to_complex_0, 1))
            self.connect((self.gr_complex_to_real_0, 0), (self.gr_float_to_complex_0, 0))
            self.connect((self.gr_float_to_complex_0, 0), (self))

        # connect the blocks
        self.connect((self._pkt_input, 0), (self.pilot, 0))
        self.connect((self._pkt_input, 1), (self.preamble, 1))
        self.connect((self.preamble, 1), (self.zerogap, 1))

        #if options.repetition == 1:
        #       self.connect(self.pilot, self.cmap, self.ifft, self.cp_adder, self.scale, self.s2v, \
        #       self.preamble, self.zerogap, self.v2s)

        #elif options.repetition > 1:
        self.connect(self.pilot, self.cmap, self.ifft, self.cp_adder, self.scale, self.s2v, self.preamble, self.zerogap, self.repeat, self.v2s)
        #else:
        #       print"Error: repetiton must be a integer number >= 1 \n"
        #       sys.exit(1)

        if options.log:
            self.connect((self._pkt_input), gr.file_sink(gr.sizeof_gr_complex    self.data_subcarriers, "ofdm_mapper.dat"))
            self.connect(self.pilot, gr.file_sink(gr.sizeof_gr_complex    (5 + self.data_subcarriers), "ofdm_pilot.dat"))
            self.connect(self.cmap, gr.file_sink(gr.sizeof_gr_complex    self.fft_length, "ofdm_cmap.dat"))
            self.connect(self.ifft, gr.file_sink(gr.sizeof_gr_complex    self.fft_length, "ofdm_ifft.dat"))
            self.connect(self.cp_adder, gr.file_sink(gr.sizeof_gr_complex, "ofdm_cp_adder.dat"))
            self.connect(self.scale, gr.file_sink(gr.sizeof_gr_complex, "ofdm_scale.dat"))
            self.connect(self.preamble, gr.file_sink(gr.sizeof_gr_complex    self.symbol_length, "ofdm_preamble.dat"))
            self.connect(self.zerogap, gr.file_sink(gr.sizeof_gr_complex    self.symbol_length, "ofdm_zerogap.dat"))

        # OFDM MODULATION END #

    def send_ofdm_symbols(self, ofdm_msg, eof=False):
        """
        Send the ofdm_msg.
        """
        if eof:
            # tell self._pkt_input we're not sending any more packets
            ofdm_msg = gr.message(1)

        return self._pkt_input.msgq().insert_tail(ofdm_msg)
```

Listing 13.2: The wlan_ofdm_tx.py Code File

```
#!/usr/bin/env python
#
# Copyright 2005, 2006 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.

import struct, numpy, math, sys
from gnuradio import gru,gr
```

```
import gr_wlan


def get_beacon_info(payload, regime, symboltime):
    # MAC header bytes for ERP-OFDM
    mac_framectrl = chr(0x80) + chr(0x00)                                                   # this is a beacon frame
    mac_duration  = chr(0x00) + chr(0x00)                                                   # dummy, value will be set by script
    mac_address1  = chr(0xff) + chr(0xff) + chr(0xff) + chr(0xff) + chr(0xff) + chr(0xff)   # some destination mac-address
    mac_address2  = chr(0x00) + chr(0x60) + chr(0xb3) + chr(0x11) + chr(0x12) + chr(0x13)   # some source mac-address - usrp2mac: 00:50:c2:85:32:73
    mac_address3  = chr(0x00) + chr(0x60) + chr(0xb3) + chr(0x11) + chr(0x12) + chr(0x13)   # some BSSID mac-address - made the same as the usrp2mac
    mac_seqctrl   = chr(0x00) + chr(0x00)                                                   # some sequence and fragment number

    return get_frame(payload, regime, symboltime, mac_framectrl, mac_duration, mac_address1, mac_address2, mac_address3, mac_seqctrl)

def get_probe_info(payload, regime, symboltime):
    # MAC header bytes for ERP-OFDM
    mac_framectrl = chr(0x40) + chr(0x00)                                                   # this is a beacon frame
    mac_duration  = chr(0x00) + chr(0x00)                                                   # dummy, value will be set by script
    mac_address1  = chr(0xff) + chr(0xff) + chr(0xff) + chr(0xff) + chr(0xff) + chr(0xff)   # some destination mac-address
    mac_address2  = chr(0x00) + chr(0x60) + chr(0xb3) + chr(0x11) + chr(0x12) + chr(0x13)   # some source mac-address - usrp2mac: 00:50:c2:85:32:73
    mac_address3  = chr(0x00) + chr(0x60) + chr(0xb3) + chr(0x11) + chr(0x12) + chr(0x13)   # some BSSID mac-address - made the same as the usrp2mac
    mac_seqctrl   = chr(0x00) + chr(0x00)                                                   # some sequence and fragment number

    return get_frame(payload, regime, symboltime, mac_framectrl, mac_duration, mac_address1, mac_address2, mac_address3, mac_seqctrl)

def get_frame(payload, regime, symboltime, mac_framectrl, mac_duration, mac_address1, mac_address2, mac_address3, mac_seqctrl):
    # pre-assemble the MPDU (duration is filled with dummy and CRC32 is missing at this point)
    packet = mac_framectrl + mac_duration + mac_address1 + mac_address2 + mac_address3 + mac_seqctrl + payload

    # Defaults
    N_cbps = 48
    rate = 1/float(2)
    modulation = "bpsk"
    N_bpsc = 1
    # switch coding rate and constellation according to regime profile
    if (regime == "1"):
        modulation = "bpsk"
        rate = 1/float(2)
        N_cbps = 48
        N_bpsc = 1
    elif(regime == "2"):
        modulation = "bpsk"
        rate = 3/float(4)
        N_cbps = 48
        N_bpsc = 1
    elif(regime == "3"):
        modulation = "qpsk"
        rate = 1/float(2)
        N_cbps = 96
        N_bpsc = 2
    elif(regime == "4"):
        modulation = "qpsk"
        rate = 3/float(4)
        N_cbps = 96
        N_bpsc = 2
    elif(regime == "5"):
        modulation = "qam16"
        rate = 1/float(2)
        N_cbps = 192
        N_bpsc = 4
    elif(regime == "6"):
        modulation = "qam16"
        rate = 3/float(4)
        N_cbps = 192
        N_bpsc = 4
    elif(regime == "7"):
        modulation = "qam64"
        rate = 2/float(3)
        N_cbps = 288
        N_bpsc = 6
    elif(regime == "8"):
        modulation = "qam64"
        rate = 3/float(4)
        N_cbps = 288
        N_bpsc = 6

    # uncoded bits per OFDM symbol
    N_dbps = int(N_cbps    rate)

    # number of DATA symbols the frame needs to have
    # 16 service bits, 32 crc bits and 6 tail bits need to be included
    N_sym = int(math.ceil((16 + 32 + 6 + 8    len(packet))/float(N_dbps)))
```

```
    # airtime of frame in microseconds
    # the additional 2 at the end of this formula is not in the
    # standard encoding rules but in the annex G reference frame it is there!
    txtime = int(5 * symboltime + symboltime * N_sym) + 2
    mac_duration = chr((txtime >> 8) & 0xff) + chr(txtime & 0xff)

    # number of uncoded bits that could be sent using N_sym DATA OFDM symbols
    N_data = int(N_sym * N_dbps)

    # number of (uncoded) padding bits that need to be added at the end
    N_pad = N_data - (16 + 32 + 6 + 8 * len(packet))

    # assemble the MPDU (now duration is correct)
    packet = mac_framectrl + mac_duration + mac_address1 + mac_address2 + mac_address3 + mac_seqctrl + payload

    #print "MAC Frame Control:", ByteToHex(mac_framectrl)
    #print "MAC duration:", ByteToHex(mac_duration)
    #print "MAC address1:", ByteToHex(mac_address1)
    #print "MAC address2:", ByteToHex(mac_address2)
    #print "MAC address3:", ByteToHex(mac_address3)
    #print "MAC seqctrl:", ByteToHex(mac_seqctrl)
    #print "Beacon frame payload:", ByteToHex(payload)
    #print "\nFinal packet: Packet Hex Bytes:", ByteToHex(packet)

    dic = {"packet":packet, "packet_len":(len(packet)+ 4), "txtime":txtime, "mac_duration":mac_duration, "modulation":modulation,\
           "rate":rate, "N_sym":N_sym, "N_data":N_data,"N_pad":N_pad, "N_cbps":N_cbps, "N_bpsc":N_bpsc, "N_dbps":N_dbps}

    return dic

def ByteToHex( byteStr ):
    return ''.join( [ "%02X " % ord( x ) for x in byteStr ] ).strip()

def gr_wlan_make(payload, regime, symboltime, frame_type):

    if frame_type == "probe":
        info = get_probe_info(payload, regime, symboltime)
    else:
        info = get_beacon_info(payload, regime, symboltime)

    packet      = info["packet"]
    packet_len  = info["packet_len"]
    N_sym       = info["N_sym"]
    N_data      = info["N_data"]
    N_pad       = info["N_pad"]
    txtime      = info["txtime"]

    # check length of MPDU
    MAXLEN = 3168
    if packet_len > MAXLEN:
        raise ValueError, "MPDU-length must be in [0, %d]" % (MAXLEN,)

    print"Txtime in microseconds:",txtime
    print"Length of MPDU in bytes =", packet_len
    print"Number of DATA OFDM symbols =", N_sym
    print"Number of padding bits =", N_pad

    # generate rate bits in SIGNAL OFDM symbol
    rate_bits = 0x0d
    if (regime == "1"):
        rate_bits = 0x0d
    elif (regime == "2"):
        rate_bits = 0x0f
    elif (regime == "3"):
        rate_bits = 0x05
    elif (regime == "4"):
        rate_bits = 0x07
    elif (regime == "5"):
        rate_bits = 0x09
    elif (regime == "6"):
        rate_bits = 0x0b
    elif (regime == "7"):
        rate_bits = 0x01
    elif (regime == "8"):
        rate_bits = 0x03

    # generate length bits in SIGNAL OFDM symbol
    app = 0
    for i in range (0,12):
        app = app | (((packet_len >> i) & 1) << (11 - i))
    Length = app

    # generate parity check bit in SIGNAL OFDM symbol
    parA = parB = 0
```

```
    for k in range (0,12):
        parA += ((Length >> k) & (0x01))
        parB += ((rate_bits   >> k) & (0x01))
    parity_bit =  (parA + parB) % 2

    # generate tail and padding bits
    if ((N_pad + 6) % 8 == 0):
        app = ''
        for i in range (0, (N_pad + 6)/8):
            app += chr(0x00)
        TAIL_and_PAD = app
    else:
        app = ''
        for i in range (0, (N_pad + 6)/8):
            app += chr(0x00)
        # add one more byte if N_pad + 6 is not a multiple of 8
        # we remove those bits again after the convolutional encoder
        TAIL_and_PAD = app + chr(0x00)

    # generate all bits for SIGNAL OFDM symbol
    Signal_tail = 0x0
    Signal_field = 0x000
    Signal_field = (rate_bits << 20) | (Length << 7) | Signal_tail | (parity_bit << 6)
    chr1 = chr((Signal_field >> 16) & 0xff)
    chr2 = chr((Signal_field >> 8) & 0xff)
    chr3 = chr(Signal_field  & 0xff)
    SIGNAL_FIELD = chr1 + chr2 + chr3

    # generate 16 all-zero SERVICE bits
    SERVICE = chr(0x00) + chr(0x00)

    PLCP_HEADER = SIGNAL_FIELD + SERVICE

    MPDU = make_MPDU (packet)

    MPDU_with_crc32 = gen_and_append_crc32(MPDU , packet)

    Length = len(MPDU_with_crc32)

    return PLCP_HEADER + MPDU_with_crc32 + TAIL_and_PAD , Length


def make_MPDU(payload):
    app = conv_packed_binary_string_to_1_0_string(payload)
    app = list(app)
    mpdu = ['0']   len(app)
    j = 0
    while (j < len(app)):
        for i in range (0, 8):
            mpdu[i+j] = app[7-i+j]      # Change into transmit order
        j += 8
    mpdu = "".join(mpdu)
    return conv_1_0_string_to_packed_binary_string(mpdu)


def interleaver (payload, regime, N_cbps, N_bpsc):

    length_payload = len(payload)
    length_payload_bit = 8   length_payload
    half_interleaved = first_permutation(payload, N_cbps, length_payload_bit)
    interleaved = second_permutation(half_interleaved, N_cbps, N_bpsc, length_payload_bit)

    return interleaved

def first_permutation(payload, N_cbps, length_payload_bit):

    app = conv_packed_binary_string_to_1_0_string(payload)
    app=list(app)
    new = ['0']   len(app)
    if(N_cbps == 48): # Modulation "bpsk"
        j=0
        while(j < length_payload_bit/N_cbps):

            for k in range (0, N_cbps):
                i = (N_cbps/16)    (k%16) + int(math.floor(k/16))
                new[i + (j  N_cbps)] = app[k + (j  N_cbps)]
            j +=1

    else:   # other modulation ---> first 48 bits alone (signal field)

        for k in range (0, 48):
            i = (48/16)    (k%16) + int(math.floor(k/float(16)))
```

```
            new[i] = app[k]
        j = 0
        while(j < (length_payload_bit - 48) / N_cbps):
            for k in range (0, N_cbps):
                i = (N_cbps/16)    (k%16) + int(math.floor(k/float(16)))
                new[i + 48 + (j N_cbps)] = app[k + 48 + (j N_cbps)]
            j +=1

    new = "".join(new)
    return conv_1_0_string_to_packed_binary_string(new)


def second_permutation(half_interleaved, N_cbps, N_bpsc, length_payload_bit):

    app = conv_packed_binary_string_to_1_0_string(half_interleaved)
    app=list(app)
    new_temp = ['0']    (len(app) - 48)
    new = app[0:48] + new_temp
    s = max(N_bpsc/2 , 1)
    k=0

    while (k < (length_payload_bit - 48)/N_cbps):
        for i in range (0 , N_cbps):
            j = (s    int(math.floor(i/s))) + (i + N_cbps - (int(math.floor(16 i/float(N_cbps))))) % s
            new[j + 48 + (k N_cbps)] = app[i + 48 + (k N_cbps)]
        k +=1

    new = "".join(new)
    return conv_1_0_string_to_packed_binary_string(new)


def conv_encoder(pkt, Length, regime, N_cbps, N_bpsc, N_sym, N_rate):

    Real_num_of_bits = N_sym    N_cbps    # We could have more then correct number we need
                                          # See  Tail and pad in gr_wlan_make function
    app = conv_packed_binary_string_to_1_0_string(pkt)
    app = list(app)
    encoded = ['0']    (2    len(app))
    g0 = 0x5b   # Generator polynomial (133 base 8)
    g1 = 0x79   # Generator polynomial (171 base 8)
    outA = 0
    outB = 0
    register = 0
    for i in range (0, len(app)):
        if(app[i] == '1'):
            register = register >> 1     # Shift the status in the conv encoder
            register = register | 0x40   # push 1 in the first element in the register after the shift
        else:
            register = register >> 1

        modA = modB = 0
        for k in range (0,8):
            modA += (((register & g0) >> k) & (0x01))     # Parity check (count the number of 1s)
            modB += (((register & g1) >> k) & (0x01))     # Parity check

        outA     =  modA % 2  # Modulo 2 sum
        outB     =  modB % 2
        encoded[2    i]    = str(outA)
        encoded[2    i + 1] = str(outB)

    # Puncturing operations-------------------------------------------
    if (regime == "1" or regime== "3" or regime == "5"):
        encoded = encoded[0:48 + Real_num_of_bits]
        encoded = "".join(encoded)

        return conv_1_0_string_to_packed_binary_string(encoded)

    elif (regime == "2" or regime == "4" or regime == "6" or regime == "8"):
        signal_coded = encoded[0:48]
        data_coded = encoded[48:]
        dynamic_offset = 0
        new = data_coded[0:3]
        while dynamic_offset + 9 < len(data_coded)-1 :
            new = new + data_coded[5 + dynamic_offset : 9 + dynamic_offset]
            dynamic_offset += 6
        new.append(data_coded[5 + dynamic_offset])
        new = new[0:Real_num_of_bits]
        new = signal_coded + new
        encoded = "".join(new)
        return conv_1_0_string_to_packed_binary_string(encoded)

    elif (regime == "7"):
        dinamic_offset = 0
```

```
        signal_coded = encoded[0:48]
        data_coded = encoded[48:]
        new = data_coded[0:3]
        while dinamic_offset + 4 < len(data_coded) -1 :
            new = new + data_coded[4 + dinamic_offset : 7 + dinamic_offset]
            dinamic_offset += 4
        new = new[0:Real_num_of_bits]
        new = signal_coded + new
        encoded = "".join(new)
        return conv_1_0_string_to_packed_binary_string(encoded)


def scrambler(pkt, Length_data):

    scrambling_seq = [0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1,\
                      0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1,\
                      0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1,\
                      0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1];

    app = conv_packed_binary_string_to_1_0_string(pkt)
    app = list(app)
    zero_forcing_index = Length_data   8
    scrambled = app[0:24]

    # Start from 24 because SIGNAL symbol mustn't be scrambled
    for k in range (24 , len(app)):
        scrambled.append(str(int(app[k])^(scrambling_seq[(k-24) % 127])))

    # Force six bit to "0" in return to 0 state at last
    for i in range (23 + zero_forcing_index +17 , 23 + zero_forcing_index + 22 + 1):
        scrambled[i] = '0'

    scrambled = "".join(scrambled)

    return conv_1_0_string_to_packed_binary_string(scrambled) , Length_data


def gen_and_append_crc32(MPDU, packet_for_crc):
    crc = gr_wlan.gr_wlan_crc32(packet_for_crc)
    return MPDU + struct.pack(">I", hexint(crc) & 0xFFFFFFFF)

def insert_preamble(length, N_sym):
    gr_wlan_preamble = [list(fft_preamble)]
    preamble = gr_wlan.ofdm_preamble(length, N_sym, gr_wlan_preamble)
    return preamble

def insert_zerogap(length, N_sym):
    gap = [list(gap_sample)]
    gr_wlan_zerogap = gr_wlan.zerogap(length, N_sym, gap)
    return gr_wlan_zerogap

def qam16(self):
    return gray_const_qam16

def qam64(self):
    return gray_const_qam64

def qpsk(self):
    return gray_const_qpsk

def bpsk(self):
    return gray_const_bpsk

def conv_packed_binary_string_to_1_0_string(s):
    """
    '\xAF' --> '10101111'
    """
    r = []
    for ch in s:
        x = ord(ch)
        for i in range(7,-1,-1):
            t = (x >> i) & 0x1
            r.append(t)

    return ''.join(map(lambda x: chr(x + ord('0')), r))

def conv_1_0_string_to_packed_binary_string(s):
    """
    '10101111' -> ('\xAF', False)

    Basically the inverse of conv_packed_binary_string_to_1_0_string,
    but also returns a flag indicating if we had to pad with leading zeros
    to get to a multiple of 8.
```

```
    """
    if not is_1_0_string(s):
        raise ValueError, "Input must be a string containing only 0's and 1's"

    # pad to multiple of 8
    padded = False
    rem = len(s) % 8
    if rem != 0:
        npad = 8 - rem
        s = '0'  npad + s
        padded = True

    assert len(s) % 8 == 0

    r = []
    i = 0
    while i < len(s):
        t = 0
        for j in range(8):
            t = (t << 1) | (ord(s[i + j]) - ord('0'))
        r.append(chr(t))
        i += 8
    return ''.join(r)


def is_1_0_string(s):
    if not isinstance(s, str):
        return False
    for ch in s:
        if not ch in ('0', '1'):
            return False
    return True

def string_to_hex_list(s):
    return map(lambda x: hex(ord(x)), s)


def hexint(mask):
    """
    Convert unsigned masks into signed ints.

    This allows us to use hex constants like 0xf0f0f0f2 when talking to
    our hardware and not get screwed by them getting treated as python
    longs.
    """
    if mask >= 2 31:
        return int(mask-2 32)
    return mask


def ascii_to_bin(char):
    ascii = ord(char)
    bin = []

    while (ascii > 0):
        if (ascii & 1) == 1:
            bin.append("1")
        else:
            bin.append("0")
        ascii = ascii >> 1
    bin.reverse()
    binary = "".join(bin)
    zerofix = (8 - len(binary))   '0'

    return zerofix + binary


#////////////////////////////////////////////////////////////////////////////
#                            Static tables
#////////////////////////////////////////////////////////////////////////////

gray_const_bpsk = (-1+0j, +1+0j)

gray_const_qpsk = (-0.7071 - 0.7071j, -0.7071 + 0.7071j, +0.7071 - 0.7071j, +0.7071 + 0.7071j)

gray_const_qam16 = (-0.9487 - 0.9487j,-0.9487 - 0.3162j,-0.9487 + 0.9487j,-0.9487 + 0.3162j,-0.3162 - 0.9487j,
-0.3162 - 0.3162j,-0.3162 + 0.9487j,-0.3162 + 0.3162j,0.9487 - 0.9487j, 0.9487 - 0.3162j, 0.9487 + 0.9487j,
0.9487 + 0.3162j, 0.3162 - 0.9487j, 0.3162 - 0.3162j, 0.3162 + 0.9487j, 0.3162 + 0.3162j)

gray_const_qam64 = (-1.0801 - 1.0801j,  -1.0801 - 0.7715j,  -1.0801 - 0.1543j,  -1.0801 - 0.4629j, -1.0801 + 1.0801j,
  -1.0801 + 0.7715j,  -1.0801 + 0.1543j,  -1.0801 + 0.4629j, -0.7715 - 1.0801j,  -0.7715 - 0.7715j,  -0.7715 - 0.1543j,
  -0.7715 - 0.4629j, -0.7715 + 1.0801j,  -0.7715 + 0.7715j,    -0.7715 + 0.1543j,  -0.7715 + 0.4629j, -0.1543 - 1.0801j,
  -0.1543 - 0.7715j,  -0.1543 - 0.1543j,  -0.1543 - 0.4629j, -0.1543 + 1.0801j,  -0.1543 + 0.7715j,  -0.1543 + 0.1543j,
```

```
    -0.1543 + 0.4629j, -0.4629 - 1.0801j,  -0.4629 - 0.7715j,  -0.4629 - 0.1543j,  -0.4629 - 0.4629j, -0.4629 + 1.0801j,
     -0.4629 + 0.7715j,  -0.4629 + 0.1543j,  -0.4629 + 0.4629j, 1.0801 - 1.0801j,   1.0801 - 0.7715j,   1.0801 - 0.1543j,
     1.0801 - 0.4629j, 1.0801 + 1.0801j,   1.0801 + 0.7715j,   1.0801 + 0.1543j,   1.0801 + 0.4629j, 0.7715 - 1.0801j,
     0.7715 - 0.7715j,   0.7715 - 0.1543j,   0.7715 - 0.4629j,
  0.7715 + 1.0801j,   0.7715 + 0.7715j,   0.7715 + 0.1543j,   0.7715 + 0.4629j, 0.1543 - 1.0801j,   0.1543 - 0.7715j,
     0.1543 - 0.1543j, 0.1543 - 0.4629j,
  0.1543 + 1.0801j,   0.1543 + 0.7715j,   0.1543 + 0.1543j,   0.1543 + 0.4629j, 0.4629 - 1.0801j,   0.4629 - 0.7715j,
     0.4629 - 0.1543j,   0.4629 - 0.4629j,
  0.4629 + 1.0801j,   0.4629 + 0.7715j,   0.4629 + 0.1543j,   0.4629 + 0.4629j)

fft_preamble = (0.3680 + 0.3680j, -1.0596 + 0.0187j, -0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j,
 -0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j, 0.0187 - 1.0596j, -0.6282 - 0.1078j,-0.1012 + 1.1421j,
0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j, 0.0187 - 1.0596j, 0.3680 + 0.3680j, -1.0596 + 0.0187j,
-0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j, -0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j,
0.0187 - 1.0596j, -0.6282 - 0.1078j, -0.1012 + 1.1421j, 0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j,
0.0187 - 1.0596j,0.3680 + 0.3680j, -1.0596 + 0.0187j, -0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j,
-0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j, 0.0187 - 1.0596j, -0.6282 - 0.1078j,-0.1012 + 1.1421j,
0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j, 0.0187 - 1.0596j, 0.3680 + 0.3680j, -1.0596 + 0.0187j,
-0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j, -0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j,
0.0187 - 1.0596j, -0.6282 - 0.1078j, -0.1012 + 1.1421j, 0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j,
0.0187 - 1.0596j, 0.3680 + 0.3680j, -1.0596 + 0.0187j, -0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j,
-0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j, 0.0187 - 1.0596j, -0.6282 - 0.1078j,-0.1012 + 1.1421j,
0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j, 0.0187 - 1.0596j, 0.3680 + 0.3680j, -1.0596 + 0.0187j,
-0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j, -0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j,
0.0187 - 1.0596j, -0.6282 - 0.1078j, -0.1012 + 1.1421j, 0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j,
0.0187 - 1.0596j, 0.3680 + 0.3680j, -1.0596 + 0.0187j, -0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j,
-0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j, 0.0187 - 1.0596j, -0.6282 - 0.1078j,-0.1012 + 1.1421j,
0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j, 0.0187 - 1.0596j, 0.3680 + 0.3680j, -1.0596 + 0.0187j,
-0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j, -0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j,
0.0187 - 1.0596j, -0.6282 - 0.1078j, -0.1012 + 1.1421j, 0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j,
0.0187 - 1.0596j, 0.3680 + 0.3680j, -1.0596 + 0.0187j, -0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j,
-0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j, 0.0187 - 1.0596j, -0.6282 - 0.1078j,-0.1012 + 1.1421j,
0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j, 0.0187 - 1.0596j, 0.3680 + 0.3680j, -1.0596 + 0.0187j,
-0.1078 - 0.6282j, 1.1421 - 0.1012j, 0.7360, 1.1421 - 0.1012j, -0.1078 - 0.6282j, -1.0596 + 0.0187j, 0.3680 + 0.3680j,
0.0187 - 1.0596j, -0.6282 - 0.1078j, -0.1012 + 1.1421j, 0 + 0.7360j, -0.1012 + 1.1421j, -0.6282 - 0.1078j,
0.0187 - 1.0596j,-1.2500, 0.0983 - 0.7808j, 0.7337 - 0.8470j, -0.7351 - 0.9210j, -0.0224 - 0.4302j, 0.6006 + 0.5923j,
-1.0186 + 0.1640j, -0.9751 + 0.1325j, -0.2803 + 1.2071j, -0.4516 + 0.1744j, -0.4825 - 0.6503j, 0.5565 - 0.1130j,
0.6577 - 0.7389j, -1.0501 - 0.5218j, -0.4577 - 0.3144j, 0.2953 - 0.7868j, 0.5000 + 0.5000j, 0.9539 + 0.0328j,
-0.1799 - 1.2853j, 0.4694 + 0.1195j, 0.1958 + 0.4683j, -1.0944 + 0.3790j, 0.0079 + 0.9200j,  0.4267 - 0.0326j,
0.7803 + 0.2071j, -0.3065 + 0.8494j, -0.9210 + 0.4414j, 0.4786 + 0.7017j, 0.1689 - 0.2231j, 0.7747 - 0.6624j,
0.3180 + 0.8893j, -0.0410 + 0.9626j, 1.2500, -0.0410 - 0.9626j, 0.3180 - 0.8893j, 0.7747 + 0.6624j, 0.1689 + 0.2231j,
0.4786 - 0.7017j, -0.9210 - 0.4414j, -0.3065 - 0.8494j, 0.7803 - 0.2071j, 0.4267 + 0.0326j, 0.0079 - 0.9200j,
-1.0944 - 0.3790j, 0.1958 - 0.4683j, 0.4694 - 0.1195j, -0.1799 + 1.2853j, 0.9539 - 0.0328j, 0.5000 - 0.5000j,
0.2953 + 0.7868j, -0.4577 + 0.3144j, -1.0501 + 0.5218j, 0.6577 + 0.7389j, 0.5565 + 0.1130j, -0.4825 + 0.6503j,
-0.4516 - 0.1744j, -0.2803 - 1.2071j, -0.9751 - 0.1325j, -1.0186 - 0.1640j, 0.6006 - 0.5923j, -0.0224 + 0.4302j,
-0.7351 + 0.9210j, 0.7337 + 0.8470j, 0.0983 + 0.7808j,-1.2500, 0.0983 - 0.7808j, 0.7337 - 0.8470j, -0.7351 - 0.9210j,
-0.0224 - 0.4302j, 0.6006 + 0.5923j, -1.0186 + 0.1640j, -0.9751 + 0.1325j, -0.2803 + 1.2071j, -0.4516 + 0.1744j,
-0.4825 - 0.6503j, 0.5565 - 0.1130j, 0.6577 - 0.7389j, -1.0501 - 0.5218j, -0.4577 - 0.3144j, 0.2953 - 0.7868j,
0.5000 + 0.5000j, 0.9539 + 0.0328j, -0.1799 - 1.2853j, 0.4694 + 0.1195j, 0.1958 + 0.4683j, -1.0944 + 0.3790j,
0.0079 + 0.9200j,  0.4267 - 0.0326j, 0.7803 + 0.2071j, -0.3065 + 0.8494j, -0.9210 + 0.4414j, 0.4786 + 0.7017j,
0.1689 - 0.2231j, 0.7747 - 0.6624j, 0.3180 + 0.8893j, -0.0410 + 0.9626j, 1.2500, -0.0410 - 0.9626j, 0.3180 - 0.8893j,
0.7747 + 0.6624j, 0.1689 + 0.2231j, 0.4786 - 0.7017j, -0.9210 - 0.4414j, -0.3065 - 0.8494j, 0.7803 - 0.2071j,
0.4267 + 0.0326j, 0.0079 - 0.9200j, -1.0944 - 0.3790j, 0.1958 - 0.4683j, 0.4694 - 0.1195j, -0.1799 + 1.2853j,
0.9539 - 0.0328j, 0.5000 - 0.5000j, 0.2953 + 0.7868j, -0.4577 + 0.3144j, -1.0501 + 0.5218j, 0.6577 + 0.7389j,
0.5565 + 0.1130j, -0.4825 + 0.6503j, -0.4516 - 0.1744j, -0.2803 - 1.2071j, -0.9751 - 0.1325j, -1.0186 - 0.1640j,
0.6006 - 0.5923j, -0.0224 + 0.4302j, -0.7351 + 0.9210j, 0.7337 + 0.8470j, 0.0983 + 0.7808j,-1.2500, 0.0983 - 0.7808j,
0.7337 - 0.8470j, -0.7351 - 0.9210j, -0.0224 - 0.4302j, 0.6006 + 0.5923j, -1.0186 + 0.1640j, -0.9751 + 0.1325j,
-0.2803 + 1.2071j, -0.4516 + 0.1744j, -0.4825 - 0.6503j, 0.5565 - 0.1130j, 0.6577 - 0.7389j, -1.0501 - 0.5218j,
-0.4577 - 0.3144j, 0.2953 - 0.7868j, 0.5000 + 0.5000j, 0.9539 + 0.0328j, -0.1799 - 1.2853j, 0.4694 + 0.1195j,
0.1958 + 0.4683j, -1.0944 + 0.3790j, 0.0079 + 0.9200j,  0.4267 - 0.0326j, 0.7803 + 0.2071j, -0.3065 + 0.8494j,
-0.9210 + 0.4414j, 0.4786 + 0.7017j, 0.1689 - 0.2231j, 0.7747 - 0.6624j, 0.3180 + 0.8893j, -0.0410 + 0.9626j)

gap_sample = (0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j,
0+0j, 0+0j, 0+0j, 0+0j,0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j,0+0j, 0+0j, 0+0j, 0+0j, 0+0j,
0+0j, 0+0j, 0+0j, 0+0j, 0+0j,0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j,0+0j, 0+0j, 0+0j, 0+0j,
0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j,0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j,0+0j, 0+0j, 0+0j,
0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j, 0+0j)
```

Listing 13.3: The wlan_utils.py Code File

13.2 gEDA Makefile

This appendix presents the geda Makefile that was used to aid with gEDA EDA, since gEDA does not provide an integrated development environment unlike most commercial EDAs.

```
# gEDA Makefile skeleton
# Copyright (C) 2010  Duncan Drennan (First Draft)
# Joseph Wamicha (Second Draft)

PROJNAME = rhinoexpansion
SHEETS = rhino-main.sch rhino-fmc.sch rhino-pmc.sch rhino-power.sch rhino-clock.sch
ELEMENTDIR = footprints
PCBNAME = rhinoexpansion
GSCH2PCBOPTS = --verbose --skip-m4
NEWSYMBOLS= symbols/generated/FMC-LPC-A.sym  symbols/generated/FMC-LPC-CTRL.sym  symbols/generated/FMC-LPC-GND.sym  symbols/generated/FMC-LPC-PWR.sym

notarget:
	@echo "Targets:"
	@echo ""
	@echo "${MAKE} open            Opens the schematics"
	@echo "${MAKE} attrib          Opens the attribute editor"
	@echo "${MAKE} renum           Numbers reference designators"
	@echo "${MAKE} bom             Creates a bom, 1 line per refdes"
	@echo "${MAKE} bom2            Creates a bom, 1 line per component type"
	@echo "${MAKE} partslist       Creates a bom with component count"
	@echo "${MAKE} drc             Runs a DRC against the schematic"
	@echo "${MAKE} netlist         Generates a gEDA netlist"
	@echo "${MAKE} pcb             Generates/updates the PCB file"
	@echo "${MAKE} openpcb         Open the PCB file"
	@echo "${MAKE} pcb-gtk         Open the PCB file"
	@echo "${MAKE} xgsch2pcb       Open the PCB file"
	@echo "${MAKE} net-tango       Generates a tango netlist"
	@echo "${MAKE} net-pads        Generate PADS compatible netlist"
	@echo "${MAKE} net-pins        Forward annotate pin/pad names from schematic to layout"
	@echo "${MAKE} net-protelII    Generate protelII compatible netlist"
	@echo "${MAKE} net-spice       Generate spice compatible netlist"
	@echo "${MAKE} net-spice-sdb   Generate enhanced spice compatible netlist"
	@echo "${MAKE} net-verilog     Generate verilog compatible netlist"
	@echo "${MAKE} net-vhdl        Generate vhdl compatible netlist"
	@echo "${MAKE} net-vams        Generate vams compatible netlist"
	@echo "${MAKE} net-switcap     Generate switcap compatible netlist"
	@echo "${MAKE} net-eagle       Generate eagle compatible netlist"
	@echo "${MAKE} printpdf        Creates a PDF of the schematics"
	@echo "${MAKE} printeps        Creates a eps file of the schematics"
	@echo "${MAKE} symcheck        Ensures new symbols are valid"
	#@echo "${MAKE} embedded        Embeds all gschem symbols into schematic for distribution"
	@echo "${MAKE} clean           Remove all temp/swap files. Recommend you do this before committing to code repo."

open: $(SHEETS)
	gschem $(SHEETS)

attrib: $(SHEETS)
	gattrib $(SHEETS)

symcheck: $(NEWSYMBOLS)
	gsymcheck $(NEWSYMBOLS)

renum: $(SHEETS)
	refdes_renum --pgskip $(SHEETS)

bom: $(SHEETS) attribs
	mkdir -p bom
	gnetlist -g bom -o bom/$(PROJNAME)_bom.csv $(SHEETS)

bom2: $(SHEETS)
	mkdir -p bom2
	gnetlist -g bom2 -o bom2/$(PROJNAME)_bom2.csv $(SHEETS)

partslist: $(SHEETS)
	mkdir -p partslist
	gnetlist -g partslist3 -o partslist/$(PROJNAME)_partslist.csv $(SHEETS)

drc: $(SHEETS)
	mkdir -p drc
	gnetlist -g drc2 -o drc/$(PROJNAME).drc $(SHEETS)

netlist: $(SHEETS)
	gnetlist -g geda -o $(PROJNAME)_geda.net $(SHEETS)
```

```
pcb: $(SHEETS)
	gsch2pcb $(GSCH2PCBOPTS) -d $(ELEMENTDIR) -o $(PCBNAME) $(SHEETS)


openpcb: $(PCBNAME).pcb
	pcb $(PCBNAME).pcb

pcb-gtk: $(PCBNAME).pcb
	pcb-gtk $(PCBNAME).pcb

xgsch2pcb: $(PCBNAME).pcb
	xgsch2pcb $(PCBNAME).pcb

net-tango: $(SHEETS)
	mkdir -p nets/tango
	gnetlist -g tango -o nets/tango/$(PROJNAME)_tango.net $(SHEETS)

net-pads: $(SHEETS)
	mkdir -p nets/pads
	gnetlist -g pads -o nets/pads/$(PROJNAME)_pads.net $(SHEETS)

net-pins: $(SHEETS)
	mkdir -p nets/pcbpins
	gnetlist -g pcbpins -o nets/pcbpins/$(PROJNAME)_pcbpins.net $(SHEETS)

net-protelII: $(SHEETS)
	mkdir -p nets/protelII
	gnetlist -g protelII -o $(PROJNAME)_protelII.net $(SHEETS)

net-spice: $(SHEETS)
	mkdir -p nets/spice
	gnetlist -g spice -o nets/spice/$(PROJNAME)_spice.net $(SHEETS)

net-spice-sdb: $(SHEETS)
	mkdir -p nets/spice-sdb
	gnetlist -g spice-sdb -o nets/spice-sdb/$(PROJNAME)_spice-sdb.net $(SHEETS)

net-verilog: $(SHEETS)
	mkdir -p nets/verilog
	gnetlist -g verilog -o nets/verilog/$(PROJNAME)_verilog.net $(SHEETS)

net-vhdl: $(SHEETS)
	mkdir -p nets/vhdl
	gnetlist -g vhdl -o nets/vhdl/$(PROJNAME)_vhdl.net $(SHEETS)

net-vams: $(SHEETS)
	mkdir -p nets/vams
	gnetlist -g vams -o nets/vams/$(PROJNAME)_vams.net $(SHEETS)

net-switcap: $(SHEETS)
	mkdir -p nets/switcap
	gnetlist -g switcap -o nets/switcap/$(PROJNAME)_switcap.net $(SHEETS)

net-eagle: $(SHEETS)
	mkdir -p nets/eagle
	gnetlist -g eagle -o nets/eagle/$(PROJNAME)_eagle.net $(SHEETS)

printpdf: $(PROJNAME).ps
	mkdir -p print
	ps2pdf -dOptimize=true -dUseFlateCompression=true -sPAPERSIZE=a4 $< $(PROJNAME).pdf
	mv *.ps *.pdf print
	mv *.ps *.pdf print

printeps: $(SHEETS)
	mkdir -p eps
	for S in $(SHEETS); \
	do \
	gschem -q -o $$S.ps -s /usr/share/gEDA/scheme/print.scm $$S; \
	ps2epsi $$S.ps $$S.eps; done
	mv *.eps *.ps eps

$(PROJNAME).ps: printps

printps: $(SHEETS)
	rm -f ONEPAGE.ps
	rm -f $(PROJNAME).ps
	for S in $(SHEETS); \
	do \
	gschem -q -o ONEPAGE.ps -s /usr/share/gEDA/scheme/print.scm $$S; \
	cat ONEPAGE.ps; done > $(PROJNAME).ps
	rm -f ONEPAGE.ps
```

```
#embedded: $(SHEETS)
#	mkdir -p "${PROJNAME}-dist"
#	gschlas -e $(SHEETS)

clean:
	rm -rf `find ./ -name "*~"`
```

Listing 13.4: The gEDA Makefile

13.3 The Edge Fed Microstrip Patch Antenna Python Script

This appendix presents the edge fed microstrip patch antenna python script that was used to generate our EBG patch antenna gerber and feko files.

```
#!/usr/bin/python

"""
    Python script for calculating edge-fed microstrip line patch antenna with EBG structures dimensions.
    References: MPAC Tcl program by Nathan Cummings

    TODO:
    1.) Calculate Surface resistance, radiation efficiency and bandwidth when EBG structures are present.
    2.) Add via calculations.
    3.) Determine formula for inset feed notch width.
"""

__author__ = "Joseph Wamicha"
__version__ = "$Revision: 1.0 $"
__date__ = "$Date: 2010/11/10 21:57:19 $"
__license__ = "Python"

import sys, getopt, os, math
from scipy.constants import constants as spconst
from ConfigParser import ConfigParser

class MPatchEBGCalc:
    # Constant Variables
    eta0 = 377
    gamma = 0.577216
    a2 = -0.16605
    a4 = 0.00761
    c2 = -0.0914153

    # Patch Variables
    eps_rel = 0 # Substrate Relative Permittivity or Dielectric Constant
    eps_eff = 0 # Substrate Effective Permittivity
    conductivity = 0 # Substrate Conductivity
    tan_loss = 0 # Substrate Tangent Loss
    h_sub = 0 # Substrate height
    Len = 0 # Patch Length
    dL = 0 # Length increase due to fringe magnetic fields along the length of the patch
    Leff = 0 # Effective Length of Patch due to Actual Length of Patch plus Fringe Fields.
    Width = 0 # Patch Width
    dW = 0 # Width increase due to fringe magnetic fields along the width of the patch
    Weff = 0 # Effective Width of Patch due to Actual Width of Patch plus Fringe Fields.
    fl = 0 # Patch Feed Inset Length
    fw = 0 # Patch Feed Notch Width
    board_len = 0 # Total Board Length
    board_width = 0 # Total Board Width
    radeff = 0 # Patch Radiation Efficiency
    bandwidth = 0 # Patch Bandwidth
    R_surface = 0 # Surface resistance of the patch
    directivity = 0

    # EBG Variables
    ebg_N = 0 # EBG rows along each length of patch margin
    ebg_len = 0 # EBG structure Length
    ebg_width = 0 # EBG structure Width
    ebg_cols = 0 # EBG columns along each length of patch margin
    discontinuity = 0 # Size of discontinuity or gap between each of the EBG structure elements

    # Other Variables
    lambda0 = 0 # Free space wavelength
    omega = 0 # Free space radial frequency
    f0 = 0 # Free space wavelength
    k0 = 0 # Free space wavenumber
    clearance = 5e-3 #board clearance eg clearance to allow for attachment of the SMA connector

    def __init__(self):
        print "New MPatchEBGCalc instance created."

    def __str__(self):
        return "This is an MPatchEBGCalc instance."

    def get_lambda0(self, f0):
        self.lambda0 = spconst.c / f0
        return self.lambda0
```

```
    def get_eps_eff(self, eps_rel, h_sub, Width):
        int1 = (eps_rel + 1.0) / 2.0
        int2 = (eps_rel - 1.0) / 2.0  (math.pow((1 + (10.0  h_sub / Width)), -0.5))
        self.eps_eff = int1 + int2
        return self.eps_eff

    def get_Width(self, eps_rel, f0):
        self.Width = spconst.c / ((2  f0)  math.sqrt((eps_rel + 1) / 2))
        return self.Width

    def get_dW(self, h_sub):
        self.dW = (math.log10(4.0) / spconst.pi)  h_sub
        return self.dW

    def get_Weff(self, Width, dW, h_sub):
        self.Weff = Width + (2.0  dW)
        return self.Weff

    def get_Leff(self, f0, eps_eff):
        self.Leff = spconst.c / (2  f0  math.sqrt(eps_eff))
        return self.Leff

    def get_dL(self, h_sub, eps_eff, Width):
        dL_1 = 0.412  h_sub  ((eps_eff + 0.3)/(eps_eff - 0.258))
        dL_2 = (Width/h_sub + 0.264) / (Width/h_sub + 0.813)
        self.dL = dL_1  dL_2
        return self.dL

    def get_Len(self, L_eff, dL):
        self.Len = L_eff - (2  dL)
        return self.Len

    def get_R_surface(self, f0, conductivity):
        self.R_surface = math.sqrt(2.0  spconst.pi  f0  spconst.mu_0 / (2.0  conductivity))
        return self.R_surface

    def get_omega(self, f0):
        self.omega = 2  spconst.pi  f0
        return self.omega

    def get_k0(self, f0, eps_rel, u_rel, omega):
        self.k0 = omega  math.sqrt(math.mu_0  u_rel  eps_rel  spconst.epsilon_0)
        return self.k0

    def get_p_param(self, k0, W_eff, L_eff):
        p_1 = 1 + ((self.a2 / 10.0)  math.pow((k0  W_eff), 2.0)) +
(math.pow(self.a2, 2.0) + (2  self.a4))  (3.0 / 560.0)  (math.pow((k0  W_eff), 4.0))
        p_2 = self.c2  (1.0 / 5.0)  (math.pow((k0  L_eff), 2)) +
self.a2  self.c2  (1.0 / 70.0)  (math.pow((k0  W_eff), 2.0))  (math.pow((k0  L_eff), 2))
        return p1 + p2

    def get_n1(self, eps_rel, u_rel):
        n1 = math.sqrt(eps_rel  u_rel)
        return n1

    def get_c1_param(self, n1):
        c1 = (1 / math.pow(n1, 2)) + ((2.0 / 5.0) / math.pow(n1, 4.0))
        return c1

    def get_dipole_radiated_power(self, lambda0, k0, h_sub, u_rel, c1_param):
        P_dipole = (1.0 / math.pow(lambda0, 2))  (math.pow((k0  h_sub), 2))  (80.0  math.pow(spconst.pi,2)  math.pow(u_rel,2)  c1_param)
        return P_dipole

    def get_s_param(self, eps_rel):
        s = math.sqrt(eps_rel - 1)
        return s

    def get_alpha0(self, s_param, k0, h_sub):
        alpha0 = s  math.tan(k0  h_sub  s_param)
        return alpha0

    def get_alpha1(self, s_param, k0, h_sub):
        alpha1 = (-1 / s_param)  (math.tan(k0  h_sub  s_param) + ((k0  h_sub  s_param) / (math.pow(math.cos(k0  h_sub  s_param), 2))))
        return alpha1

    def get_x0(self, eps_rel, alpha0, alpha1):
        x0_a = -1  math.pow(eps_rel, 2) + (alpha0  alpha1)
        x0_b = math.pow(eps_rel, 2) - (2  alpha0  alpha1) + math.pow(alpha0, 2)
        x0_c = math.pow(eps_rel, 2) - math.pow(alpha1, 2)
        x0 = 1 + ((x0_a + eps_rel  math.sqrt(x0_b)) / x0_c)
        return x0

    def get_x1(self, x0, eps_rel):
```

```
        x1 (math.pow(x0, 2) - 1) / (eps_rel - math.pow(x0, 2))
        return x1

    def get_surface_wave_power(self, k0, eps_rel, x0, x1, h_sub):
        pwr_a = self.eta0   math.pow(k0, 2) / 8.0
        pwr_b = eps_rel   math.pow((math.pow(x0, 2) - 1), 3.0 / 2.0)
        pwr_c = eps_rel   (1 + x1)
        pwr_d = (k0   h_sub)   math.pow((math.pow(x0, 2) - 1), 1.0 / 2.0)
        pwr_e = 1 + math.pow(eps_rel, 2)   x1
        P_surface_waves = pwr_a   ((pwr_b) / (pwr_c + (pwr_d   pwr_e)))
        return P_surface_waves

    def get_dipole_radiation_efficiency(self, P_dipole, P_surface_waves):
        radeff_dipole = power_dipole / (P_dipole + P_surface_waves)
        return radeff_dipole

    def get_radiation_efficiency(self, tan_loss, R_surface, u_rel, h_sub, lambda0, eps_rel, p_param, c1_param, L_eff, W_eff, dipole_radeff):
        radeff_1 = tan_loss + ((R_surface / (spconst.pi   self.eta0   u_rel))   (1 / (h_sub / lambda0)))
        radeff_2 = (3.0 / 16.0)   (eps_rel / (p_param   c1_param))   (L_eff / W_eff)   (1 / (h_sub / lambda0))
        self.radeff = radeff_dipole / (1 + radeff_dipole   radeff_1   radeff_2)
        return self.radeff

    def get_bandwidth(self, R_surface, u_rel, h_sub, lambda0, p_param, c1_param, radeff_dipole, tan_loss):
        bw_1 = (R_surface / (spconst.pi   self.eta0   u_rel))   (1 / (h_sub / lambda0))
        bw_2 = (16.0 / 3.0)   (p_param   c1_param / eps_rel)
        bw_3 = (h_sub / lambda0)   (W_eff / L_eff)   (1 / radeff_dipole)
        bw = (1 / math.sqrt(2.0))   (tan_loss + bw_1 + (bw_2   bw_3))
        self.bandwidth = "%2.2f" % bw
        return self.bandwidth

    def get_directivity(self, k0, n1, h_sub, p_param, c1_param, u_rel, eps_rel):
        D_arg = math.tan(k0   h_sub   n1) / (k0   h_sub   n1)
        D_1 = self.eta0 / (40.0   spconst.pi)   (1 / (p_param   c1_param))
        D_2 = math.pow(D_arg, 2)
        D_final = D_1   (D_2 / (1 + ((u_rel   D_2) / eps_rel)))
        self.directivity = "%2.2f" % (10   math.log10(D_final))
        return self.directivity

    def get_inset_feed_length(self, eps_rel, Len):
        fl1 = (0.001699   math.pow(eps_rel, 7)) + (0.13761   math.pow(eps_rel, 6)) + \\
              (6.1783   math.pow(eps_rel, 5)   -1) + (93.187   math.pow(eps_rel, 4)) + \\
              (682.69   math.pow(eps_rel, 3)   -1) + (2561.9   math.pow(eps_rel, 2)) + \\
              (4043   eps_rel   -1) + 6697
        self.fl = fl1   math.pow(10, -4)   Len/2
        return self.fl

    def get_feed_notch_width(self, eps_eff, f0, f_resonant):
        self.fw = (spconst.c / math.sqrt(2   eps_eff))   ((4.6   math.pow(10, -14))   (1 / (f_resonant - (f0 / 1.01))) )
        return self.fw

    # ebg_N is the number of ebg structures. Use discontinuity size to determine ebg_N.
    def get_ebgN(self, Len, Leff, discontinuity):
        # The total gap size will be twice (Leff - Len) to ensure even fringing fields don't get through
        # The waveguide cut-off frequency below which no transmission will occur will be the patch resonant frequency
        ebg_total_gap = (Leff - Len)
        #(ebg_N+1)   discontinuity = ebg_total_gap
        self.ebg_N = int(math.ceil(ebg_total_gap / discontinuity)) - 1
        return self.ebg_N

    def get_ebg_gap(self, Len, Leff, ebg_N):
        ebg_total_gap = (Leff - Len)
        self.ebg_gap = (ebg_total_gap / (ebg_N + 1))
        return self.ebg_gap

    # ebg len is a function of the gap between the ebg structures. Use ebg_N to determine ebg_len
    def get_ebg_len(self, Len, Leff, ebg_N):
        ebg_total_gap = (Leff - Len)
        #((ebg_N + 1)   discontinuity) + (ebg_N   ebg_len) = Leff
        # ebg_total_gap + (ebg_N   ebg_len) = Leff
        if ebg_N == 0:
            # then the discontinuity might be so high, that its impossible to have ebg structures
            return 0

        self.ebg_len = (Leff - ebg_total_gap) / ebg_N
        self.ebg_width = self.ebg_len
        return self.ebg_len

    # We'll assume ebg_width is equal to ebg_len. We're not really concerned about
    # controlling the ebg_width since we aren't trying to suppress TM(0,X) modes.
    def get_board_width(self, ebg_width, ebg_cols, Weff, discontinuity):
        self.board_width = (self.dW   2) + (self.discontinuity   2) +
self.Width + (ebg_width   (ebg_cols)   2) + (discontinuity   (ebg_cols)   2)
        return self.board_width
```

```
241
    def get_board_len(self, Leff, discontinuity):
        #print "dL=%f, discontinuity=%f, clearance=%f" % (self.dL, (2    self.discontinuity), self.clearance)
        self.board_len = self.clearance + (self.dL    2) + ((2    self.discontinuity)    2) + self.Len
        return self.board_len
246
class PatchDetailsWriter:
    "A class for writing patch detail properties files."
    def __init__(self):
        self.filename = "mspebg"
251     self.iterable = []

    def set_write_file(self, filename):
        self.filename = filename+".txt"

256 def set_iterable(self, iterable):
        self.iterable = iterable

    def produce_details_file(self):
        if self.iterable == None:
261         return

        file = open(self.filename, "w")
        fileoutput = "[config]\n"
        for item in self.iterable:
266         fileoutput += item[0]
            fileoutput += "="
            fileoutput += str(item[1])
            fileoutput += '\n'

271     file.write(fileoutput)
        file.close()

class PatchDetailsEater:
    "A class for eating patch details properties files."
276
    def __init__(self):
        self.filename = ""
        self.properties = []

281 def set_read_file(self, filename):
        self.filename = filename

    def get_properties(self):
        file = open(self.filename)
        for readline in file.readlines():
286         line = readline.decode("utf-8")
            if line[0] != '\n' and line[0] != '#':
                i = string.find(line, '=')
                if i > 0:
291                 self.properties.append([line[:i], line[i + 1:-1]])


def usage ():
    print ""
296 print "For help: "
    print "mpatchebgcalc.py --help [-h]"
    print ""
    print "Usage: "
    print ""
301 print "mpatchebgcalc.py --frequency [-f] --dielectric-constant [-d] --substrate-height [-s]
--tangent-loss [-t] --patch-conductivity [-c] --ebg-cols [-ec] --minimum-discontinuity [-m]
--frequency-resonant [-r] --out-file [-o] --filetype [-y] --in-file [-i] --help [-h]"
    print ""
    print "--frequency [-f] is a mandatory argument."
306 print "--dielectric-constant [-d] is a mandatory argument."
    print "--substrate-height [-s] is a mandatory argument."
    print "--tangent-loss [-t] is a mandatory argument."
    print "--patch-conductivity [-c] is a mandatory argument."
    print "If the --ebg-cols [-e] option is not provided, 10 ebg rows will be used."
311 print "If the --minimum-discontinuity [-m] option is not provided, a minimum discontinuity
of 3.82e-2 (15 mils) will be used."
    print "If the --frequency-resonant [-r] option is not provided, the patch frequency won't
be adjusted when determining feed notch width since the 'actual' resonant frequency of the
patch will be assumed to be equal to the calculated resonant frequency of the patch
316 (no adjustments would be made to vary the actual resonant frequency using notch width variations)."
    print "If the --out-file [-o] option is not provided, the prefix name mpatchebgcalc will be used by default."
    print "If the --filetype [-y] option is not provided, the patch antenna gerber files will be output by default."
    print "If the --in-file [-i] option is not provided, non will be used by the mpatchebgcalc.py script."
    print ""
321 sys.exit(0)

def help ():
```

```
    print ""
    print "Python␣script␣for␣calculating␣the␣best␣Patch␣antenna,␣Edge-fed␣Microstrip␣line␣and␣EBG␣dimensions."
    print "All␣input␣units␣should␣be␣in␣meters."
    print ""
    print "Script␣command␣arguments:"
    print "To␣calculate␣the␣dimensions␣for␣a␣2.4GHz␣patch␣antenna:␣"
    print "--frequency␣2.4e9"
    print "[-f]"
    print "The␣frequency␣option␣is␣mandatory."
    print ""
    print "To␣specify␣the␣dielectric␣constant␣of␣the␣substrate:␣"
    print "--dielectric-constant␣3.38"
    print "[-d]"
    print "The␣substrate␣dielectric␣constant␣option␣is␣mandatory."
    print ""
    print "To␣specify␣the␣substrate␣height␣option␣in␣meters:␣"
    print "--substrate-height␣8.13e-3"
    print "[-s]"
    print "The␣substrate␣height␣option␣is␣mandatory."
    print ""
    print "To␣specify␣the␣tangent␣loss:␣"
    print "--tangent-loss"
    print "[-t]"
    print "The␣tangent␣loss␣option␣is␣mandatory."
    print ""
    print "To␣specify␣the␣conductivity␣of␣the␣patch:␣"
    print "--patch-conductivity␣5.69e7"
    print "[-c]"
    print "The␣patch␣conductivity␣option␣is␣mandatory."
    print ""
    print "To␣specify␣the␣number␣of␣ebg␣columns␣to␣use␣around␣each␣margin␣along␣the␣length␣of␣the␣patch:␣"
    print "--ebg-cols␣2"
    print "[-e]"
    print "The␣number␣of␣ebg␣columns␣on␣each␣side␣of␣the␣patch,␣along␣the␣patch␣length."
    print ""
    print "To␣specify␣the␣discontinuity␣size␣or␣gap␣between␣each␣of␣the␣ebg␣structures:␣"
    print "--minimum-discontinuity␣3.81e-3"
    print "[-m]"
    print "The␣minimum␣discontinuity␣size␣(or␣gap)␣allowed␣between␣the␣ebg␣structures␣in␣meters."
    print ""
    print "To␣specify␣what␣actual␣resonant␣frequency␣the␣patch␣is␣experiencing␣so␣appropriate
adjustments␣to␣the␣feed␣notch␣width␣can␣be␣made:␣"
    print "␣␣--frequency-resonant␣"
    print "␣␣-␣To␣use␣the␣calculated␣frequency␣when␣determining␣the␣feed␣notch␣width,␣leave␣out␣this␣argument."
    print "[-r]"
    print ""
    print "To␣specify␣the␣out␣file␣prefix␣name:␣"
    print "--out-file␣my2.4GhzPatch"
    print "[-o]"
    print ""
    print "To␣specify␣the␣types␣of␣files␣to␣output␣for␣the␣patch␣antenna:␣"
    print "--filetype␣[gerber/feko]"
    print "␣␣-␣To␣display␣the␣patch␣antenna␣gerber␣files␣(this␣is␣the␣default␣option):␣"
    print "␣␣--filetype␣gerber"
    print "␣␣-␣To␣display␣the␣patch␣antenna␣feko␣files␣for␣modelling:␣"
    print "␣␣--filetype␣feko"
    print "[-y]"
    print ""
    print "To␣specify␣the␣in␣file␣name␣which␣contains␣the␣patch␣properties␣that␣will␣be
over-ridden␣when␣writing␣out␣the␣Feko␣.pre␣file␣or␣Gerber␣.gbr␣file:␣"
    print "--in-file␣myPatchProperties"
    print "[-i]"
    print ""
    print "Examples:␣"
    print "./mpatchebgcalc.py␣--frequency␣2.4e9␣--dielectric-constant␣3.38␣--substrate-height␣8.13e-3
--tangent-loss␣2.7e-3␣--patch-conductivity␣5.69e7␣--ebg-cols␣2␣--minimum-discontinuity␣1e-3
--out-file␣mspebg␣--filetype␣gerber"
    print "./mpatchebgcalc.py␣-f␣2.4e9␣-d␣3.38␣-s␣8.13e-3␣-t␣2.7e-3␣-c␣5.69e7␣-e␣2␣-m␣1e-3␣-o␣mspebg␣-y␣gerber"
    print "Override␣patch␣calculated␣dimensions␣using␣an␣in␣file:"
    print "./mpatchebgcalc.py␣-f␣2.4e9␣-d␣3.38␣--substrate-height␣8.13e-3␣--tangent-loss␣2.7e-3
--patch-conductivity␣5.69e7␣--ebg-cols␣2␣--minimum-discontinuity␣5.08e-3␣--out-file␣mspebg␣--filetype
gerber␣--in-file␣mspebg.txt"
    print "or"
    print "./mpatchebgcalc.py␣-f␣2.4e9␣-d␣3.38␣-s␣8.13e-3␣-t␣2.7e-3␣-c␣5.69e7␣-e␣2␣-m␣5.08e-3
-o␣mspebg␣-y␣gerber␣-i␣mspebg.txt"
    usage()

def is_number(s):
    try:
        float(s)
        return True
    except ValueError:
        return False
```

```
def meters_to_inches(value):
    return "%.4f" % (value * 39.3700787402)

def gerber_inches(value):
    gerber_value = meters_to_inches(value)
    return gerber_value.replace(".", "")

def meters_to_mm(value):
    return "%.4f" % (value * 1e3)

def feko_mm(value):
    feko_value = meters_to_mm(value)
    return feko_value

def write_file(filename, text):
    "Write text to file. No append."
    file_out = open(filename, "w")
    file_out.write(text)
    file_out.close()

def write_file_append(filename, text):
    "Write text to file. Append."
    file_out = open(filename, "a+")
    file_out.write(text)
    file_out.close()

def write_file_definition(mspatchebg, filename):
    iterables = [ ["Frequency", mspatchebg.f0], ["Dielectric-Constant", mspatchebg.eps_rel],
["Substrate-Height", mspatchebg.h_sub], \
                  ["Patch-Length", mspatchebg.Len], ["Patch-Width", mspatchebg.Width],
["Feed-Inset-Length", mspatchebg.fl], ["Feed-Notch-Width", mspatchebg.fw], \
                  ["EBG-Length", mspatchebg.ebg_len], ["EBG-Width", mspatchebg.ebg_width],
["EBG-Rows", mspatchebg.ebg_N], ["EBG-Columns", mspatchebg.ebg_cols], \
                  ["Board-Length", mspatchebg.board_len], ["Board-Width", mspatchebg.board_width],
["Radiation-Efficiency-Patch", mspatchebg.radeff], \
                  ["Bandwidth-Patch", mspatchebg.bandwidth], ["Directivity-Patch", mspatchebg.directivity] ]

    details_writer = PatchDetailsWriter()
    details_writer.set_write_file(filename)
    details_writer.set_iterable(iterables)
    details_writer.produce_details_file()

def write_gerber(mspatchebg, filename):

    x = 25.758139e-3
    y = 22.87016e-3

    gerber_bottom = []
    gerber_bottom.append("G04 ======= Begin FILE IDENTIFICATION ======")
    gerber_bottom.append("G04 File Format:  Gerber RS274X ")
    gerber_bottom.append("G04 ======= End FILE IDENTIFICATION ======")
    gerber_bottom.append("%FSLAX24Y24*%")
    gerber_bottom.append("%MOIN*%")
    gerber_bottom.append("%SFA1.0000B1.0000*%")
    gerber_bottom.append("%OFA0.0B0.0*%")
    gerber_bottom.append("%ADD10C,0.000004*%")
    gerber_bottom.append("%LNMSEBG-BOT*%")
    gerber_bottom.append("%IPPOS*%")
    gerber_bottom.append("%LPD*%")
    gerber_bottom.append("G75 ")
    gerber_bottom.append("G36 ")
    print "\n----------\nDrawing bottom layer:"
    print "1. x=%f, y=%f" % (x, y)
    gerber_bottom.append("G01X" + gerber_inches(x) + "Y" + gerber_inches(y) + "D02 ")
    y += mspatchebg.board_len
    print "2. x=%f, y=%f" % (x, y)
    gerber_bottom.append("G01Y" + gerber_inches(y) + "D01 ")
    x += mspatchebg.board_width
    print "3. x=%f, y=%f" % (x, y)
    gerber_bottom.append("G01X" + gerber_inches(x) + "D01 ")
    y = 22.87016e-3
    print "4. x=%f, y=%f" % (x, y)
    gerber_bottom.append("G01Y" + gerber_inches(y) + "D01 ")
    x = 25.758139e-3
    print "5. x=%f, y=%f \n----------" % (x, y)
    gerber_bottom.append("G01X" + gerber_inches(x) + "D01 ")
    gerber_bottom.append("G37 ")
    gerber_bottom.append("M02 ")
    write_file(filename + "-bot.gbr", "\n".join(gerber_bottom))

    x = 25.758139e-3
    y = 22.87016e-3
    gerber_top = []
```

```
    gerber_top.append("")
491 gerber_top.append("G04 ====== Begin FILE IDENTIFICATION =====")
    gerber_top.append("G04 File Format:  Gerber RS274X ")
    gerber_top.append("G04 ====== End FILE IDENTIFICATION =====")
    gerber_top.append("%FSLAX24Y24*%")
    gerber_top.append("%MOIN*%")
496 gerber_top.append("%SFA1.0000B1.0000*%")
    gerber_top.append("%OFA0.0B0.0*%")
    gerber_top.append("%ADD10C,0.000004*%")
    gerber_top.append("%LNMSEBG-TOP*%")
    gerber_top.append("%IPPOS*%")
501 gerber_top.append("%LPD*%")
    gerber_top.append("G75*")
    gerber_top.append("G36*")

    #"""
506 #y += mspatchebg.fl + mspatchebg.clearance + (2    mspatchebg.discontinuity)
    y = 22.87016e-3 + mspatchebg.fl + mspatchebg.dL + (2    mspatchebg.discontinuity)
+ mspatchebg.clearance
    #print "dL=%f, discontinuity=%f, clearance=%f" % (mspatchebg.dL,
(2    mspatchebg.discontinuity), mspatchebg.clearance)
511 #print "---> distance bot: %f" % (mspatchebg.dL + (2    mspatchebg.discontinuity)
+ mspatchebg.clearance)
    #print "---> distance bottom be: %f" % (mspatchebg.dL + (2    mspatchebg.discontinuity)
+ mspatchebg.clearance)
    #x += mspatchebg.discontinuity + (mspatchebg.ebg_width    (mspatchebg.ebg_cols))
516 + (mspatchebg.discontinuity    (ebg_cols + 1)) + mspatchebg.Weff/2 - mspatchebg.fw/2
    x = 25.758139e-3 + mspatchebg.board_width/2
    x -= mspatchebg.fw/2
    print "Drawing top layer:"
    print "1. x=%f, y=%f" % (x, y)
521 gerber_top.append("G01X" + gerber_inches(x) + "Y" + gerber_inches(y) + "D02*")
    x -= mspatchebg.fw
    print "2. x=%f, y=%f" % (x, y)
    gerber_top.append("G01X" + gerber_inches(x) + "D01*")
    y -= mspatchebg.fl
526 print "3. x=%f, y=%f" % (x, y)
    gerber_top.append("G01Y" + gerber_inches(y) + "D01*")
    x -= mspatchebg.Width/2
    x += mspatchebg.fw + mspatchebg.fw/2
    print "4. x=%f, y=%f" % (x, y)
531 gerber_top.append("G01X" + gerber_inches(x) + "D01*")
    y += mspatchebg.Len
    print "5. x=%f, y=%f" % (x, y)
    gerber_top.append("G01Y" + gerber_inches(y) + "D01*")
    x += mspatchebg.Width
536 print "6. x=%f, y=%f" % (x, y)
    gerber_top.append("G01X" + gerber_inches(x) + "D01*")
    y -= mspatchebg.Len
    print "7. x=%f, y=%f" % (x, y)
    gerber_top.append("G01Y" + gerber_inches(y) + "D01*")
541 x -= mspatchebg.Width/2
    x += mspatchebg.fw + mspatchebg.fw/2
    print "8. x=%f, y=%f" % (x, y)
    gerber_top.append("G01X" + gerber_inches(x) + "D01*")
    y += mspatchebg.fl
546 print "9. x=%f, y=%f" % (x, y)
    gerber_top.append("G01Y" + gerber_inches(y) + "D01*")
    x -= mspatchebg.fw
    print "10. x=%f, y=%f" % (x, y)
    gerber_top.append("G01X" + gerber_inches(x) + "D01*")
551 y -= mspatchebg.fl + mspatchebg.dL + (2    mspatchebg.discontinuity) + mspatchebg.clearance
    print "11. x=%f, y=%f" % (x, y)
    gerber_top.append("G01Y" + gerber_inches(y) + "D01*")
    x -= mspatchebg.fw
    print "12. x=%f, y=%f" % (x, y)
556 gerber_top.append("G01X" + gerber_inches(x) + "D01*")
    y += mspatchebg.fl + mspatchebg.dL + (2    mspatchebg.discontinuity) + mspatchebg.clearance
    print "13. x=%f, y=%f \n----------" % (x, y)
    gerber_top.append("G01Y" + gerber_inches(y) + "D01*")
    #"""
561

    print "---->>>> EBG COLUMNS: %f" % mspatchebg.ebg_cols
    if mspatchebg.ebg_N == 0 or mspatchebg.ebg_cols == 0:
        print "Either the discontinuity is too large to have EBG structures for
this patch or 0 ebg columns has been set."
566 else:
        #"""
        # Begin drawing the EBG structures...
        # Draw EBG structures to the left of the patch...
        print "Drawing left EBGs:"
571     for i in xrange(0, mspatchebg.ebg_N, 1):
            for j in xrange(0, mspatchebg.ebg_cols, 1):
```

```
                x = (25.758139e-3 + mspatchebg.discontinuity) + mspatchebg.discontinuity
+ (mspatchebg.discontinuity    (j)) + (mspatchebg.ebg_width    j)
                y = (22.87016e-3 + mspatchebg.clearance + (2    mspatchebg.discontinuity))
+ (mspatchebg.discontinuity    (i + 1)) + (mspatchebg.ebg_len    i)
                print "--------\nLeft. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01X" + gerber_inches(x) + "Y" + gerber_inches(y) + "D02 ")
                x += mspatchebg.ebg_width
                print "Left. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01X" + gerber_inches(x) + "D01 ")
                y += mspatchebg.ebg_len
                print "Left. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01Y" + gerber_inches(y) + "D01 ")
                x -= mspatchebg.ebg_width
                print "Left. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01X" + gerber_inches(x) + "D01 ")
                y -= mspatchebg.ebg_len
                print "Left. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01Y" + gerber_inches(y) + "D01 ")
        print "--------"

        # Draw EBG structures to the right of the patch...
        print "Drawing Right EBGs:"
        for i in xrange(0, mspatchebg.ebg_N, 1):
            for j in xrange(0, mspatchebg.ebg_cols, 1):
                x = (25.758139e-3 + mspatchebg.board_width / 2 + mspatchebg.Width / 2)
+ mspatchebg.dW + (mspatchebg.discontinuity    (j)) + (mspatchebg.ebg_width    j)
                y = (22.87016e-3 + mspatchebg.clearance + (2    mspatchebg.discontinuity))
+ (mspatchebg.discontinuity    (i + 1)) + (mspatchebg.ebg_len    i)
                print "--------\nRight. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01X" + gerber_inches(x) + "Y" + gerber_inches(y) + "D02 ")
                x += mspatchebg.ebg_width
                print "Right. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01X" + gerber_inches(x) + "D01 ")
                y += mspatchebg.ebg_len
                print "Right. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01Y" + gerber_inches(y) + "D01 ")
                x -= mspatchebg.ebg_width
                print "Right. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01X" + gerber_inches(x) + "D01 ")
                y -= mspatchebg.ebg_len
                print "Right. i=%d, j=%d : x=%f, y=%f" % (i, j, x, y)
                gerber_top.append("G01Y" + gerber_inches(y) + "D01 ")
        print "--------"
        #"""

    gerber_top.append("G37 ")
    gerber_top.append("M02 ")
    #write_file("mpebg-top.gbr", "\n".join(gerber_top) )
    write_file(filename + "-top.gbr", "\n".join(gerber_top))

def write_feko(mspatchebg, filename):
    prefeko = []
    prefeko.append("  PREFEKO input file generated by mspatchebgcalc.py version " + __version__)
    prefeko.append("\n")
    prefeko.append("  Scaling factor: all dimensions below in mm")
    prefeko.append("SF: 1 : : : : : 0.001")
    prefeko.append("\n")
    prefeko.append("  Constants")
    prefeko.append("\n")
    prefeko.append("#eta0 = 377")
    prefeko.append("#gamma = 0.577216")
    prefeko.append("\n")
    prefeko.append("  Patch and Substrate Variables")
    prefeko.append("#eps_rel = %f" % mspatchebg.eps_rel)
    prefeko.append("#eps_eff = %f" % mspatchebg.eps_eff)
    prefeko.append("#conductivity = %f" % mspatchebg.conductivity)
    prefeko.append("#tan_loss = %f" % mspatchebg.tan_loss)
    prefeko.append("#h_sub = %s" % feko_mm(mspatchebg.h_sub))
    prefeko.append("#Len_x = %s" % feko_mm(mspatchebg.Len))
    prefeko.append("#dL = %s" % feko_mm(mspatchebg.dL))
    prefeko.append("#Leff = %s" % feko_mm(mspatchebg.Leff))
    prefeko.append("#Width_y = %s" % feko_mm(mspatchebg.Width))
    prefeko.append("#dW = %s" % feko_mm(mspatchebg.dW))
    prefeko.append("#Weff = %s" % feko_mm(mspatchebg.Weff))
    prefeko.append("  Feed location variables")
    prefeko.append("#fl = %s" % feko_mm(mspatchebg.fl))
    prefeko.append("#fw = %s" % feko_mm(mspatchebg.fw))
    prefeko.append("  Board width")
    prefeko.append("#board_width_y = %s" % feko_mm(mspatchebg.board_width))
    prefeko.append("  Board length")
    prefeko.append("#board_len_x = %s" % feko_mm(mspatchebg.board_len))
    prefeko.append("#radeff = %s" % feko_mm(mspatchebg.radeff))
    prefeko.append("#bandwidth = %s" % feko_mm(mspatchebg.bandwidth))
```

```
    prefeko.append("#R_surface = %s" % feko_mm(mspatchebg.R_surface))
    prefeko.append("\n")
    prefeko.append("  EBG Variables")
    prefeko.append("#ebg_N = %f" % mspatchebg.ebg_N)
    prefeko.append("#ebg_len = %s" % feko_mm(mspatchebg.ebg_len))
    prefeko.append("#ebg_width = %s" % feko_mm(mspatchebg.ebg_width))
    prefeko.append("#ebg_cols = %f" % mspatchebg.ebg_cols)
    prefeko.append("#discontinuity = %s" % feko_mm(mspatchebg.discontinuity))
    prefeko.append("\n")
    prefeko.append("  Other Variables")
    prefeko.append("#clearance = %s" % feko_mm(mspatchebg.clearance))
    prefeko.append("  Frequency (for the discretisation)")
    prefeko.append("#freq = %f" % mspatchebg.f0)
    prefeko.append("#lam = 1000  #c0 / #freq / sqrt(#eps_rel)      Wavelength in mm")
    prefeko.append("\n")
    prefeko.append("  Segmentation parameters")
    prefeko.append("#tri_len = #lam / 12")
    prefeko.append("#fine_tri = #lam / 16")
    prefeko.append("#segl = #lam / 15")
    prefeko.append("#segr = #fw / 2")
    prefeko.append("IP                                  #segr     #tri_len  #segl")
    prefeko.append("\n")
    prefeko.append("  Generate one half of the structure")
    prefeko.append("  Define the points")
    prefeko.append("#x0 = #Len_x - #fl")
    prefeko.append("#x1 = #Len_x/2 - #fl - #board_len_x/2")
    prefeko.append("#x2 = #Len_x/2 - #fl + #board_len_x/2")

    # Geometry points
    prefeko.append("  Geometry points")
    prefeko.append("DP: A : : : : : -#fl : 0 : 0")
    prefeko.append("DP: B : : : : : #x0 : 0 : 0")
    prefeko.append("DP: C : : : : : #x0 : #Width_y/2 : 0")
    prefeko.append("DP: D : : : : : 0 : 0 : 0")
    prefeko.append("DP: E : : : : : -#fl : #Width_y/2 : 0")
    prefeko.append("DP: F : : : : : #x2 : 0 : 0")
    prefeko.append("DP: G : : : : : #x2 : #board_width_y/2 : 0")
    prefeko.append("DP: H : : : : : #x1 : #board_width_y/2 : 0")
    prefeko.append("DP: I : : : : : #x1 : 0 : 0")
    prefeko.append("DP: J : : : : : #x2 : 0 : -#h_sub")
    prefeko.append("DP: K : : : : : #x2 : #board_width_y/2 : -#h_sub")
    prefeko.append("DP: L : : : : : #x1 : #board_width_y/2 : -#h_sub")
    prefeko.append("DP: M : : : : : #x1 : 0 : -#h_sub")
    prefeko.append("DP: N : : : : : 0 : 0 : -#h_sub")
    prefeko.append("\n")

    #EBG points
    """
    prefeko.append("!!for #i = 0 to #ebg_N step 1")
    prefeko.append("!! for #j = 0 to #ebg_cols")
    prefeko.append("#x = #x2 - ((#clearance + (2   #discontinuity))
+ (#discontinuity   (#i + 1)) + (#ebg_len   #i))")
    prefeko.append("#y = #Width_y/2 + (#discontinuity   (#j + 1)) + (#ebg_width   #j)")
    prefeko.append("#tag = #i + #j")
    prefeko.append("DP: EBGR_A_#tag : : : : : #x : #y : 0")
    prefeko.append("DP: EBGR_B_#tag : : : : : #x+(0.5#ebg_len) : #y : 0")
    prefeko.append("DP: EBGR_C_#tag : : : : : #x+#ebg_len : #y : 0")
    prefeko.append("DP: EBGR_D_#tag : : : : : #x+#ebg_len : #y+#ebg_width : 0")
    prefeko.append("DP: EBGR_E_#tag : : : : : #x : #y+#ebg_width : 0")
    prefeko.append("!! next")
    prefeko.append("!!next")
    """

    """
    for i in xrange(0, mspatchebg.ebg_N, 1):
        for j in xrange(0, mspatchebg.ebg_cols, 1):
            x2 = mspatchebg.Len/2 - mspatchebg.fl + mspatchebg.board_len/2
            x = x2 - ((mspatchebg.clearance + (2   mspatchebg.discontinuity))
+ (mspatchebg.discontinuity   (i + 1)) + (mspatchebg.ebg_len   i))
            y = mspatchebg.Width/2 + (mspatchebg.discontinuity   (j + 1)) + (mspatchebg.ebg_width   j)
            prefeko.append("DP: A"+str(i)+""+str(j)+" : : : : : "+ feko_mm(x) +" : "+ feko_mm(y) +" : 0")
            prefeko.append("DP: B"+str(i)+""+str(j)+" : : : : : "+ feko_mm(x
+ mspatchebg.ebg_len/2) +" : "+ feko_mm(y) +" : 0")
            prefeko.append("DP: C"+str(i)+""+str(j)+" : : : : : "+ feko_mm(mspatchebg.ebg_len)
+" : "+ feko_mm(y) +" : 0")
            prefeko.append("DP: D"+str(i)+""+str(j)+" : : : : : "+ feko_mm(mspatchebg.ebg_len)
+" : "+ feko_mm(y + mspatchebg.ebg_width) +" : 0")
            prefeko.append("DP: E"+str(i)+""+str(j)+" : : : : : "+ feko_mm(x)+" : "
+ feko_mm(y + mspatchebg.ebg_width) +" : 0")
    prefeko.append("\n")
    """

    # Dielectric substrate Meshing
```

```
    prefeko.append("  Dielectric Substrate Meshing")
    prefeko.append("ME   1    0")
741 prefeko.append("BQ   B    F    G    C                                   #fine_tri")
    prefeko.append("BQ   C    G    H    E                                   #fine_tri")
    prefeko.append("BQ   E    H    I    A                                   #fine_tri")
    prefeko.append("BP   F    J    K    G")
    prefeko.append("BP   G    K    L    H")
746 prefeko.append("BP   H    L    M    I")
    prefeko.append("\n")

    prefeko.append("  Metallic Patch Meshing")
    prefeko.append("ME   1    0    1")
751 prefeko.append("BT   D    B    C              #fine_tri #fine_tri #fine_tri")
    prefeko.append("BQ   D    C    E    A         #fine_tri #fine_tri #fine_tri #fine_tri")
    prefeko.append("\n")

    """
756 prefeko.append("  EBG Patches Meshing")
    prefeko.append("!! for #i = 0 to #ebg_N step 1")
    prefeko.append("!!  for #j = 0 to #ebg_cols")
    prefeko.append("#tag = #i + #j")
    prefeko.append("ME   1    0    1")
761 prefeko.append("BT   EBGR_A_#tag    EBGR_B_#tag    EBGR_E_#tag #fine_tri #fine_tri #fine_tri")
    prefeko.append("BQ   EBGR_B_#tag    EBGR_C_#tag    EBGR_D_#tag    EBGR_E_#tag  #fine_tri #fine_tri #fine_tri #fine_tri")
    prefeko.append("!!  next")
    prefeko.append("!! next")
    prefeko.append("\n")
766 """

    """
    prefeko.append("  EBG Patches Meshing")
    for i in xrange(0, mspatchebg.ebg_N, 1):
771     for j in xrange(0, mspatchebg.ebg_cols, 1):
            #prefeko.append("ME   1    0    1")
            prefeko.append("BT   A"+str(i)+""+str(j)+"    B"+str(i)+""+str(j)+"    E"+str(i)+""+str(j)
+"              #fine_tri #fine_tri #fine_tri")
            prefeko.append("BQ   B"+str(i)+""+str(j)+"    C"+str(i)+""+str(j)+"    D"+str(i)+""+str(j)
776 +"   E"+str(i)+""+str(j)+"          #fine_tri #fine_tri #fine_tri #fine_tri")
    prefeko.append("\n")
    """

    prefeko.append("  Metallic ground plane")
781 prefeko.append("BT   N    K    J")
    prefeko.append("BQ   N    M    L    K")
    prefeko.append("\n")
    prefeko.append("  Symmetry to create the full structure")
    prefeko.append("SY   1    0    3    0")
786 prefeko.append("\n")
    prefeko.append("  Feed wire (will be a single segment) with label 1")
    prefeko.append("LA   1")
    prefeko.append("BL   N    D")
    prefeko.append("\n")
791 prefeko.append("  End of geometry input")
    prefeko.append("EG   1    0    0    0    0")
    prefeko.append("\n")
    prefeko.append("  Dielectric properties")
    prefeko.append("DI   1                        #eps_rel    1")
796 prefeko.append("\n")
    prefeko.append("  Frequency")
    prefeko.append("FR   1    0                   #freq")
    prefeko.append("\n")
    prefeko.append("  Voltage source at the wire centre with impressed power")
801 prefeko.append("PW   1                        1")
    prefeko.append("A1   0    1                   1         0")
    prefeko.append("\n")
    prefeko.append("  Far-field pattern")
    prefeko.append("FF   1    73   1    1         0         0         5")
806 prefeko.append("FF   1    73   1    1         0         90        5")
    prefeko.append("\n")
    prefeko.append("  Compute the radiated power in the far-field (only one half)")
    prefeko.append("FF   3    37   37   0         0         0         5         5")
    prefeko.append("")
811 prefeko.append("  End")
    prefeko.append("EN")

    write_file(filename + ".pre", "\n".join(prefeko))

816 def main():

    try:
        opts, args = getopt.gnu_getopt(sys.argv[1:], "f:d:s:t:c:e:m:r:o:y:i:h",
                                   [ "frequency=", "dielectric-constant=", "substrate-height=",
821 "tangent-loss=", "patch-conductivity=", "ebg-cols=", "minimum-discontinuity=", "frequency-resonant=",
```

```
"out-file=", "filetype=", "in-file=", "help"])
    except getopt.GetoptError, err:
        print str(err)
        usage()
        sys.exit(2)

    if len(opts) < 3:
        usage()
        sys.exit()

    "Initialize all needed variables..."
    f0 = 0.0
    eps_rel = 0.0
    h_sub = 0.0
    conductivity = 0.0
    tan_loss = 0.0
    ebg_cols = 0
    discontinuity = 3.81e-3
    f_resonant = 0
    outfile = "mpatchebgcalc"
    outfiletype = "gerber"
    infile = None


    for o, v in opts:
        if o in ("-h", "--help"):
            help()
            sys.exit()

        if (o in ("-f", "--frequency")):
            f0 = float(v)

        if (o in ("-d", "--dielectric-constant")):
            eps_rel = float(v)

        if (o in ("-s", "--substrate-height")):
            h_sub = float(v)

        if (o in ("-t", "--tangent-loss")):
            tan_loss = float(v)

        if (o in ("-c", "--patch-conductivity")):
            conductivity = float(v)

        if (o in ("-e", "--ebg-cols")):
            ebg_cols = int(v)

        if (o in ("-m", "--minimum-discontinuity")):
            discontinuity = float(v)

        if (o in ("-r", "--frequency-resonant")):
            f_resonant = float(v)

        if (o in ("-o", "--out-file")):
            outfile = v

        if (o in ("-y", "--filetype")):
            outfiletype = v

        if (o in ("-i", "--in-file")):
            infile = v

    "frequency is a mandatory argument"
    if f0 == "" or is_number(f0) == False:
        print "Invalid frequency option."
        sys.exit(2)
    else:
        print "f0=%s" % f0

    if eps_rel == "" or is_number(eps_rel) == False:
        print "Invalid dielectric option."
        sys.exit(2)
    else:
        print "Epsilon_r=%s" % eps_rel

    if h_sub == "" or is_number(h_sub) == False:
        print "Invalid subsrate height option."
        sys.exit(2)
    else:
        print "Height_sub=%s" % h_sub

    if tan_loss == "" or is_number(tan_loss) == False:
        print "Invalid tangent loss option."
```

```
        sys.exit(2)
    else:
        print "Tan Loss=%s" % tan_loss

    if conductivity == "" or is_number(conductivity) == False:
        print "Invalid conductivity option."
        sys.exit(2)
    else:
        print "Conductivity=%s" % conductivity

    if ebg_cols == "" or (is_number(ebg_cols) == False) or (is_number(ebg_cols) == True and ebg_cols < 0):
        print "Invalid ebg columns value. Using default value of 2"
        ebg_cols = 2
    else:
        print "Ebg cols=%s" % ebg_cols

    if is_number(discontinuity) == False:
        print "Invalid minimum discontinuity or ebg gap value. Using default value of 3.81e-3"
        discontinuity = 3.81e-3
    else:
        print "Discontinuity=%s" % discontinuity

    if is_number(f_resonant) == False:
        print "Invalid --frequency-resonant option. Assuming calculated resonant frequency is equal to actual resonant frequency."
        f_resonant = 0
    else:
        print "Frequency resonant=%s" % f_resonant

    if outfile == "":
        print "No outfile specified. Using default value of mpatchebgcalc."
        outfile = "mpatchebgcalc"
    else:
        print "Out file=%s" % outfile

    if outfiletype == "" :
        print "No outfiletype specified. Using default value of gerber."
        outfiletype = "gerber"
    else:
        print "Outfile type=%s" % outfiletype

    if infile == "":
        print "No infile specified for over-riding patch properties."
        infile = None
    else:
        print "Infile=%s" % infile

    # Set Mpatch dimensions from arguments
    mspatchebg = MPatchEBGCalc()
    mspatchebg.f0 = f0
    mspatchebg.eps_rel = eps_rel
    mspatchebg.tan_loss = tan_loss
    mspatchebg.conductivity = conductivity
    mspatchebg.ebg_cols = ebg_cols
    mspatchebg.discontinuity = discontinuity
    mspatchebg.h_sub = h_sub

    # Begin dimension calculations...
    lambda0 = mspatchebg.get_lambda0(f0)
    print "Free space wavelength=%f" % lambda0

    Width = mspatchebg.get_Width(eps_rel, f0)
    print "Patch width=%f" % Width

    eps_eff = mspatchebg.get_eps_eff(eps_rel, h_sub, Width)
    print "Effective permittivity=%f" % eps_eff

    dW = mspatchebg.get_dW(h_sub)
    print "Width Extension=%f" % dW

    Weff = mspatchebg.get_Weff(Width, dW, h_sub)
    print "Effective Width=%f" % Weff

    dL = mspatchebg.get_dL(h_sub, eps_eff, Width)
    print "Length Extension=%f" % dL

    Leff = mspatchebg.get_Leff(f0, eps_eff)
    print "Effective Length=%f" % Leff

    Len = mspatchebg.get_Len(Leff, dL)
    print "Patch Length=%f" % Len

    fl = mspatchebg.get_inset_feed_length(eps_rel, Len)
    print "Inset Feed Length Length=%f" % fl
```

```
    # The notch width formula doesn't seem to work. It barely has an effect on changing
    # the resonant frequency. So we choose a value of 3mm
    f_resonant = f0
    #fw = mspatchebg.get_feed_notch_width(eps_eff, f0, f_resonant)
    mspatchebg.fw = 0.0010
    print "Feed Notch Width=%f" % mspatchebg.fw

    ebg_N = mspatchebg.get_ebgN(Len, Leff, discontinuity)
    print "Number of ebg structures=%f" % ebg_N

    ebg_gap = mspatchebg.get_ebg_gap(Len, Leff, ebg_N)
    print "EBG Gap size=%f" % ebg_gap

    ebg_len = mspatchebg.get_ebg_len(Len, Leff, ebg_N)
    print "EBG-Length=%f" % ebg_len
    print "EBG-Width=EBG-Length=%f" % ebg_len

    board_len = mspatchebg.get_board_len(Leff, dL)
    print "Board-Length=%f" % board_len

    ebg_width = ebg_len
    board_width = mspatchebg.get_board_width(ebg_width, ebg_cols, Weff, discontinuity)
    print "Board-Width=%f" % board_width

    # Over-ride calculated variables if you have to, using the infile when producing feko or gerber file
    if infile:
        cfg = ConfigParser()
        cfg.read(infile)
        if cfg.get("config", "Patch-Width") != None and cfg.get("config", "Patch-Width") != "":
            mspatchebg.Width = float(cfg.get("config", "Patch-Width"))
            print "Config Patch-Width=%s" % cfg.get("config", "Patch-Width")
        if cfg.get("config", "Patch-Length") != None and cfg.get("config", "Patch-Length") != "":
            mspatchebg.Len = float(cfg.get("config", "Patch-Length"))
            print "Config Patch-Length=%s" % cfg.get("config", "Patch-Length")
        if cfg.get("config", "Feed-Inset-Length") != None and cfg.get("config", "Feed-Inset-Length") != "":
            mspatchebg.fl = float(cfg.get("config", "Feed-Inset-Length"))
            print "Feed-Inset-Length=%s" % cfg.get("config", "Feed-Inset-Length")
        if cfg.get("config", "Feed-Notch-Width") != None and cfg.get("config", "Feed-Notch-Width") != "":
            mspatchebg.fw = float(cfg.get("config", "Feed-Notch-Width"))
            print "Feed-Notch-Width=%s" % cfg.get("config", "Feed-Notch-Width")
        if cfg.get("config", "EBG-Length") != None and cfg.get("config", "EBG-Length") != "":
            mspatchebg.ebg_len = float(cfg.get("config", "EBG-Length"))
            print "EBG-Length=%s" % cfg.get("config", "EBG-Length")
        if cfg.get("config", "EBG-Width") != None and cfg.get("config", "EBG-Width") != "":
            mspatchebg.ebg_width = float(cfg.get("config", "EBG-Width"))
            print "EBG-Width=%s" % cfg.get("config", "EBG-Width")

        dL = mspatchebg.get_dL(h_sub, eps_eff, Width)
        Leff = mspatchebg.get_Leff(f0, eps_eff)
        board_len = mspatchebg.get_board_len(Leff, discontinuity)
        print " New Board-Length=%f" % board_len

        board_width = mspatchebg.get_board_width(ebg_width, ebg_cols, Weff, discontinuity)
        print "New Board-Width=%f" % board_width

    if outfiletype == "feko":
        write_feko(mspatchebg, outfile)
    else:
        write_gerber(mspatchebg, outfile)

    write_file_definition(mspatchebg, outfile)


# --- Main program
if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

Listing 13.5: The mpatchebgcalc.py Code File

13.4 The Edge Fed Microstrip Antenna Graph Plots

13.4.1 Patch 1: 0 EBGs

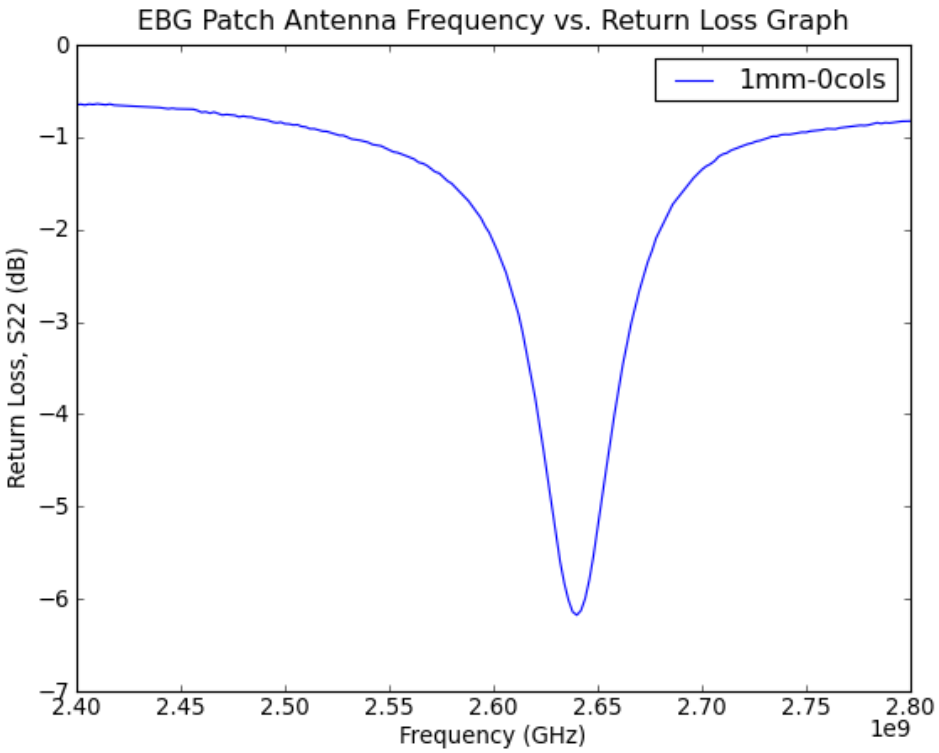


Figure 13.1: Return Loss Plot of patch with 0 EBGs

13.4.2 Patch 2: 2 EBG columns with 1mm spacing

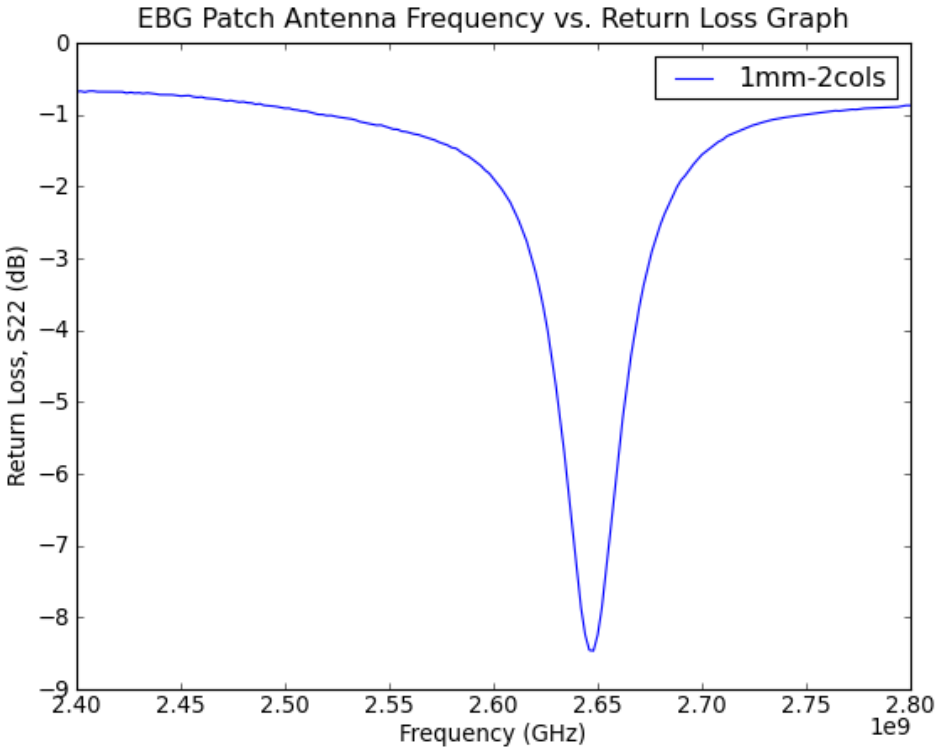


Figure 13.2: Return Loss Plot of patch having 2 EBG columns with 1mm spacing

13.4.3 Patch 3: 4 EBG columns with 1mm spacing

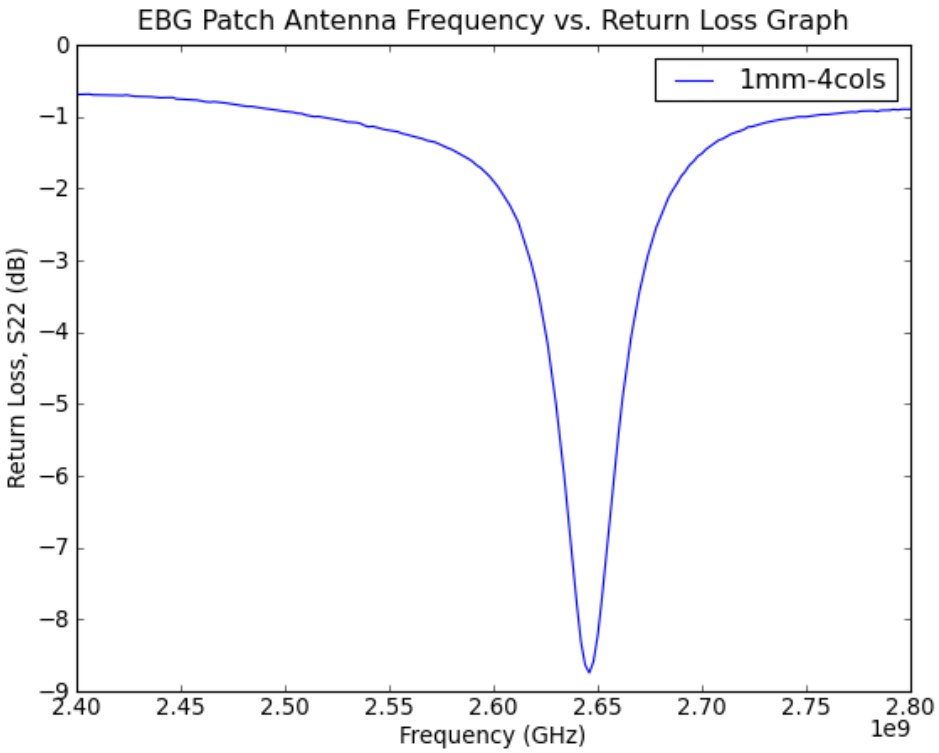


Figure 13.3: Return Loss Plot of patch having 4 EBG columns with 1mm spacing

13.4.4 Patch 4: 4 EBG columns with 0.5mm spacing

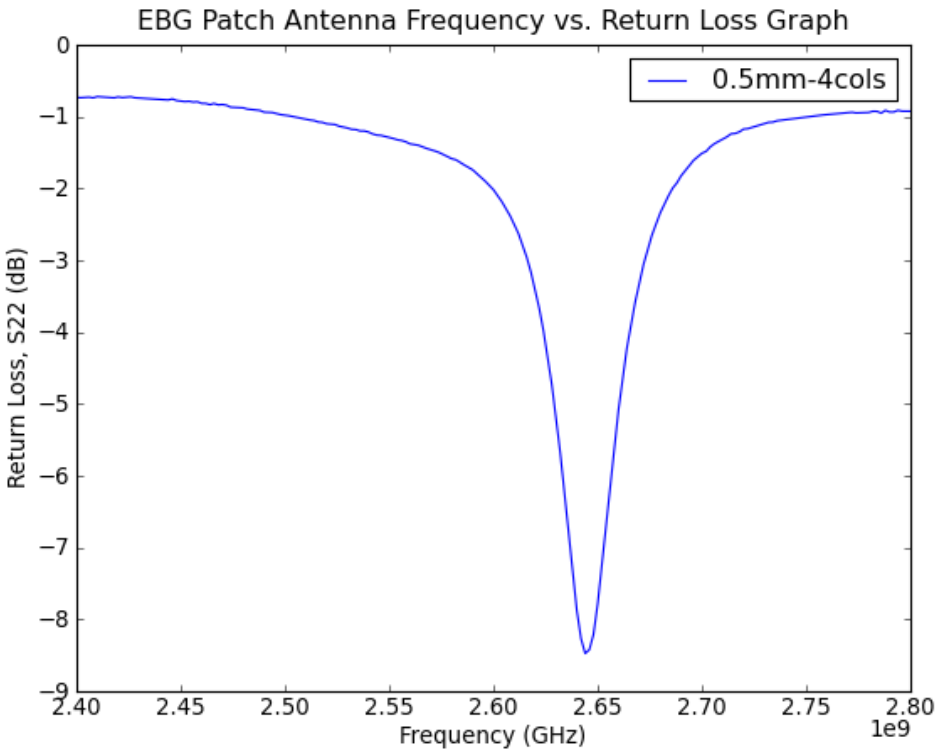


Figure 13.4: Return Loss Plot of patch having 4 EBG columns with 0.5mm spacing

13.4.5 Combined graph of all the Patch Antennas

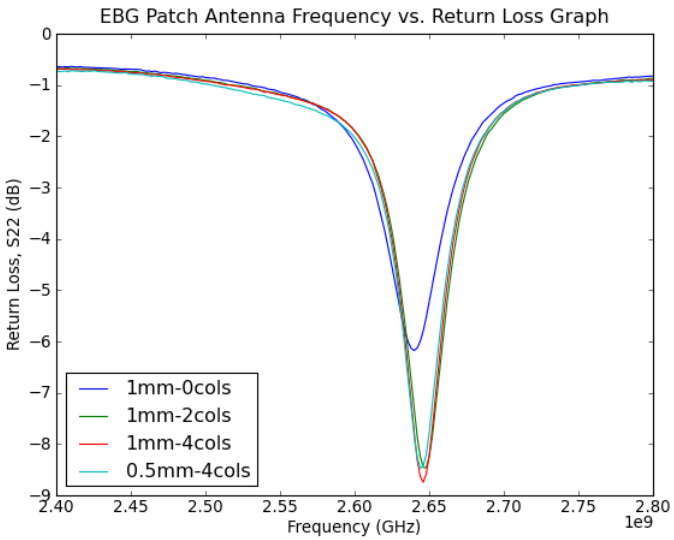


Figure 13.5: Return Loss Plots of all the Antennas

13.5 Rhino Expansion Board Schematics: Iteration 1

This appendix presents the first version of the Rhino expansion board schematics.

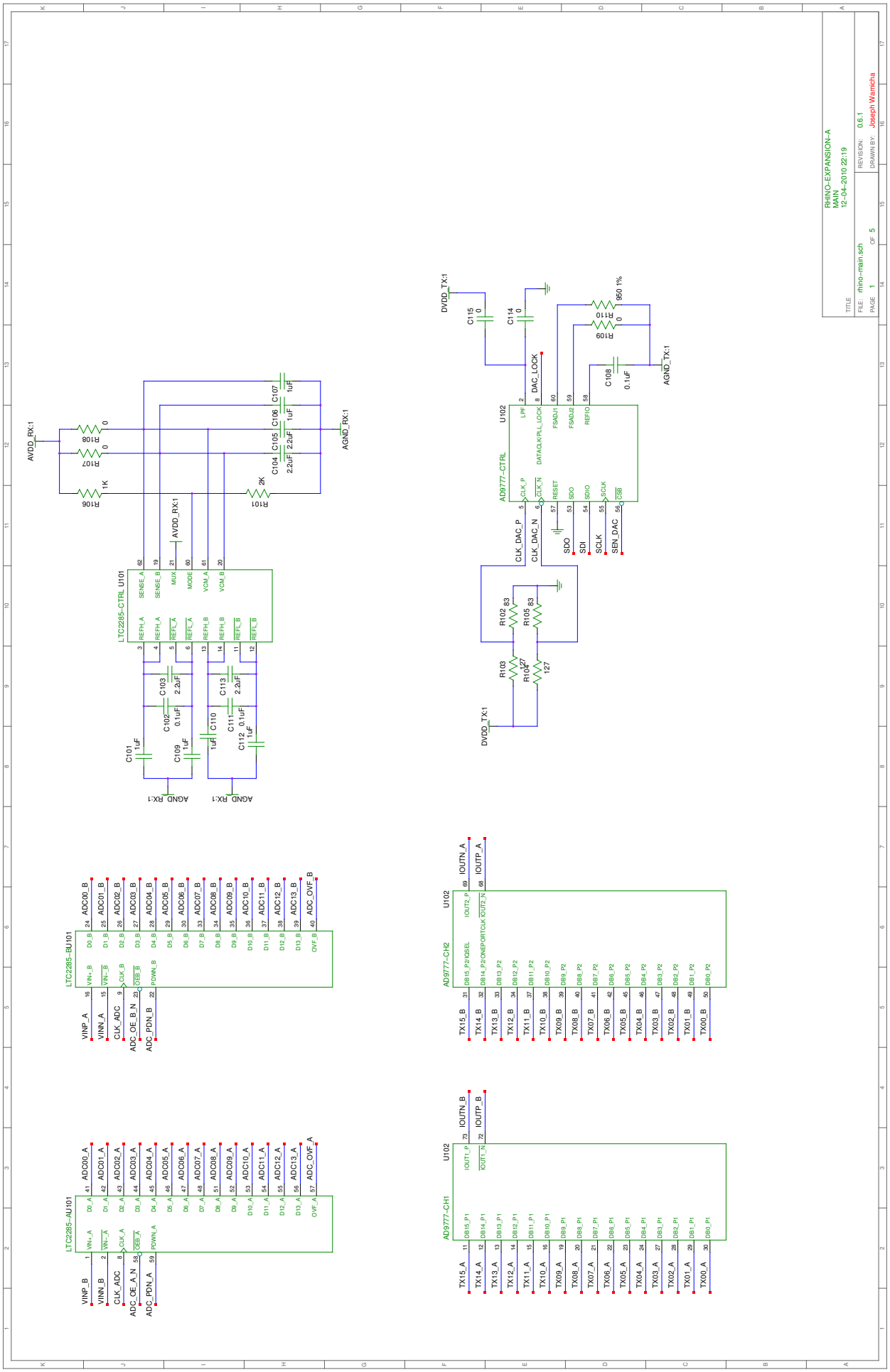


Figure 13.6: Rhino Expansion Board V1: Main

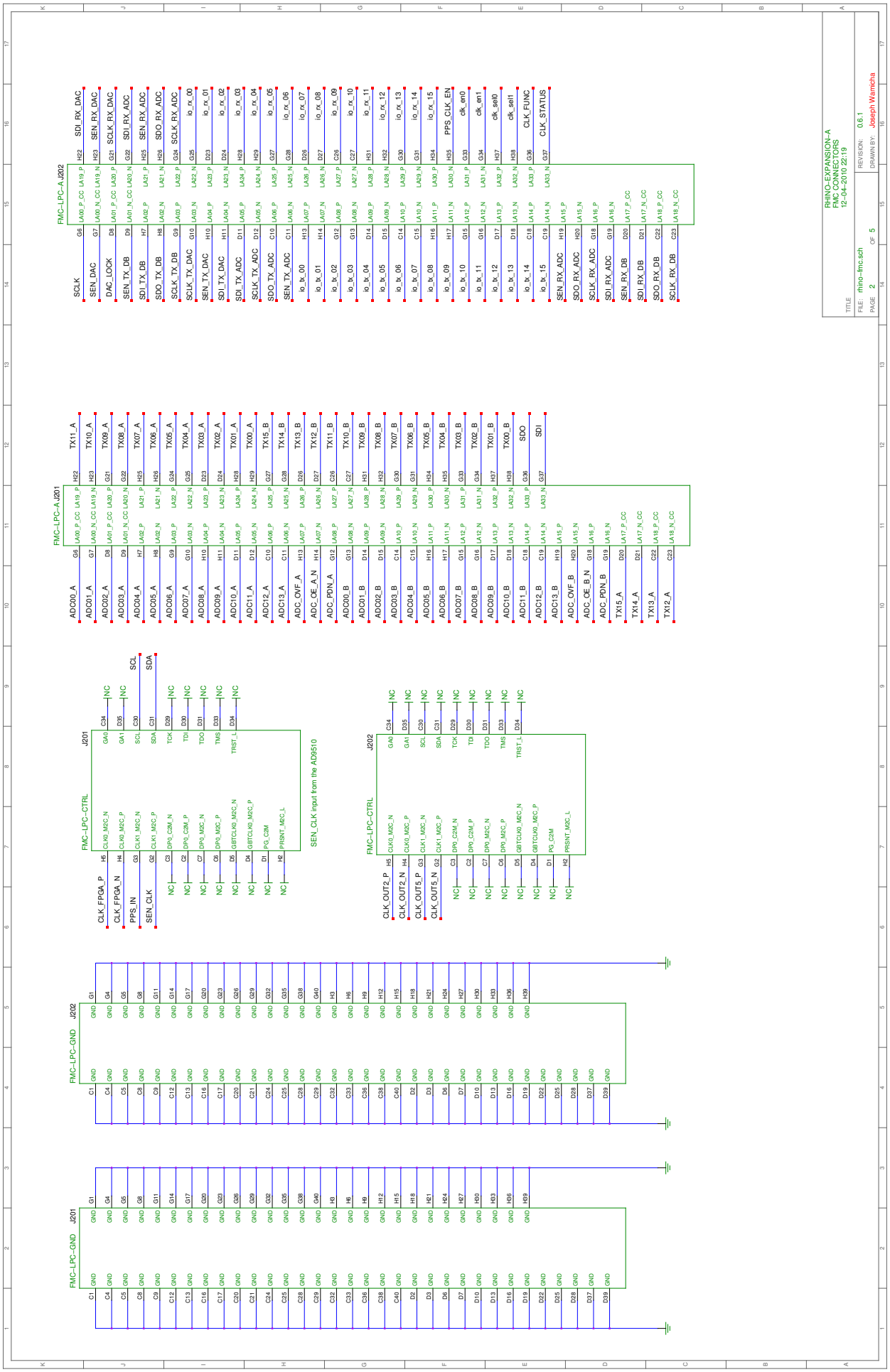


Figure 13.7: Rhino Expansion Board V1: FMC

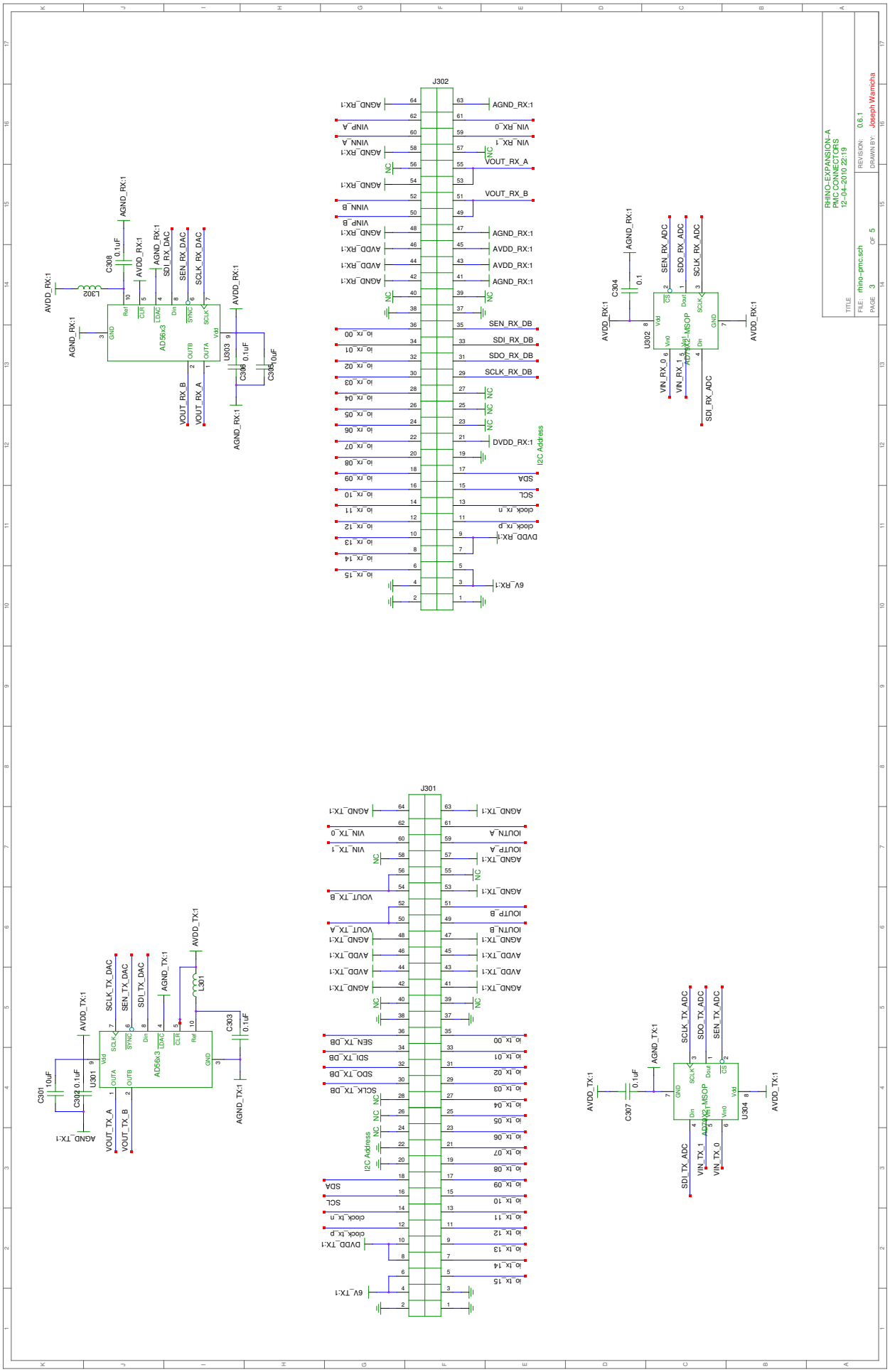


Figure 13.8: Rhino Expansion Board V1: PMC

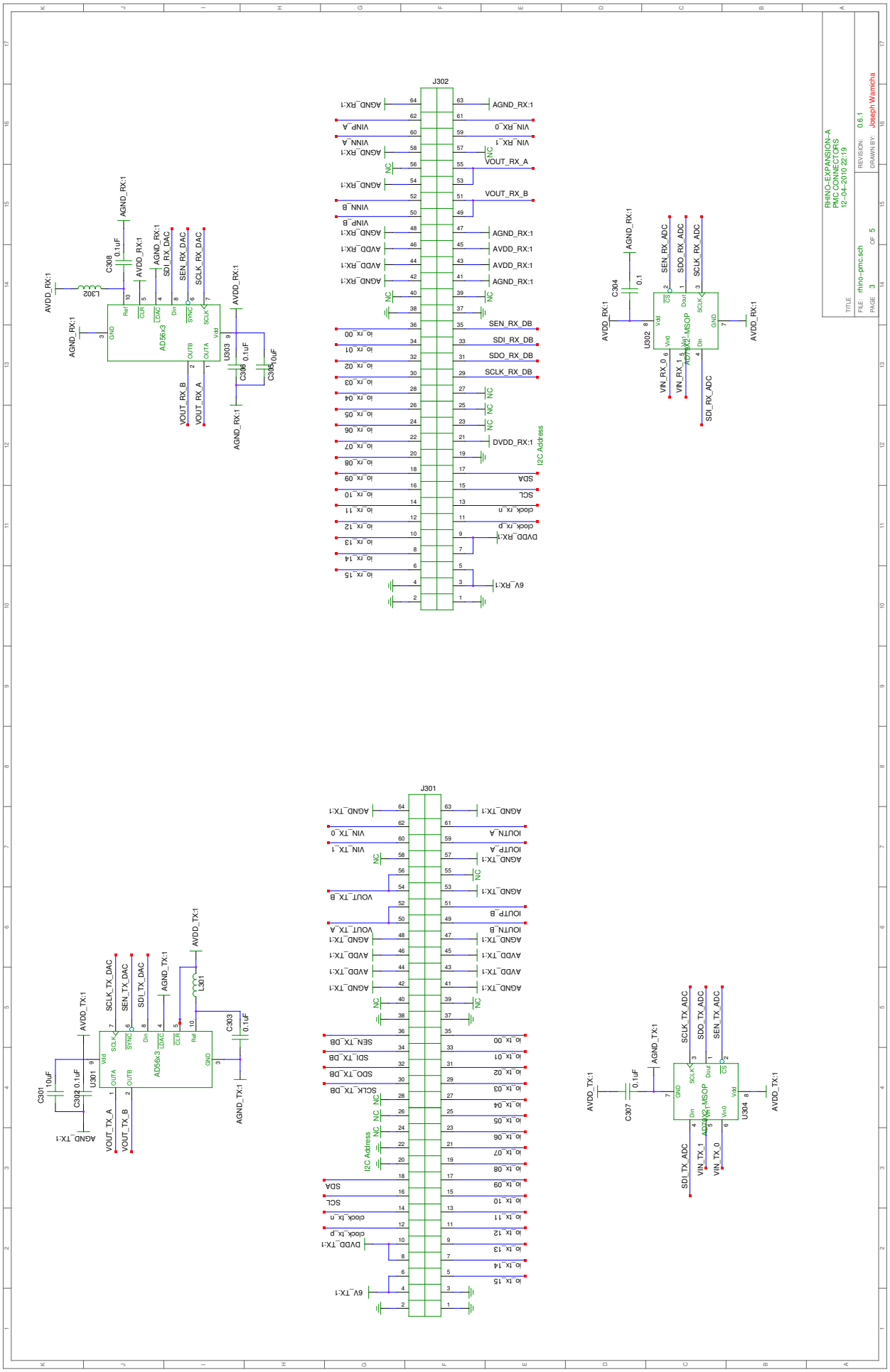


Figure 13.9: Rhino Expansion Board V1: Power

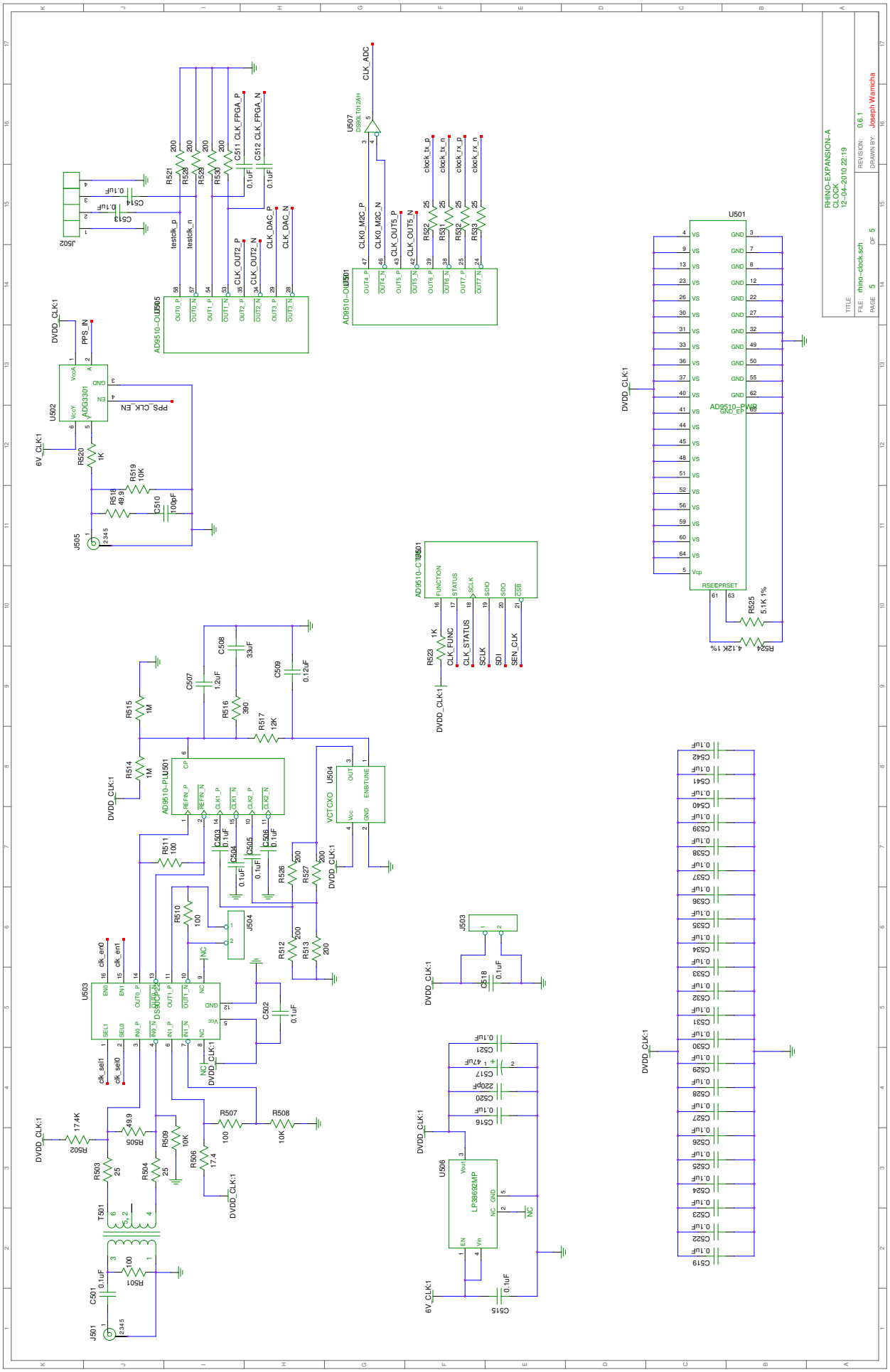


Figure 13.10: Rhino Expansion Board V1: Power

13.6 Rhino Expansion Board BOM: Iteration 1

This section presents the Bill of Materials for the first version of the Rhino expansion Board.

Table 13.1: Rhino Expansion Board BOM: Iteration 1

refdes	footprint	value	description	qty
U101	QFN64	unknown	Dual 14-bit 125 MSps ADC	1
R101	603	2K	unknown	1
U102	TQFP80	unknown	400 MS/s 16-bit DAC	1
C103,C104,C105,C113	1206	2.2uF	capacitor	4
R103,R104	603	127	resistor	2
R102,R105	603	83	resistor	2
R107,R108,R109	402	0	resistor	3
R110	603	950 1%	resistor	1
C114,C115	603	0	capacitor	2
J201	fmc	unknown	FMC Connector	2
J301,J302	pmc	unknown	PMC Connectors	2
C304	603	0.1	capacitor	1
U301,U303	MSOP10	unknown	AD5623/AD5643/AD5663 (+R) Dual 12/14/16-bit serial DAC	2
C301,C305	1206	10uF	capacitor	2
U302,U304	MSOP8	unknown	AD7912/7922 Analog Devices Dual 10/12-bit serial ADC	2
L301,L302,L401,L402,L403,L404,L405,L406,L407	1206	unknown	inductor	9
C101,C106,C107,C109,C110,C112,C407,C426,C449	1206	1uF	capacitor	9
R401,R402	603	6	resistor	2
U502	SC70-6	unknown	Level translator	1
T501	603	unknown	unknown	1
J502	SIP4	unknown	unknown	1
J503,J504	SIP2N	unknown	generic connector	2
R502	1206	17.4K	unknown	1
U503	TSSOP	unknown	2x2 LVDS Crosspoint Switch, in SO16 or TSSOP16	1
R506	603	17.4	resistor	1
R501,R507,R510,R511	603	100	resistor	4
U504	vctcxo	unknown	VCTCXO	1
R514,R515	603	1M	resistor	2
C507	1206	1.2uF	capacitor	1
C508	1206	33uF	capacitor	1
R516	603	390	resistor	1
R517	603	12K	resistor	1
C509	603	0.12uF	capacitor	1
J501,J505	SMA_VERT	unknown	unknown	2
R505,R518	603	49.9	resistor	2
R508,R509,R519	603	10K	resistor	3
C510	603	100pF	capacitor	1
U501,U505	LFCSP64	unknown	Clock Divider and PLL	2
U506	SOT223-5	unknown	National Fixed LDO, 1A	1
C406,C408,C413,C418,C423,C427,C450,C517	1206	47uF	polarized capacitor	8
U507	SOT23-5	unknown	LVDS Receiver	1
R106,R520,R523	603	1K	resistor	3
R524	603	4.12K 1%	resistor	1
R525	603	5.1K 1%	resistor	1
R512,R513,R521,R526,R527,R528,R529,R530	603	200	resistor	8
R503,R504,R522,R531,R532,R533	603	25	resistor	6
C403,C405,C410,C412...C541,C542	603	0.1uF	capacitor	76

13.7 Rhino Expansion Board Schematics: Iteration 2

This appendix presents the second version of the Rhino expansion board schematics.

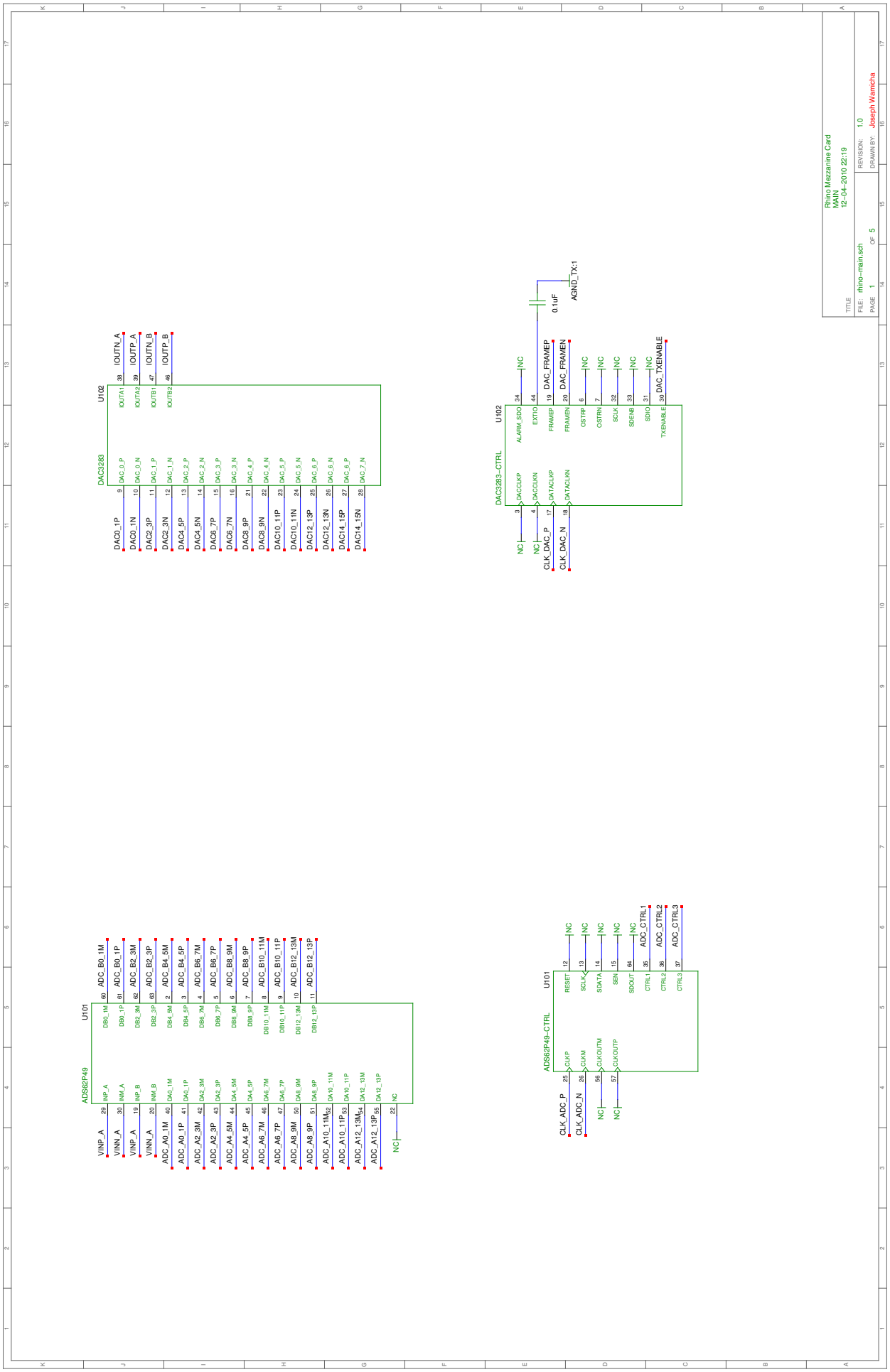


Figure 13.11: Rhino Expansion Board V2: Main

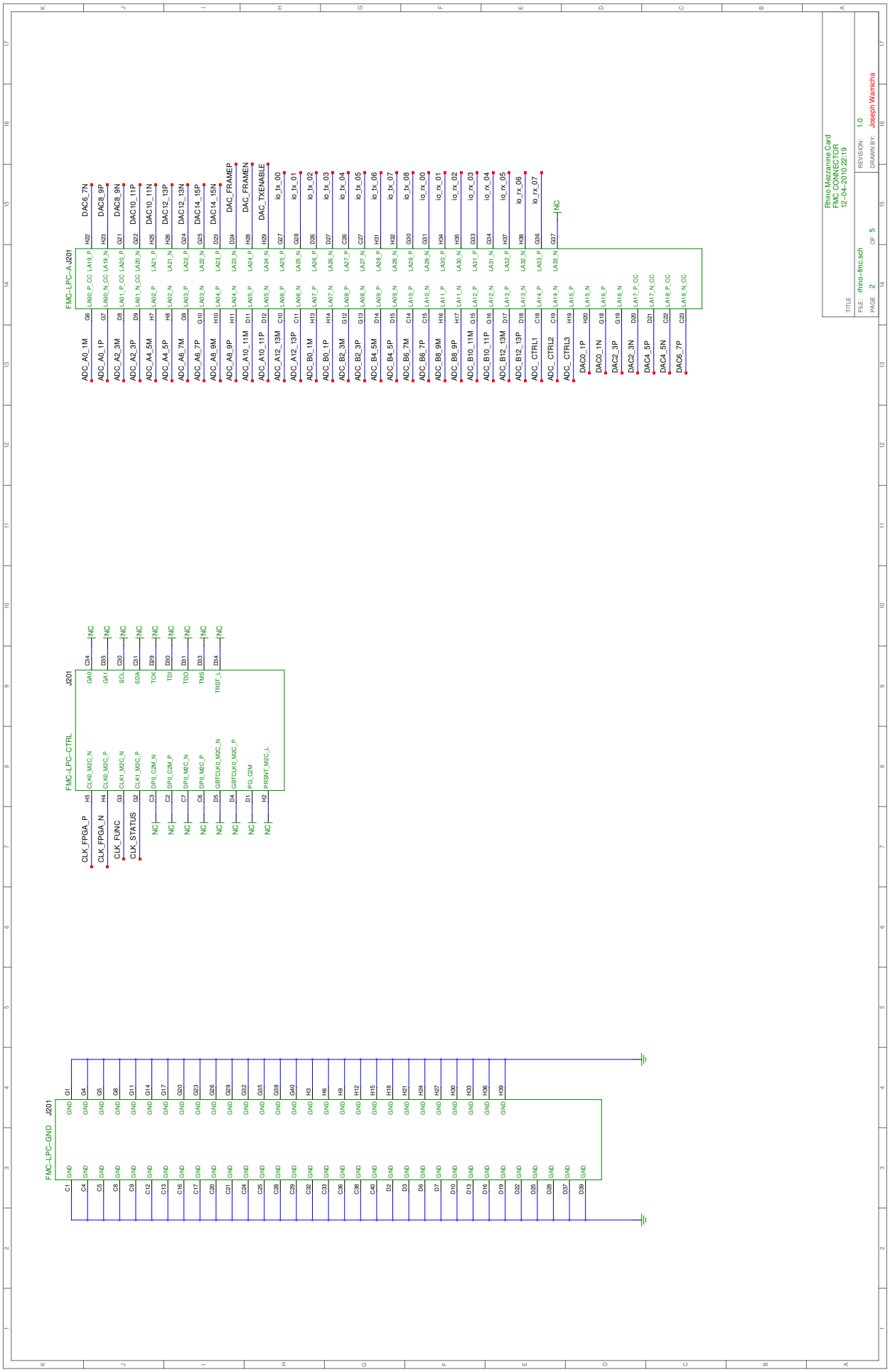


Figure 13.12: Rhino Expansion Board V2: FMC

Figure 13.13: Rhino Expansion Board V2: PMC

Figure 13.14: Rhino Expansion Board V2: Power

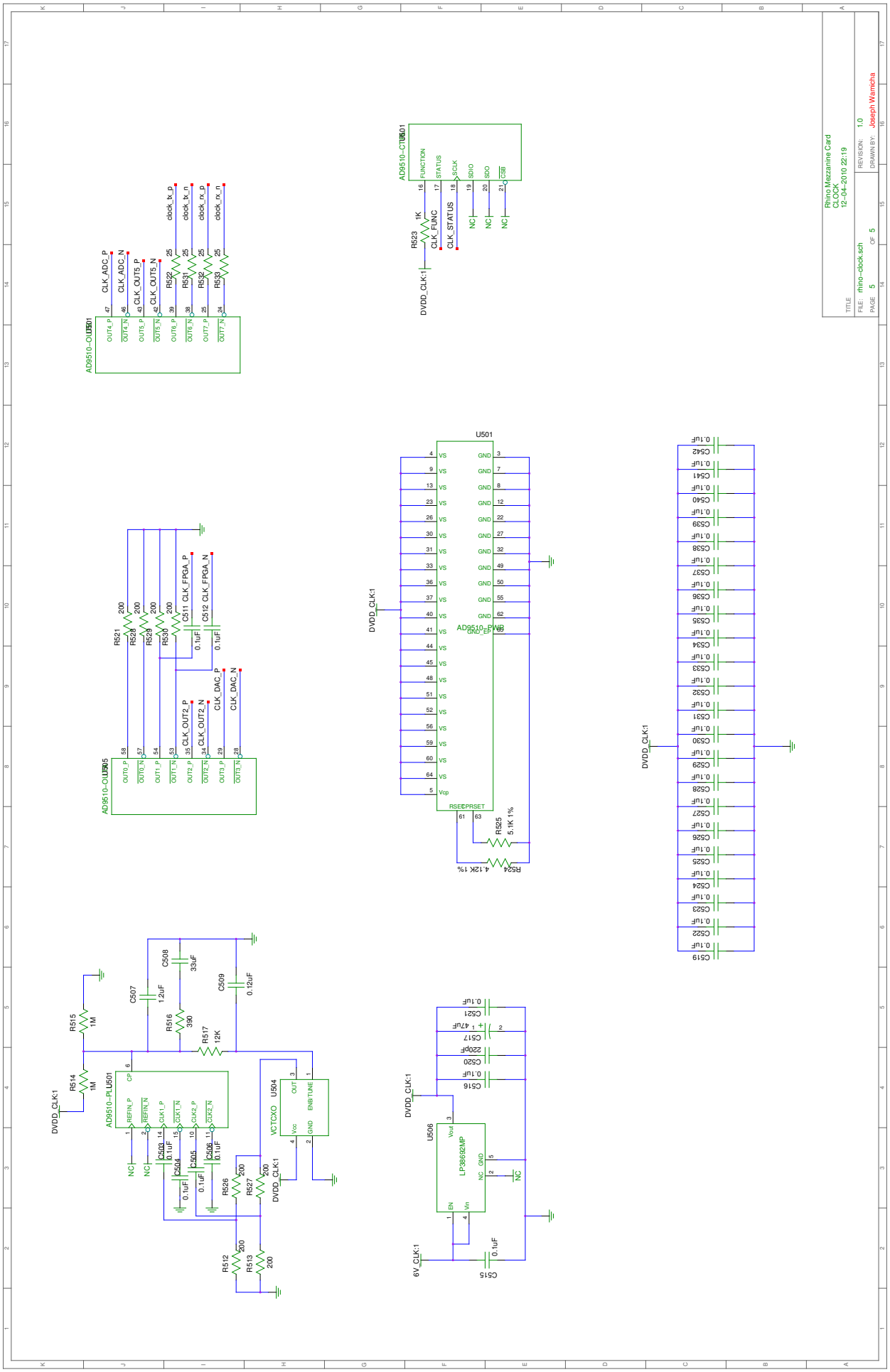


Figure 13.15: Rhino Expansion Board V2: Power

13.8 Rhino Expansion Board BOM: Iteration 2

Table 13.2: Rhino Expansion Board BOM: Iteration 2

R516	603	390	resistor	1
R517	603	12K	resistor	1
C509	603	0.12uF	capacitor	1
U501,U505	LFCSP64	unknown	Clock Divider and PLL	2
U506	SOT223-5	unknown	National Fixed LDO, 1A	1
C406,C408,C413,C418,C423,C427,C450,C517	1206	47uF	polarized capacitor	8
R523	603	1K	resistor	1
R524	603	4.12K 1%	resistor	1
R525	603	5.1K 1%	resistor	1
R512,R513,R521,R526,R527,R528,R529,R530	603	200	resistor	8
R522,R531,R532,R533	603	25	resistor	4
C403,C405,C410...C539,C540,C541,C542	603	0.1uF	capacitor	45

Bibliography

[1] Sainati Robert A. *CAD of Microstrip Antennas for Wireless Applications*. Artech House, first edition, 1996.

[2] Carruthers Jefferey B. Wireless infrared communications. *Wiley Encyclopedia of Telecommunications*, 2002.

[3] Davidson David B. *Computational Electromagnetics for RF and Microwave Engineering*. Cambridge University Press, first edition, 2005.

[4] Saltzberg Burton R. Bahai Ahmad R. S. *Multi-Carrier Digital Communications: Theory and Applications of OFDM*. John Wiley & Sons, 2002.

[5] Ammann M. J. Bao X. L., Ruvio G. and John M. A novel gps patch antenna on a fractal hi-impedance surface substrate. *IEEE Antennas AND Wireless Propagation Letters,*, 5(1), December 2006.

[6] Goldberg Bar-Giora. *Digital Frequency Synthesis Demystified: DDS and Fractional-N PLLs*. LLH Technology Publishing, 1999.

[7] Cherubini Giovanni Benvenuto Nevio. *Algorithms for Communication Systems and their Applications*. John Wiley & Sons, 2002.

[8] Bhushan Bharat, editor. *Handbook of Nanotechnology*. Springer, 2004.

[9] Yang Samuel C. *CDMA RF System Engineering*. Artech House, 1998.

[10] Godara Lal Chand, editor. *Compact and Broadband Microstrip Antenna*. CRC Press, 2002.

[11] Chiung-Jang Chen. Multiuser diversity for antenna optimal combining in interference-limited systems. *IEEE Communications Letters*, 12(5), May 2008.

[12] ETSI DVB-T2 Committee. *Frame structure channel coding and modulation for a second generation digital terrestrial television broadcasting system (DVB-T2)*. ETSI, May 2008.

[13] IEEE LAN/MAN Committee. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. IEEE, 2007.

[14] IEEE WAN Committee. *Air Interface for Broadband Wireless Access Systems*. IEEE, 2009.

[15] Gibson Jerry D. *The Communications Handbook*. John Wiley & Sons, second edition, 2002.

[16] Parsons J. D. *The Mobile Radio Propagation Channel*. John Wiley & Sons, second edition, 2002.

[17] E. da Silva. *High Frequency Microwave Engineering*. Butterworth Heinemann, 2001.

[18] Skold Johan Beming Per Dahlman Erik, Parkvall Stefan. *3G Evolution: HSPA and LTE for Mobile Broadband*. Elsevier Ltd., second edition, 2008.

[19] Minoli Daniel. *Nanotechnology Applications to Telecommunications and Networking*. John Wiley & Sons, 2006.

[20] Roddy Dennis. *Satellite Communications*. McGraw Hill, third edition, 2001.

[21] Laurent Doldi. *Validation of Communications Systems with SDL: The art of SDL Simulation and Reachability Analysis*. John Wiley & Sons, 2003.

[22] Torrieri Don. *Principles of Spread-Spectrum Communication Systems*. Springer, 2005.

[23] Nakagawa Masao Esmailzadeh Riaz. *TDD-CDMA for Wireless Communications*. Artech House, 2003.

[24] Ettus Mark et al. Usrp2 schematics. Documentation from the gnuradio project, 2009.

[25] Ettus Mark et al. Usrp2 general faq. Documentation from the gnuradio project, 2010.

[26] Ettus Mark et al. Usrp2 ofdm signal processing blocks. Documentation from the gnuradio project, 2010.

[27] Ettus Mark et al. Usrp2 rf daughter boards. Documentation from the gnuradio project, 2010.

[28] Kaiser S. Fazel K. *Multi-Carrier and Spread Spectrum Systems: From OFDM and MC-CDMA to LTE and WiMAX*. John Wiley & Sons, second edition, 2008.

[29] Enrico Forestieri. *Optical Communication Theory and Techniques*. Springer, 2005.

[30] Xiong Fuqin. *Digital Modulation Techniques*. Artech House, 2000.

[31] Auslander Edgar Gatherer Alan. *The Application of Programmable DSPs in Mobile Communications*. John Wiley & Sons, 2002.

[32] Tsoulos George. *MIMO System Technology for Wireless Communications*. CRC Press, 2006.

[33] Sidiropoulos N.D. Gershman A.B. *Space Time Processing for MIMO Communications*. John Wiley & Sons, 2005.

[34] Neshati M.H. Ghassemi N., Mohassel Rashed J. Microstrip antenna design for ultra wideband application by using two slots. China, March 2008. Progress In Electromagnetics Research.

[35] Kohno R. Ghavami M., Michael L.B. *Ultra Wideband Signals and Systems in Communication Engineering*. John Wiley & Sons, 2004.

[36] Larson Lawrence E. Groe John B. *CDMA Mobile Radio Design*. Artech House, 2000.

[37] Schantz Gregory Hans. *Nano Antenna Apparatus and Method*, June 2005.

[38] Bullinger Hans-Jorg, editor. *Technology Guide: Principles Applications Trends*. Springer, 2009.

[39] Keller T. Hanzo L., Webb W.T. *Single and Multi Carrier Quadrature Amplitude Modulation: Principles and Applications for Personal Communications, WLANs and Broadcasting*. John Wiley & Sons, second edition, 2000.

[40] Yee M. S. Hanzo L., Wong C.H. *Adaptive Wireless Transceivers: Turbo-Coded, Turbo-Equalized and Space-Time Coded TDMA, CDMA, and OFDM Systems*. John Wiley & Sons, 2002.

[41] Blogh J.S. Hanzo L.S. *Third Generation Systems and Intelligent Wireless Networking: Smart Antennas and Adaptive Modulation*. IEEE Press, John Wiley & Sons, 2002.

[42] Prasad Ramjee Harada Hiroshi. *Simulation and Software Radio for Mobile Communications*. Artech House, 2002.

[43] Anderson Harry. *Fixed Broadband Wireless System Design*. John Wiley & Sons, 2003.

[44] Sagkob Holger Heine Gunnar. *GPRS Gateway to Third Generation Mobile Networks*. Artech House, 2003.

[45] Raida Zbynek Horak Jiri. Influence of ebg structures on the far-field pattern of patch antennas. *Radio Engineering*, 18(2), June 2009.

[46] Hammuda Husni. *Cellular Mobile Radio Systems: Designing Systems for Capacity Optimization*. John Wiley & Sons, 1998.

[47] Mitola Joseph III. *Software Radio Architecture: Object Oriented Approaches to Wireless Systems Engineering*. John Wiley & Sons, 2000.

[48] Bates Regis J. *Broadband Telecommunications Handbook*. McGraw Hill, second edition, 2002.

[49] Carr Joseph J. *Antenna ToolKit*. Newnes, 2001.

[50] Zijlstra Tony Kroug Mathias Kooi Jacob W. Stern Jeffrey A. Klpawijk Teun M. Jackson Brian D., Gert de Lange. Low-noise 0.8-0.96 and 0.96-1.12 thz superconductor-insulator-superconductor mixers for the herschel space observatory. *IEEE Transactions on Microwave Theory and Techniques*, 54(2), February 2006.

[51] Parsons J.D. *Mobile Radio Propagation Channel*. John Wiley & Sons, second edition, 2000.

[52] Shanmugan Sam K. Jeruchim Michael C., Balaban Philip. *Simulation of Communication Systems: Modeling, Methodology and Techniques*. Kluwer Academic Publishers, second edition, 2000.

[53] Wong Kai-Kit Gao Xiqi Jin Shi, McKay Mathew R. Transmit beamforming in rayleigh product mimo channels: Capacity and performance analysis. *IEEE Transactions on Signal Processing*, 56(10), October 2008.

[54] Winn Joshua N. Meade Robert D. Joannopoulos John D., Johnson Steven G. *Photonic Crystals: Moulding the flow of light*. Princeton University Press, second edition, 2008.

[55] Mitola Joe. The software radio architecture. *IEEE Communications Magazine*, 33(5), 1995.

[56] Proakis John, editor. *Digital Communications*. McGraw Hill, 2007.

[57] Sethares William A. Johnson Richard C. Jr. *Telecommunication Breakdown: Concepts of Communication Transmitted via Software Defined Radio*. Prentice Hall, 2003.

[58] Smith Jr. *Apparatus and Methods for Tuning Antenna Impedance using Transmitter and Receiver Parameters*, January 2006.

[59] Drexler Eric K. *Engines of Creation 2.0: The Coming Era of Nanotechnology*. www.wowio.com, 2007.

[60] Simon Marvin K. *Bandwidth-Efficient Digital Modulation wiht Application to Deep-Space Communications*. California Institute of Technology, 2001.

[61] Tsachtsiris Goeroge F. Soras Constantine F. Makios Vasileios T. Karaboikis Manos P., Papamichael Vasileios. Integrating compact printed antennas onto small diversity/mimo terminals. *IEEE Transactions on Antennas and Propagation*, 56(7), July 2008.

[62] Tsachtsiris Goeroge F. Soras Constantine F. Makios Vasileios T. Karaboikis Manos P., Papamichael Vasileios. Integrating compact printed antennas onto small diversity/mimo terminals. *IEEE Transactions on Antennas and Propagation*, 56(7), July 2008.

[63] Sarraf Mohsen Karim M.R. *W-CDMA and CDMA2000 for 3G Mobile Networks*. McGraw Hill, 2002.

[64] Eskelinen Pekka Kihara Masami, Ono Sadayasu. *Digital Clocks for Synchronization and Communications*. Artech House, 2003.

[65] Koo Insoo Kim Kiseon. *CDMA Systems Capacity Engineering*. Artech House, 2005.

[66] Engin Ege Yun Wansuk Toyota Yoshitaka Swaminathan Madhavan Kim Tae Hong, Chung Daehyun. A novel synthesis method for designing electromagnetic band gap (ebg) structures in packaged mixed signal systems. In *Electronic Components Technology Conference*. IEEE, 2006.

[67] Wong Kin-Lu. *Compact and Broadband Microstrip Antenna*. John Wiley & Sons, 2002.

[68] Girish Kumar and K.P. Ray. *Broadband Microstrip Antenna*. Artech House, 2003.

[69] Freeman Roger L. *Fundamentals of Telecommunications*. John Wiley & Sons, second edition, 200.

[70] Calhoun George M. *Third Generation Wireless Systems*. Artech House, 2003.

[71] Leung Martin. *Microstrip Antenna Design Using Mstrip40*. University of Canberra, first edition, November 2002.

[72] Patzold Mathias. *Mobile Fading Channels*. John Wiley & Sons, 2002.

[73] Engheta Nader Mc Vay John, Hoorfar Ahmad. Radiation characteristics of microstrip dipole antennas over a high-impedance metamaterial surface made of hilbert inclusions. *University of Pennsylvania, Department of Electrical and Systems Engineering*, 2003.

[74] Ibnkahla Mohamed. *Signal Processing for Wireless Communications Handbook*. CRC Press, 2004.

[75] Hyung G. Myung. *Technical Overview of 3GPP LTE*. IEEE.org, 2008.

[76] Melodia Tommaso et al. Nagaraju Pradeep B., Ding Lei. Dynamic spectrum allocation algorithm on usrp2 radios. In *IEEE SECON 2007 Conference Proceedings*. Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference, June 2010.

[77] Milstein Laurence B. Nagvanshi Preeti, Masry Elias. Optimum trasmit-receive beamforming with noisy channel estimates for correlated mimo rayleigh channels. *IEEE Transactions on Communications*, 56(11), November 2008.

[78] Wu Z. Wiegandt D. Zekavat S.A. Nassar Carl R., Natarajan B. *Multi-Carrier Technologies for Wireless Communication*. Kluwer Academic Publishers, 2002.

[79] Rodgers James Neil. *RFID SILICON ANTENNA*, March 2007.

[80] Kolawole Michael O. *Satellite Communications Engineering*. Marcel Decker, 2002.

[81] Agrawal Govind P. *Fiber-Optic Communication Systems*. John Wiley & Sons, 2002.

[82] Brenner Pablo. A technical tutorial on the 802.11 protocol. July 1997.

[83] Warner Steven B. Kasilingam Dayalan Ajayan Pulickel M. Notaros Branislav Razdan Sandeep Simmons Trevor J. Laddha Arpit Yilmaz Erdem Patra Prabir K., Calvert Paul D. *Textile Based Carbon Nanostructured Flexible Antenna*, June 2007.

[84] Burns Paul. *Software Defined Radio for 3G*. Artech House, 2003.

[85] Stavroulakis Peter. *Interference Analysis and Reduction for Wireless Systems*. Artech House, 2003.

[86] Elbert Bruce R. *The Satellite Communication Applications Handbook*. Artech House, 2004.

[87] Stephens Donald R. *Phase-Locked Loops for Wireless Communications: Digital, Analog and Optical Implementations*. Kluwer Academic Publishers, second edition, 2002.

[88] KB Yip Ramesh M. Design formula for inset fed microstrip patch antenna. *Journal of Microwaves and Optoelectronics*, 3(3), 2003.

[89] Lee Jhong S. *Miller Leonard E.* Artech House, 1998.

[90] Glisic Savo. *Adaptive WCDMA: Theory and Practice*. John Wiley & Sons, 2003.

[91] Luders Christian Schulze Henrik. *Theory and Applications of OFDM and CDMA: Wideband Wireless Communications*. John Wiley & Sons, 2005.

[92] Baker Mathew Sesia Stefania, Toufik Issam. *LTE: The UMTS Long Term Evolution from Theory to Practice*. John Wiley & Sons, 2009.

[93] Zigangirov Kamil Sh. *Theory of Code Division Multiple Access Communication*. IEEE Press, John Wiley & Sons, 2004.

[94] Hu Fun Y. Sheriff Ray E. *Mobile Satellite Communication Networks*. John Wiley & Sons, 2001.

[95] Haykin Simon. *Communication Systems*. John Wiley & Sons, fourth edition, 2001.

[96] Alouini Mohamed-Slim Simon Marvin K. *Digital Communications over Fading Channels: A Unified Approach to Performance Analysis*. John Wiley & Sons, 2000.

[97] Bregni Stefano. *Synchronization of Digital Telecommunications Networks*. John Wiley & Sons, 2002.

[98] Ishizaki Suguru. *Improvisational Design: Continuous, Responsive, Digital Communication*. MIT Press, 2003.

[99] Ha Tri T. *Digital Satellite Communications*. McGraw Hill, second edition, 1990.

[100] Vieracker Tobias. *Planning DVB-T2: Advance and Challenge*. LS Telecom, 2010.

[101] Omura Jim K. Viterbi Andrew. *Principles of Digital Communication and Coding*. McGraw Hill, 1979.

[102] Collins Gerald W. *Fundamentals of Digital Television Transmission*. John Wiley & Sons, 2001.

[103] Smith Steven W. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, second edition, 1999.

[104] Tuttlebee Walter. *Software Defined Radio: Enabling Technologies*. John Wiley & Sons, 2002.

[105] Tuttlebee Walter. *Software Defined Radio: Origins, Drivers and International Perspectives*. John Wiley & Sons, 2002.

[106] Jiangzhou Wang. *High-Speed Wireless Communications, Ultra-wideband, 3G Long-Term Evolution and 4G Mobile Systems*. Cambridge University Press, 2008.

[107] Webb William. *Understanding Cellular Radio*. Artech House, 1998.

[108] Webb William. *The Complete Wireless Communications Professional: A Guide for Engineers and Managers*. Artech House, 1999.

[109] Poor Vincent H. Wornell Gregory W., editor. *Wireless Communications: Signal Processing Perspectives*. Prentice Hall, 1998.

[110] Blum Rick S. Xu Zhemin, Sfar Sana. Analysis of mimo systems with receive antenna selection in spatially correlated rayleigh fading channels. *IEEE Transactions on Vehicular Technology*, 58(1), January 2009.

[111] Guo Jay Y. *Advances in Mobile Radio Access Networks*. Artech House, 2004.

[112] Rahmat-Samii Yahya Yang Fan. *Electromagnetic Band Gap Structures in Antenna Engineering*. Cambridge University Press, first edition, 2009.

[113] Ma Yiran Yang Qi, Shieh William. Guard-band influence on orthogonal band multiplexed coherent optical ofdm. *Wiley Encyclopedia of Telecommunications*, 33(19), October 2008.

[114] Sun Yichuang. *Wireless Communication Circuits and Systems*. IEEE Press London, 2004.