2012

Implementation of a Software Defined Spread Spectrum Communication System

Mir Ali

Implementation of a Software Defined Spread Spectrum Communication System

A Thesis
Presented for the
Master of Science
Degree
in Engineering Science
The University of Mississippi

Mir Murtuza Ali

Dec 6, 2012

Abstract

The goal of this thesis is to develop a framework to prototype a software defined direct sequence spread spectrum transceiver that can be used as a node in an ad hoc network. We introduce the concept of a software radio, the current state of art, and GNU Radio and its concepts. We discuss in detail the design and development methods of GNU Radio and develop a flowgraph in Python to demonstrate the method of development. We present a mathematical analysis of direct-sequence spread-spectrum (DSSS) modulation and demodulation schemes along with the transmitter and receiver design. We use this design to develop an analogous design in GNU Radio using the signal processing blocks that are present in GNU Radio and ones we develop. We perform simulations and tests to validate the algorithms, signal processing blocks and flowgraphs that we developed. We find that the signal acquistion algorithm is capable of determining the code and frequency offset in a received DSSS signal. We also find that the carrier tracking loop is capable of tracking the received carrier when the signal has a high signal to noise ratio (SNR).

We conclude that GNU Radio as a technology can be used to prototype transceivers that are highly configurable and expandable. Finally, we identify and suggest some possible areas where this design can be developed and improved further.

List of Abbreviations

ADC analog-to-digital converter

AGC automatic gain control

AWGN additive white Gaussian noise

BPSK binary phase shift keying

CDMA code division multiple access

CSMA carrier-sense multiple access

CWC Center for Wireless Communications

DAC digital-to-analog converter

DBPSK differential phase-shift keyed

DCO digitally controlled oscillator

DDC digital-down-converter

DPLL digital phase-locked loop

DS-BPSK direct-sequence spread binary-phase-shift-keyed

DSSS direct-sequence spread-spectrum

DUC digital-up-converter

FFT fast Fourier transform

FSF Free Software Foundation

FPGA field programmable gate array

IF intermediate frequency

MAC medium access

MANET mobile ad hoc network

NCO numerically controlled oscillator

PGA programmable gain amplifier

PHY physical layer

PI proportional-integral

PLL phase-locked loop

RF radio frequency

SDR software-defined radio

SNR signal to noise ratio

SoC System on Chip

sps samples per second

USB Universal Serial Bus

USRP Universal Software Radio Peripheral

Acknowledgements

This thesis represents years of work with the Center for Wireless Communications at University of Mississippi, and because of that, there are a large number of people to thank. First, thanks to Drs. John N. Daigle, Lei Cao and Allison W. Glisson, my advisors and committee members. Each have contributed in significant ways to my education, either through my personal relationships and work experiences with them, or in classroom.

I want to extend a very special thanks to Dr. John N. Daigle under whom I studied over the past six years. He has always been a constant source of encouragement, advice and insipiration to me. I feel fortunate to be his student and I look forward to continuing our relationship in the future.

I would particularly like to thank Dr. Xiao Di and Tao Shi with whom I did some of my best work as a graduate research assistant at NCPA. My work with them has contributed to my education and career in many ways.

Finally, I cannot forget the influence of my family, friends and all the students of EE I worked with over the years. My parents and siblings have been significant role models in my life and their unwavering support and encouragment has always been of tremendous help. I am also grateful to my best friends Sadiq, Nishchal, Himanshu, Pradeep and Swasti for their help and support. I am also thankful to Eli who was always around to support and bring a smile to me.

University, Mississippi — Mir Murtuza Ali
December 2012

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The objective of this thesis is to develop a software radio framework to demonstrate the feasability of using GNU Radio to prototype a mutli-code DSSS transceiver using Universal Software Radio Peripheral (USRP) that can be used as an ad hoc networking node, capable of communicating with similar nodes in a mobile ad hoc network.

In this thesis, we start by introducing the concepts of GNU Radio and USRP. We then discuss the propsed design for the GNU Radio based DSSS transceiver. We then discuss the implementation of the proposed along with tests and validation of these algorithms.

1.1 Motivation for the Thesis

An ad hoc network is a collection of communication and processing stations that spontaneously organize themselves into a network and cooperate to facilitate execution of distributed applications. In a mobile ad hoc network (MANET), the nodes are free to move at will and without prior notification. This class of networks has potential to have enormous impact in a number of emergency domains where an extensive communications infrastructure is not likely to exist, such as tactical military operations, disaster recovery, law enforcement and fire fighting.

The Center for Wireless Communications at University of Mississippi, under the leadership of Dr. John N. Daigle, has continuously been involved in research in the area of ad

hoc networks during the past one decade. The focus of the present research at Center for Wireless Communications (CWC) is on developing protocols to manage single radio systems whose wireless resources are shared among the various functions using code division multiple access (CDMA).

The specific objectives of this research is described in detail in [4]. One particular objective of this research is to prototype a multi-code CDMA transceiver that can be used to perfrom tests that quantify performance of the protocols that were developed at CWC. In order to develop such a protoype, we decided to use GNU Radio and Universal Software Radio Peripheral (USRP) to develop a framework that implements multi-code CDMA. GNU Radio is a software library that provides a framework to develop software radios that are highly configurable, expandable and scalable. USRP provides the hardware frontend that works in unison with GNU Radio and together they provide the infrastructure to prototype a multi-code CDMA transceiver.

1.2 Introduction to SDR

A software-defined radio (SDR) is a radio communication system in which, the components that are typically implemented in hardware in a traditional modern digital radio communication system as represented in Figure 1.1 [18], are instead implemented in software which is executing on an embedded computing device or on a personal computer.

An ideal SDR, represented in Figure 1.2, is a reconfigurable radio based solely on software, and has the analog-to-digital conversion occurring directly at the antenna. The modulated analog information signal received at the antenna, is immediately converted to digital domain by an analog-to-digital converter (ADC), and is processed in software by a computing device [22]. Conversely, in the transmitter section, the software produces a modulated information signal in digital domain which is converted to an analog signal by a digital-to-analog converter (DAC) ultimately transmitted through the antenna.

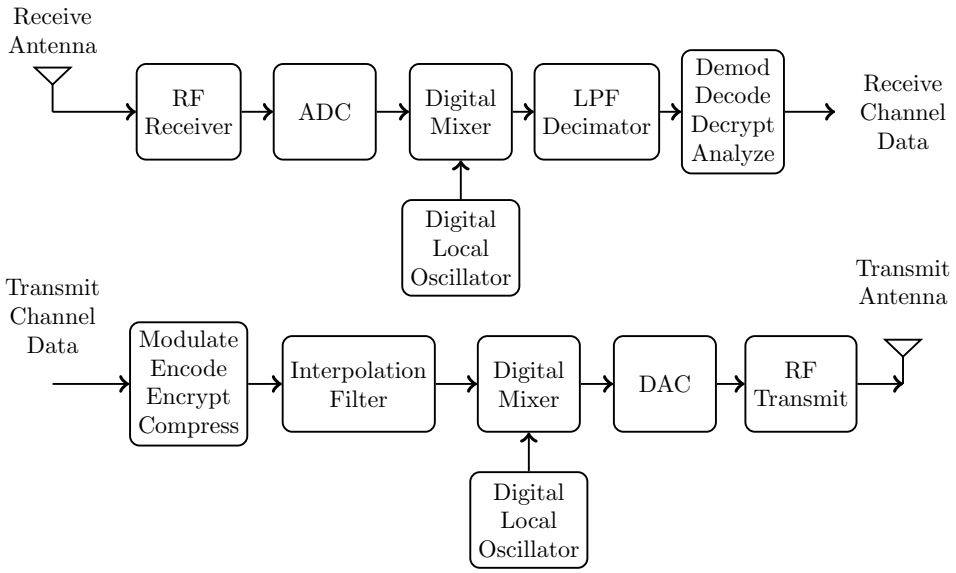

Figure 1.1. Signal processing blocks of a modern communications transceiver system

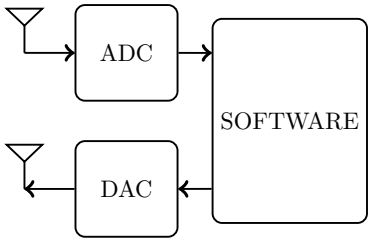

Figure 1.2. Functional block diagram of an ideal software defined radio

The ideal SDR architecture of Figure 1.2 imposes some difficult specifications, [19], upon each of the elements in the system, which necessitates trade-offs towards implementing a practical SDR. Nonetheless, the development of a radio system that can change its operating frequency, modulation, operating bandwidth and network protocol without the need to change the system hardware is highly desirable. Figure 1.3 represents a practical design for a software-defined radio.

In this design, in the receiver, an RF to IF conversion [28] is performed prior to analog-to-digital conversion. The digital output of ADC is then passed on to the software, where it is downcoverted from IF to baseband, and processed further for demodulation. Similarly in the transmitter, the software generates the modulated information signal at baseband,

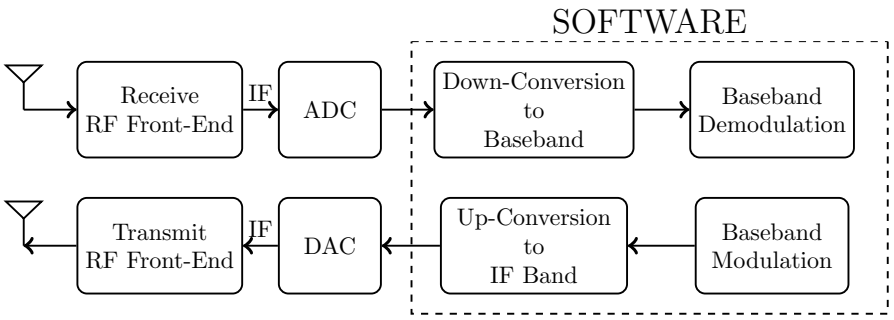

Figure 1.3. Functional block diagram of a practical software defined radio

digitally upconverts it to IF and sends it to the DAC. The DAC converts the digital IF signal to an analog IF signal which, after conversion to RF by the RF front-end, is transmitted through the antenna.

The heart of an SDR is the computing device performing the digital signal processing in software. This computing device is capable of executing complex signal processing algorithms in real time that perform functions such as downconversion, upconversion, modulation, demodulation, encoding, decoding and error checking. Depending upon the cost, needs, power consumption, architecture, software programming methods and ease of software reconfigurability, the computing device can be implemented using a field programmable gate array (FPGA), digital signal processors, general purpose processors, programmable System on Chip (SoC) or other application specific programmable processors. As these devices are reprogrammable, an SDR's function can be modified or, new features added to it, without requiring changes in hardware.

As the current generation of personal computers (PC) are equipped with high speed CPUs and high capacity memories, performing complex signal processing tasks in real-time on them is now possible. With this, a whole new area of research, development and use of SDR has emerged where PCs running general purpose operating systems such as Linux are used to perform complex signal processing tasks in real-time. The ease of developing algorithms on a PC using popular programming languages such as C, C++,

Python, and signal processing tools such as MATLAB has helped in attracting more enthusiasts and researchers towards this technology. Dedicated software frameworks for implementing software defined radios using PCs are currently being developed and two such open source frameworks that are available today are GNU Radio [11] and OSSIE [9].

1.3 GNU Radio and USRP

GNU Radio is an open source project of the Free Software Foundation (FSF) that is currently one of the most popular SDR implementations and is widely used in hobbyist, academic and commercial environments. It is a pure software toolkit developed in C++ and Python programming languages with the design philosophy of performing all the functions of a digital communication system in software instead of hardware. It can perform all the essential signal processing tasks such as, source encoding, channel encoding, encryption, decryption, modulation, demodulation, synchronization, multiplexing, demultiplexing and can be used to implement almost any type of communication system.

As GNU Radio exclusively is a software toolkit, it does absolutely no radio communication without means to interface it to the radio frequency (RF) domain. A device is required to interface the digital software domain of GNU Radio to the analog RF domain. The USRP is one such device that provides the RF Front-End and ADC/DAC capability to a GNU Radio based SDR. Figure 1.4 represents the block diagram of an SDR that uses GNU Radio and the USRP.

The USRP consists of two separate units; a motherboard and one or more daughterboards and, is connected through a Universal Serial Bus (USB) cable to a computer running GNU Radio. The USRP Motherboard performs the intermediate frequency (IF) signal processing of up and down conversions, decimation, interpolation and filtering whereas, the final radio frequency's analog up and down conversion, filtering and amplification is performed by the daughterboards. This way, the USRP provides the interface

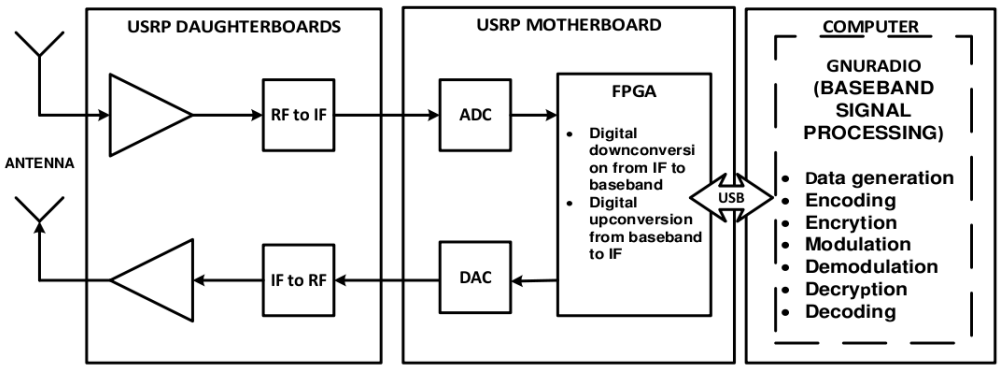

Figure 1.4. A Generic block diagram representation of a GNU Radio based SDR

between the analog signal and the digital baseband signal processed by GNU Radio on the computer and together they form a powerful SDR platform that is easily configurable and expandable.

1.4 Outline of Thesis

The thesis is organized as follows. The next chapter introduces the user to GNU Radio and its architecture along with a method to use this technology for communicating between two ad hoc nodes. Chapter 3 analyses the functioning of a direct-sequence spread binary-phase-shift-keyed (DS-BPSK) transmitter and receiver. Further, we propose the design of our framework for implementing a DS-BPSK based physical layer in GNU Radio. In Chapter 4 we describe the software we developed and present the results from our tests to validate the algorithms and design. Chapter 5 concludes the thesis.

Chapter 2

GNU Radio Design

In this chapter, we begin by describing the architecture of GNU Radio applications. We introduce the reader to various building blocks of a GNU Radio based SDR applications. We then discuss a GNU Radio flowgraph development in `Python`. We later introduce the hardware used for developing software radios and finally finish the chapter by discussing the framework for TCP/IP communication using GNU Radio.

2.1 GNU Radio Architecture

GNU Radio is an open-source software package that can run on various hardware platforms. Coupled with USRP, GNU Radio provides an ideal software platform for developing wireless protocols at the physical and data link layers of the protocol stack. GNU Radio uses a modular, block-based architecture with a hybrid Python/C++ programming model that provides a convenient and high performance platform for the development of software radios. It adopts a flowgraph design, as shown in Figure 2.1, for a simpler abstraction and visualization of an SDR application.

The GNU Radio framework consists of an extensive library of pre-defined and tested signal processing units called *blocks*. Blocks are an abstraction of a C++ class that implements a certain signal processing function while hiding the inner workings and implementation details of the class. They are basic operation units that process continuous data

streams. Each block has a number of input and output streams. The input to the block is the data that is used by the signal processing function, and the output of the block is the data produced by the signal processing function. Currently, GNU Radio provides a large number of signal processing blocks such as filters, interpolators, decimators, waveform generators, clock and carrier synchronization blocks, modulators, demodulators, blocks performing various simple to complex arithmetic operations and transformations.

Although blocks are written in `C++`, a GNU Radio flowgraph is typically implemented in `Python`. Through the use of *Simplifed Wrapper and Interface Generator* (SWIG) [12], the C++ classes representing various blocks are imported to a Python environment and with the use of an extensive API available in GNU Radio an SDR flowgraph is constructed. This has the advantage of simplifying the procedure to develop an SDR in GNU Radio as scripting in Python is simpler than developing applications in C++. The performance critical signal processing tasks are implemented in C++, whereas the high level organization, flowgraph construction, GUI and other less performance-critical functions are implemented in Python.

An SDR is built by creating a graph, as shown in Figure 2.1, where the nodes are units that implement a signal processing function, and the edges that connect these nodes represent the data flow between the nodes. In GNU Radio the nodes are called *blocks*, whereas the edges are often referred to as *streams*. A typical SDR built in GNU Radio will have the following elements:

- **Sources** - A source is a block without an input stream but one or more output streams. It is the head of the processing chain, and it feeds data into the flowgraph so that, it is processed by the other signal processing blocks in the flowgraph. A GNU Radio application will have at least one source. Block A and B in Figure 2.1 represent the source blocks.

- **Sinks** - A sink is a block without an output stream but one or more input streams. It

forms the tail of the processing chain and it is where the signal processing terminates in the flowgraph. A GNU Radio application will have at least one sink. Block F in Figure 2.1 represents a sink with two input streams.

- **Intermediate Blocks** - These blocks are found between a source and the sink and together they complete a GNU Radio flowgraph. The intermediate blocks have both input and output streams and they perform the intermediate signal processing in the flowgraph.

- **Flowgraph** - A GNU Radio application links together each source and sink pair, as well as, any intermediate blocks, (blocks C, D, E in Figure 2.1), that are required to transform the data stream from the source(s) into a format that is understandable by the sink(s). This interconnection of blocks from the source to a sink forms a flowgraph. The information in a flowgraph starts at one or more source blocks, flows sequentially through the intermediate blocks and terminates at one or more sink blocks. The input to a block is called an input stream whereas, the output from the block is called an output stream. The flowgraph may include any number of paths, sources or sinks, as long as there are no loops and no unconnected ports.

- **Schedulers** - A scheduler is associated with each active flowgraph and is responsible for moving data through the flowgraph. It iterates through the blocks in a flowgraph, identifies blocks that have sufficient data on their input(s) and sufficient space on their output(s) to be able to process data. It then triggers the processing function for these blocks.

Figure 2.2 represents the flowgraph of a simple GNU Radio application flowgraph called `dbpsk_test.py`. This application performs differential phase-shift keyed (DBPSK) modulation on randomly generated bits.

Python script `dbpsk_test.py` shown in Listing 2.1 implements the flowgraph of Figure

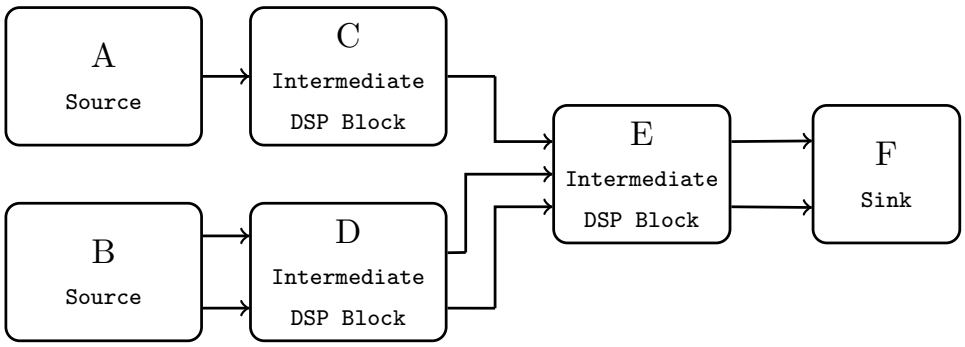

Figure 2.1. GNU Radio flowgraph

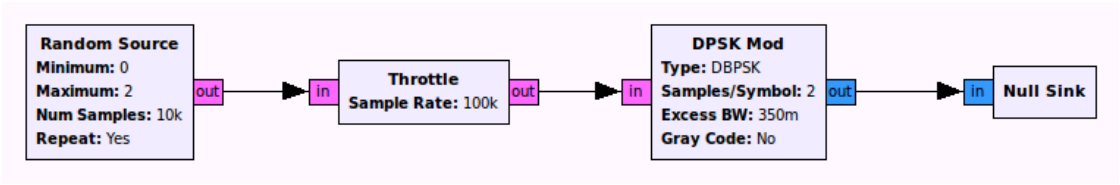

Figure 2.2. An example of a GNU Radio flowgraph

2.2. It uses the *random source* block to produce a stream of random bits. The stream of random bits is then fed into a *throttle* block that controls the speed of execution of a GNU Radio flowgraph by preventing it from hogging the CPU. The controlled output of the throttle block is later fed into a *DPSK Mod* block that performs DBPSK modulation on the input bits. The output of the block is a baseband DBPSK signal which we, in order to complete the flowgraph, discard by feeding it into a *null sink* block. The null sink block functions like Unix's `/dev/null` file.

Line 1 of Listing 2.1 imports module `gr` from the `gnuradio` package which causes most of the GNU Radio signal processing blocks to be imported into `Python` environment. This module is always loaded in a GNU Radio application. Similarly, line 2 imports the GNU Radio module that provides the functionality of most of the modulators and demodulators implemented in GNU Radio. Lines `5` to `27` define a class called `dbpsk_top_block` which is derived from `gr.top_block` class. This class is a container for the flowgraph and by deriving it from `gr.top_block` all the essential functions necessary for creating a flowgraph

are inherited.

```
1  from gnuradio import gr
2  from gnuradio import blks2
3  import numpy
4
5  class dbpsk_top_block(gr.top_block):
6
7    def __init__(self):
8      gr.top_block.__init__(self)
9
10     self.samp_rate = 100000
11     # Generate a list of random bits
12     self.random_bits = map(numpy.random.randint(0, 2, 10000))
13
14     self.random_source  = gr.vector_source_b(self.random_bits, True)
15     # DBPSK modulator
16     self.dbpsk_mod = blks2.dbpsk_mod(
17       samples_per_symbol=2,
18       excess_bw=0.35,
19       gray_code=False,
20       verbose=False,
21       log=False)
22     # rate limiter
23     self.throttle = gr.throttle(gr.sizeof_char, self.samp_rate)
24     # null sink
25     self.null_sink  = gr.null_sink(gr.sizeof_gr_complex)
26
27
28     self.connect(self.random_source, self.throttle)
29     self.connect(self.throttle, self.dbpsk_mod)
30     self.connect(self.dbpsk_mod, self.null_sink)
31
32 if __name__ == '__main__':
33   try:
34     dbpsk_top_block().run()
35   except KeyboardInterrupt:
36     pass
```

Listing 2.1. Python script implementing the flowgraph of Figure 2.2

The variable `sample_rate` defines the sample rate of the signals that are generated by various blocks in the flowgraph. The random source block of the flowgraph is instantiated as an object of GNU Radio module `gr.vector_source_b`. This module repeatedly outputs a list containing `10000` randomly generated bits (line `12`). In line `16` to `21` a DBPSK modulator is instantiated as an object of module `blks2.dbpsk_mod`. In line `23` a throttle block is instantiated as an object of class `gr.throttle`. Similarly, in line `25` a null sink is instantiated as an object of module `gr.null_sink`.

Once the blocks are instantiated, they are connected together to complete the graph.

This is done in lines `28-30`. The remaining part of the code is synonymous to a `main()` function in a `C/C++` program, and is called upon execution of `dbpsk_test.py`.

2.2 Universal Software Radio Peripheral

The USRP is a hardware device designed and manufactured by Ettus Research Inc, and is widely used as an affordable hardware front end for various software defined radio systems. Figure 2.3 shows the picture of the first generation USRP. The rest of the discussion in this section has been included from references [11] and [15].

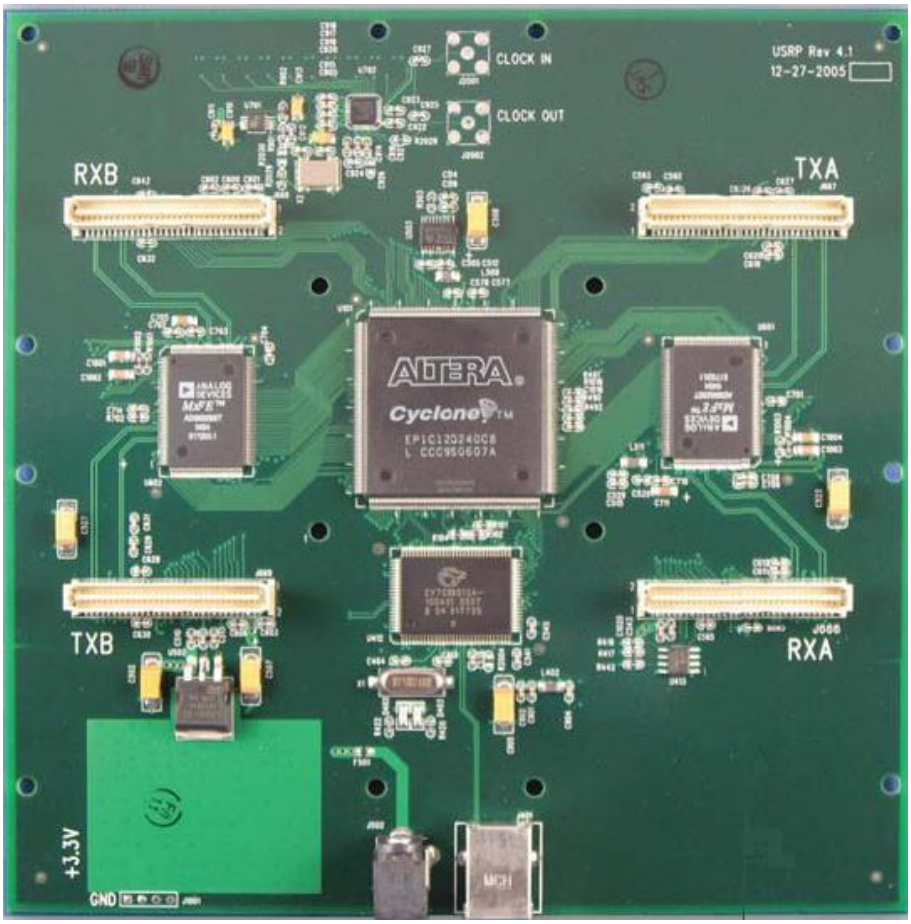

Figure 2.3. Universal Software Radio Peripheral

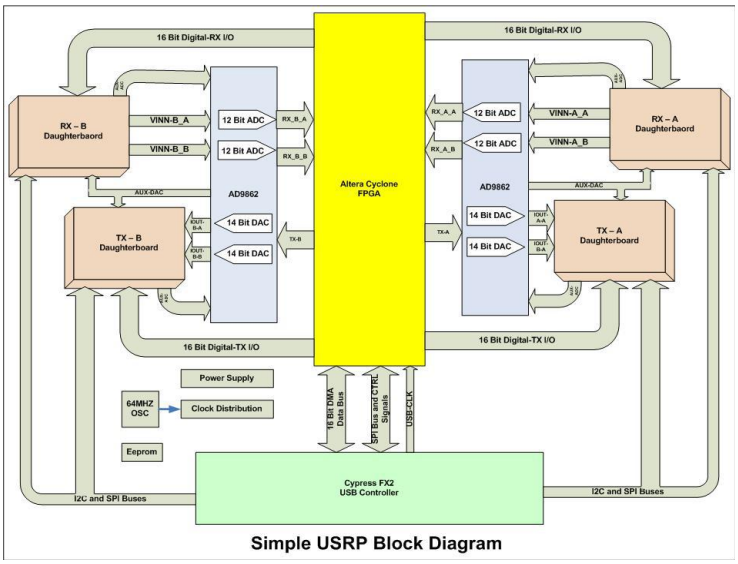

Figure 2.4. A simple USRP block diagram

As shown in figure 2.4 the USRP consists of the following major components,

- Analog to Digital Converter (ADC) section
- Digital to Analog Converter (DAC) section
- Field Programmable Gate Array (FPGA)

The ADC section consists of four high speed 12-bit ADCs with a sampling rate of 64MHz and capable of digitizing a band as wide as 32MHz. The full scale voltage range of the ADCs is 2V peak-to-peak and the input is 50Ω differential. A software programmable gain amplifier (PGA) with an amplifier gain of upto 20dB is used before the ADCs to amplify the input signal in order to utilize the entire input range of the ADC in case of weak signals.

The DAC section, which is present in the transmission section, consists of four high speed 14-bit DACs with a sampling rate of 128MHz and capable of converting a signal of bandwidth as wide as 64MHz from digital to analog. The DACs are capable of providing

an output of 1V peak to a 50Ω differential load. A software PGA with a gain up to 20dB is also used to amplify the signal after it is coverted to analog by the DACs.

The FPGA section is the most important part of the USRP. The host computer running the software radio algorithms produces digital signal samples at a lower sample rate that is incompatible with the ADC/DAC. The FPGA uses the stream of data samples from the host computer and performs high sample rate signal processing to enable the resultant digital signal to be compatible with the ADC/DAC requirements. The high sample-rate processing is performed on the FPGA while the lower sample-rate processing is done on the host computer running the SDR algorithms.

In signal receive mode the FPGA is programmed to function as a digital-down-converter (DDC). The output of the ADC is a discrete IF signal at a sample rate of 64Msps with the data bandwidth centered around the intermediate frequency f_{if}. The FPGA functioning as a DDC translates the bandwidth of this signal from f_{if} to base-band and reduces the sample rate from 64Msps to a lower rate that is capable of being transferred over the USB bus and is within the host computer's processing capability.

When the USRP is used as a transmitter, the FPGA functions as a digital-up-converter (DUC). The discrete baseband signal generated by the host computer is sent over the USB bus to the USRP. As the signal generated by the computer is at a lower rate compared to the DAC sampling rate, the FPGA interpolates the low rate baseband data signal by a factor N chosen from the range {4, 512} such that the final sample rate of the interpolated signal is 128Msps. The interpolated signal is then translated from baseband to an intermediate frequency band by mixing it with an intermediate frequency carrier and finally the DAC converts the discrete IF signal to an analog IF signal.

The daughterboards transform a USRP motherboard into a complete RF transceiver system. They are analogous to the RF front end of a radio transceiver system. A daughterboard consists of a programmatically tunable oscillator, a mixer and bandpass filters

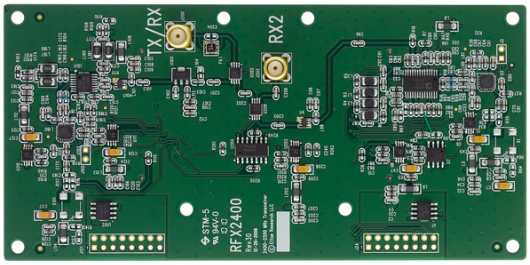

Figure 2.5. RFX-2400 daughterboard

and it converts the IF signal coming from the USRP motherboard to an RF signal in transmit mode or converts the received RF signal to an IF signal in the receive mode.

Ettus Research Inc, produces a wide range of daughterboards that function in various ISM radio bands [5]. In our experiments, we used an RFX-2400 daughterboard which has a frequency range of 2.3 GHz to 2.9 GHz with a maximum transmit power of 17dBm and is shown in Figure 2.5.

2.3 Framework for TCP/IP Communication

As indicated earlier, our design goal is to implement a framework to implement functional prototypes of direct-sequence spread-spectrum (DSSS) based MANET nodes. GNU Radio provides a framework that integrates Linux's network layer functionality with the physical layer (PHY) and medium access (MAC) layer functionality implemented in GNU Radio. This enables easy prototyping of MANET nodes that are capable of transmitting application data that are based on TCP/IP protocol but use a PHY and MAC implemented in GNU Radio.

Figure 2.6 describes this framework using the TCP/IP model. At the top of the framework is the *Application* layer and it represents various TCP/IP based applications such as SSH, SFTP and Ping. The next block represents the *Transport*, *Network* and *Link*

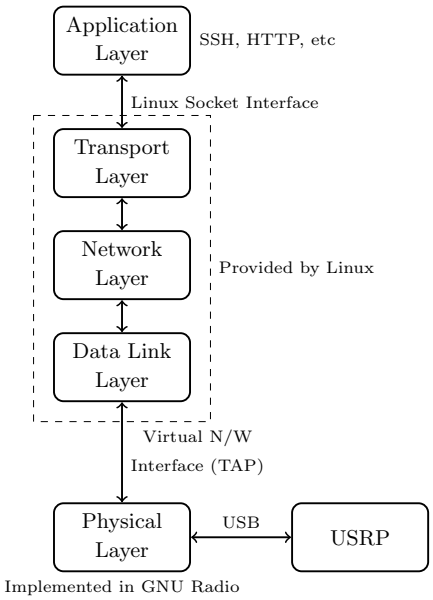

Figure 2.6. `tunnel` framework

layers which are implemented natively in the Linux kernel. The last block represents the *MAC* and *Physical* layers and are implemented in GNU Radio.

The PHY is interfaced to the upper layers through Linux's native virtual network kernel device TUN/TAP [10]. TUN/TAP provides packet reception and transmission from user space programs and can be used as an Ethernet device, which, instead of receiving packets from physical media, receives them from user space program and instead of sending packets via physical media writes them to the user space program. In our application, the user space program is a Python script that instantiates two threads called `transmit_path` and `receive_path`. `transmit_path` reads packets from TUN/TAP and sends it to PHY for transmission while, `receive_path` sends packets received through PHY to the TUN/TAP device.

Listing 2.2 describes the algorithm used by the user space program to access TUN/-TAP device to send and receive packets `payload` using GNU Radio/USRP. In `main()` method, an instance of the application class `app_top_block` is created. This class upon initialization creates two objects called `transmit_path` and `receive_path` which run as

separate threads. After initialization of these threads, the TUN/TAP interface is opened for read and write in line 47.

Next we set the carrier sense threshold in line 49 and instantiate the carrier-sense multiple access (CSMA) object class `csma_mac`. This class consists of two methods; `phy_rx_callback` and `main_loop()`. When PHY receives a packet through `receive_path` it calls the callback [13] function `phy_rx_callback()` and pushes the received `payload` to the upper layer by writing the `payload` object to the TUN/TAP device's file descriptor `tunnel_fd`.

`main_loop()` method implements the carrier-sense MAC and it controls the access of the shared transmission medium. It continuously checks for available packets at the TUN/TAP interface (line 27). If there is data available, it calls the method `carrier_sensed()`, which is implemented in the `receive_path` class, to check if there is a carrier detected on the medium. If `carrier_sensed()` returns `True`, it means that the channel is busy and the protocol keeps executing *backoff* until the channel becomes idle as the transmission attempt is delayed for a time period equal to `delay`. This backoff function is implemented as a thread sleep using the Python method `time.sleep(delay)`. The carrier is sensed before every transmission attempt and upon a failed transmission attempt, `delay` is increased exponentially (lines 38-39). When `carrier_sensed()` returns `False`, the payload is transmitted by the transmit thread by calling its method `send_packet(payload)`.

We utilize this framework in our proposed system to communicate using TCP/IP using a DS-BPSK based physical layer. The MAC and PHY implementation consists of implementations of `transmit_path`, `receive_path` and the algorithm for CSMA.

```
1 class app_top_block(gr.top_block):
2
3   def __init__():
4     transmit_path = initialize_transmit_path()
5     self.connect(transmit_path)
6
7     receive_path = initialize_receive_path()
8     self.connect(receive_path)
```

```
9
10 class csma_mac(object):
11
12   def __init__(self, tunnel_fd, transmit_path, receive_path):
13     """
14     Initialize CSMA MAC object
15     """
16
17   def phy_rx_callback(self, ok, payload):
18
19     if ok:
20       os.write(tunnel_fd, payload)
21
22   def main_loop(self):
23
24     minimum_delay = 0.001 # delay in seconds
25
26     while True:
27       payload = os.read(tunnel_fd, PACKET_SIZE)
28
29       if payload is NULL:
30         transmit_path.send_pkt(eof=True)
31         break
32
33       delay = minimum_delay
34
35       while receive_path.carrier_sensed():
36         sys.stderr.write("Entering back-off period")
37         sleep(delay)
38         if delay < 0.050:
39           delay = delay * 2 # exponential back-off
40
41       transmit_path.send_packet(payload)
42
43 def main():
44
45   app = app_top_block()
46
47   tunnel_fd = open_tun_interface("/dev/net/tun")
48
49   app.receive_path.set_carrier_threshold(THRESHOLD_dB)
50
51   mac = csma_mac(tunnel_fd, app.transmit_path, app.receive_path)
52
53   mac.main_loop()
54
55 if __name__ == '__main__':
56   try:
57     main()
58   except KeyboardInterrupt:
59     pass
```

Listing 2.2. Python pseudo code describing framework for TCP/IP communication using GNU Radio

Chapter 3

Direct Sequence Spread Spectrum

In this chapter, we describe the transmitter and receiver design in GNU Radio for the proposed DS-BPSK transceiver. We begin by describing a DS-BPSK transmitter and receiver. Later, we present the GNU Radio flowgraphs for the DS-BPSK transmitter and the receiver, and identify and describe the functions of the signal processing blocks used in the flowgraphs. We also identify the native GNU Radio blocks used in the flowgraph and describe the design of the non-native blocks that were developed.

Spread-spectrum is a telecommunications technique in which the information signal is spread over a bandwidth considerably greater than is necessary so as to resist jamming and other interference. In order to be considered spread-spectrum, a signal must have the following characteristics [23]:

- The transmitted signal energy must occupy a bandwidth which is larger than the information bit rate and which is approximately independent of the information bit rate.
- Demodulation must be accomplished, in part, by correlation of the received signal with a replica of the signal used in the transmitter to spread the information signal.

In spread-spectrum (SS) modulation the lower rate information signal, $d(t)$, is modified directly or indirectly by a sequence, $\{c_n\}$, also known as the *spreading* or *chipping* sequence such that, the modified signal occupies a bandwidth that is much larger than that of the

original information signal. This subsequent increase in bandwidth results in an improved system performance of a spread-spectrum communication system without requiring high signal to noise ratio (SNR).

Although spread-spectrum modulation can be implemented in multiple ways [24, 25], we adopt direct-sequence spread-spectrum (DSSS) technique utilizing binary phase shift keying (BPSK) due to its simplicity and ease of implementation. We call the signal modulated using this technique as DS-BPSK signal. In the following sections, we will present the design of the DS-BPSK transmitter and receiver that we intend to implement along with a mathematical representation of the signals that are generated by various components of the designed system.

3.1 DS-BPSK Transmitter

A DS-BPSK signal is generated by the direct mixing of the data with a spreading waveform before the final carrier modulation using BPSK. Figure 3.1 represents the functional block diagram design of a DS-BPSK modulator. This design is based on the digital DSSS modem design of [1].

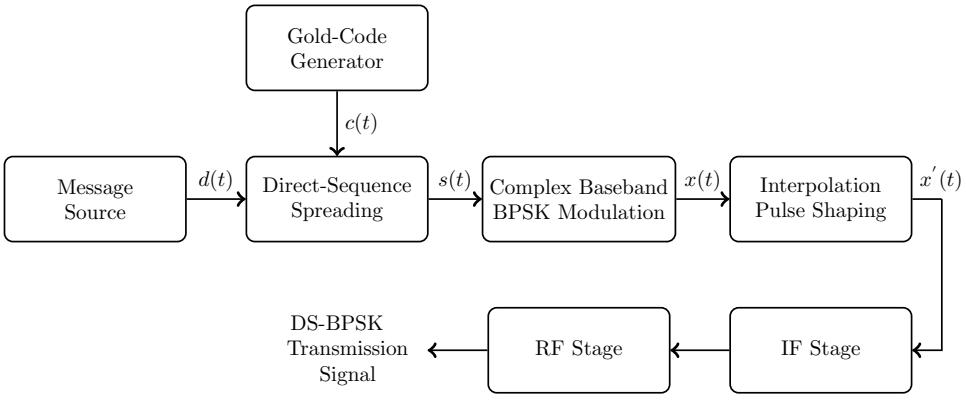

Figure 3.1. Functional block diagram of a simple DSSS modulator

Let $d(t)$ represent a binary data pulse train generated by the *Message Source* at a

rate of $\frac{1}{T_b}$ bits per second. If $\{d_m\}$ denotes the sequence of binary data bits, $d(t)$ can be mathematically represented as,

$$d(t) = \sum_{m=0}^{\infty} d_m \delta\left(t - mT_b\right), \tag{3.1.1}$$

where,

$$d_m \in \{-1, 1\} \quad \forall \quad m \in \{0, 1, \cdots, \infty\}. \tag{3.1.2}$$

Similarly, let $c(t)$ represent a binary pulse train representing a sequence of chips of a unique Gold-code sequence G of length N generated by the *Gold Code Generator* at a rate of $\frac{1}{T_c}$ chips per second. In multiple access systems G is chosen from a set of similar Gold-codes. If $\{c_n\}$ represents the binary chips belonging to the Gold-code sequence, $c(t)$ can be mathematically represented as,

$$c(t) = \sum_{n=0}^{\infty} c_n \delta\left(t - nT_c\right), \tag{3.1.3}$$

where,

$$c_n \in \{-1, 1\} \quad \forall \quad n \in \{0, 1, \cdots, \infty\}. \tag{3.1.4}$$

$d(t)$ and $c(t)$ are input into a *Direct-Sequence Spreading* block that performs direct-sequence spreading on the data sequence bits by multiplying each data bit with N chips of the Gold-code sequence. If the input bit is d_m, the corresponding spread-spectrum signal is a contiguous set of N spread-spectrum chip pulses and is mathematically represented as,

$$s_m(t) = d_m \sum_{n=0}^{N-1} c_n \delta\left(t - nT_c\right), \tag{3.1.5}$$

The ratio of T_b to T_c, is referred to as the *processing gain* (N) of the spread-spectrum system and is often used to characterize system performance [24]. Using (3.1.5), the

spread-spectrum binary sequence is mathematically represented as:

$$\begin{aligned} s(t) &= \sum_{m=0}^{\infty} s_m \left(t - mT_b\right) \\ &= \sum_{m=0}^{\infty} d_m \sum_{n=0}^{N-1} c_n \delta \left(t - nT_c - mT_b\right) \\ &= \sum_{m=0}^{\infty} \sum_{n=0}^{N-1} d_m c_n \delta \left(t - nT_c - mT_b\right) \end{aligned} \tag{3.1.6}$$

$s(t)$ is then passed onto a *Complex Baseband Modulator* that performs BPSK modulation on the input spread spectrum bits where, the input bit $+1$ is converted to complex symbol $(+1 + j0)$, and -1 is converted to $(-1 + j0)$ to produce, the complex baseband DS-BPSK signal $x(t)$ consisting of an in-phase component $x_i(t)$ as defined in (3.1.7) and a zero quadrature component $x_q(t)$.

$$x_i(t) = \sum_{m=0}^{\infty} \sum_{n=0}^{N-1} d_m c_n \delta \left(t - nT_c - mT_b\right), \tag{3.1.7}$$

$$x_q(t) = 0, \tag{3.1.8}$$

The baseband DS-BPSK signal is then passed to an *Interpolation Pulse Shaping* block where each BPSK symbol is interpolated to K samples according to some pulse shaping function [17]. If rectangular pulse shaping is used, the pulse shaping process yields the following in-phase and quadrature components for the baseband DS-BPSK output $x^{'}(t)$:

$$\begin{aligned} x_i^{'}(t) &= \sum_{m=0}^{\infty} \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} d_m c_n \ \Pi_{T_c} \left(t - \frac{kT_c}{K} - nT_c - mT_b\right), \\ x_q^{'}(t) &= 0, \end{aligned} \tag{3.1.9}$$

where $\Pi_{T_c}(.)$ represents the rectangular pulse defined as,

$$\Pi_{T_c} = \begin{cases} 1, & 0 < \ t \ \leq T_c \\ -1, & \text{otherwise.} \end{cases} \tag{3.1.10}$$

The pulse-shaped baseband DS-BPSK signal $x^{'}(t)$ is then multiplied with digital complex sinusoid of frequency f_{IF} to produce the digital IF signal $x_{\text{IF}}(t)$ as shown in the equation

below.

$$\begin{aligned} x_{\text{IF}}(t) &= Re\left[x^{'}(t)e^{-j2\pi f_{\text{IF}}t}\right], \\ &= x_i^{'}(t)\cos\left(2\pi f_{\text{IF}}t\right) \end{aligned} \tag{3.1.11}$$

The digital IF signal is then passed on to the DAC block followed by the RF block, where it is converted to an analog signal and is modulated onto a carrier signal of radio frequency f_c to produce the analog DS-BPSK transmission signal.

3.2 DS-BPSK Receiver

Figure 3.2 represents a functional block diagram of a direct-conversion DS-BPSK receiver. This design is based upon the design of DSSS modem of [1]. The receiver can be divided into two separate sections; the *Downconversion* section and the *Baseband Demodulator* section. The downconversion section consists of the RF and IF stages and converts the received analog DS-BPSK signal into a discrete baseband signal sampled at a rate of K samples per chip duration (T_c). The baseband demodulator section demodulates the discrete baseband DS-BPSK signal and determines the transmitted data from the signal.

Let us assume that the transmission DS-BPSK signal was transmitted via a distortion-less channel having a transmission delay of T_d, and was received at the receiver together with some type of interference and Gaussian noise (AWGN). The received signal can also suffer from carrier frequency offsets introduced due to Doppler shifts (in case of mobile receivers), and also due to the physical differences between the local oscillator crystals in the transmitter and the receiver. If the receiver's local oscillator is tuned to f_c, the received carrier frequency may appear with respect to the receiver's local oscillator at an offset of Δf. Using the definition of the baseband DS-BPSK signal at the transmitter as shown in (3.1.9) the received discrete complex baseband DS-BPSK signal $r(t)$ at the

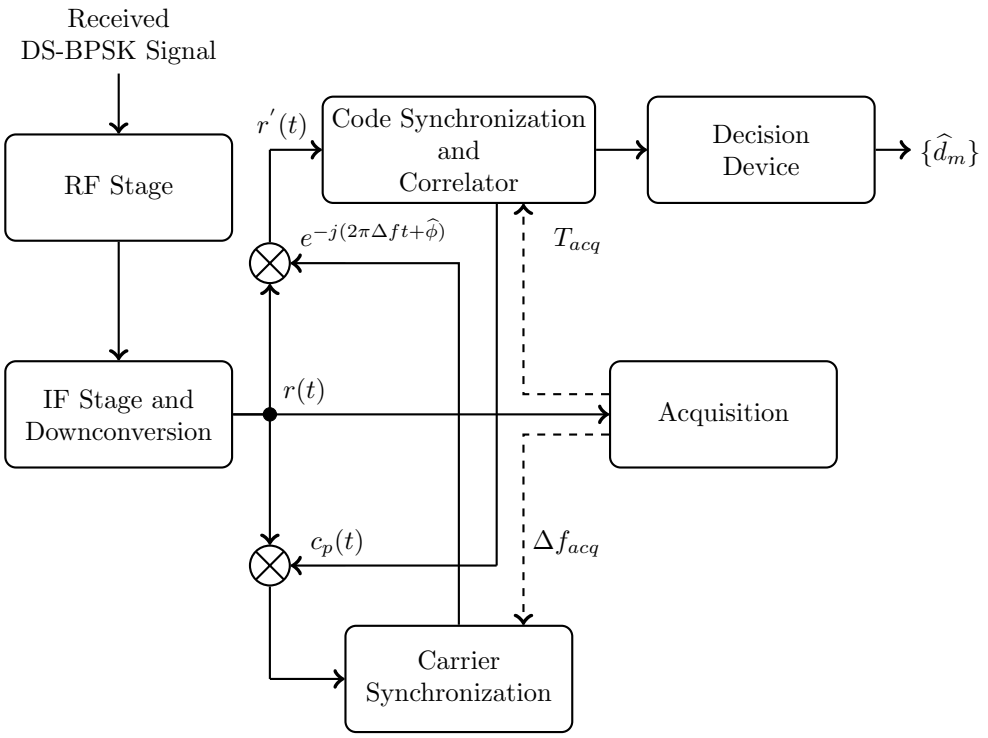

Figure 3.2. A flowgraph representing all the stages of a DSSS demodulation

output of the *IF Stage and Downconversion* block can be represented as,

$$r(t) = r_i(t) + jr_q(t) \tag{3.2.1}$$

where, the in-phase component $r_i(t)$ is,

$$\begin{aligned} r_i(t) &= d\,(t - T_d)\, c\,(t - T_d) \cos\left(2\pi\Delta f t + \phi\right) + n_i(t) && (3.2.2) \\ &= \sum_{m=0}^{\infty} \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} d_m c_n \Pi_{T_c} \left(t - T_d - \frac{kT_c}{K} - nT_c - mT_b \right) \cos\left(2\pi\Delta f t + \phi\right) + n_i(t), && (3.2.3) \end{aligned}$$

and the quadrature component $r_q(t)$ is,

$$\begin{aligned} r_q(t) &= d(t - T_d) c(t - T_d) \sin\left(2\pi\Delta f t + \phi\right) + n_q(t) && (3.2.4) \\ &= \sum_{m=0}^{\infty} \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} d_m c_n \Pi_{T_c} \left(t - T_d - \frac{kT_c}{K} - nT_c - mT_b \right) \sin\left(2\pi\Delta f t + \phi\right) + n_q(t). && (3.2.5) \end{aligned}$$

ϕ in the above equations represents the phase difference between the received carrier and the local carrier whereas, $n_i(t)$ and $n_q(t)$ represents the in-phase and quadrature components of noise present in the received signal.

Demodulation of a spread-spectrum signal is accomplished in part by remodulating it with the spreading code appropriately delayed as shown in Figure 3.3. This remodulation, or *correlation* of the received signal with the delayed spreading waveform is called *despreading*, and is a critical function in all spread-spectrum systems. Due to the carrier frequency offset between the transmit and received signals, the spectrum of $r(t)$ is centered away from the baseband frequency of 0Hz, to the offset frequency Δf. In order to demodulate the signal correctly, the spectrum of $r(t)$ must be translated to be centered at 0Hz. This is done by mixing $r(t)$ with the quadrature carrier $e^{j(2\pi\Delta f t+\phi)}$ [21] as shown in Figure 3.3.

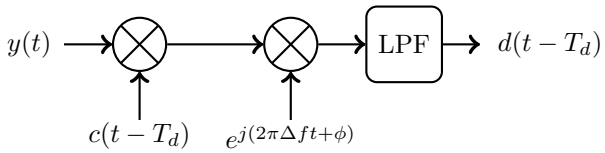

Figure 3.3. Functional block diagram of a DSSS demodulator

Spread-spectrum demodulation requires that the transmitter and receiver spreading waveforms be synchronized at all times. If the two waveforms are out of sync by as little as one chip, insufficient signal energy will reach the receiver data demodulator for reliable data detection. This task of achieving and maintaining code synchronization is always delegated to the receiver.

There are two components to solving this synchronization problem [2, 26]. The first component is the determination of the signal delay T_d and the offset frequency Δf at the beginning of the demodulation process. This is called *code and frequency acquisition* and is described in Section 3.7. It is carried out by the *Acquisition* block in the DS-BPSK receiver of Figure 3.2.

The second component of the synchronization problem is the problem of maintaining code and carrier synchronization after initial acquisition. This problem is called *code and carrier tracking* and is described in Section 3.4. The carrier synchronization is carried out by the *Carrier Synchronization* block and the code synchronization is carrier out by the *Code Synchronization and Correlator* block in the DS-BPSK receiver of Figure 3.2.

Once synchronization is established in the DS-BPSK receiver, the synchronization blocks will generate the following two waveforms:

- The carrier synchronization block generates a quadrature carrier signal that is used to translate $r(t)$ to baseband before despreading. This quadrature carrier is represented as,

$$e^{j(2\pi\Delta f t+\widehat{\phi})} = \cos\left(2\pi\Delta f t + \widehat{\phi}\right) + j\sin\left(2\pi\Delta f t + \widehat{\phi}\right), \tag{3.2.6}$$

  where $\widehat{\phi}$ is the tracking block's best estimate of the received carrier phase such that under synchronization,

$$\widehat{\phi} \approx \phi \tag{3.2.7}$$

- The code synchronization block generates a local replica of the transmitter's Gold-code waveform sampled at the same rate as $r(t)$. This replica is also known as the *prompt* signal $\mathrm{c}_p(t)$, and is synchronized in time with the spreading signal present on the received signal.

$$c_p(t) = \sum_{m=0}^{\infty}\sum_{n=0}^{N-1}\sum_{k=0}^{K-1} c_n \Pi_{T_c}\left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d\right), \tag{3.2.8}$$

  where $\widehat{T}_d$ is the code tracking loop's best estimate of the received signal delay T_d such that, under synchronization,

$$\widehat{T}_d \approx T_d \tag{3.2.9}$$

The prompt signal $c_p(t)$ is mixed with the received signal $r(t)$ prior to feeding it to the carrier synchronization block. This mixing results in despreading of $r(t)$, and the

despread product becomes a data modulated carrier signal. The in-phase and quadrature components of the despread signal $r^{'}(t)$ are:

$$\begin{aligned} r_i^{'}(t) &= \sum_{m=0}^{\infty} d_m \cos\left(2\pi\Delta f t + \phi\right) + n_i^{''}(t), \\ r_q^{'}(t) &= \sum_{m=0}^{\infty} d_m \sin\left(2\pi\Delta f t + \phi\right) + n_q^{''}(t), \end{aligned} \tag{3.2.10}$$

where, $n_i^{''}(t)$ and $n_q^{''}(t)$ are the in-phase and quadrature noise components. The carrier synchronization block uses the despread signal $r^{'}(t)$ to track the carrier and estimate the carrier phase ϕ.

In the code synchronization section, $r(t)$ is multiplied with the synchronized local carrier output of the carrier synchronization block to produce $r_1(t)$ where,

$$r_1(t) = r(t)e^{j(2\pi\Delta f t+\phi)}. \tag{3.2.11}$$

When condition of (3.2.7) is satisfied, ignoring the noise component, the in-phase component of the product ($r_{1i}(t)$) is devoid of carrier offset and the quadrature component ($r_{1q}(t)$) is approximately equal to zero as shown below.

$$r_{1i}(t) = \sum_{m=0}^{\infty} d_m \sum_{n=0}^{N-1}\sum_{k=0}^{K-1} c_n \Pi_{T_c}\left(t - T_d - \frac{kT_c}{K} - nT_c - mT_b\right) \tag{3.2.12}$$

$$r_{1q}(t) \approx 0 \tag{3.2.13}$$

$r_1(t)$ is then multiplied with the prompt sequence $c_p(t)$ and integrated over a period equal to the spreading sequence period (NT_c), to produce a result of integration that can be used to determine the transmitted bit sequence. Ignoring the noise component, the result of the correlation for any m over a code period is,

$$I_m \approx d_m \sum_{n=0}^{N-1}\sum_{k=0}^{K-1} c_n^2 \Pi_{T_c}^2\left(t - T_d - \frac{kT_c}{K} - nT_c - mT_b\right) \tag{3.2.14}$$

Due to the excellent autocorrelation properties of Gold-codes, the integration above raises the power in the data signal by a factor of 'KN' while, simultaneously reducing the noise

power due to its low cross-correlation values. Ultimately,

$$I_m \approx KNd_m \tag{3.2.15}$$

The *Decision Device* compares I_m with a predefined threshold T_H and outputs the estimated data bit value $\widehat{d}_m$ as,

$$\widehat{d}_m = \begin{cases} +1 & \text{if } I_m \geq T_H, \\ -1 & \text{if } I_m \leq -T_H. \end{cases} \tag{3.2.16}$$

3.3 DS-BPSK based PHY Design in GNU Radio

Physical layer is the interface between the medium-access layer and the transmission medium. It is the entity in charge of acutal transmission using different modulation schemes over the medium. It performs various encoding and signaling functions such as encoding, synchronization, modulation, that transform the digital data bits from the upper layer frames into signals that can be sent over the medium. Similary, it demodulates and decodes the received signal into digital bits and pushes them to the upper layers to be transformed into data packets.

As mentioned in the earlier chapter, GNU Radio coupled with USRP provides an excellent platform for developing PHY for software radios. The implementation of physical layer in GNU Radio means performing most of the physical layer signaling in software. Referring to Figure 3.4, the implementation requires development of the `transmit_path` and `receive_path` classes along with the algorithm for carrier-sense MAC and the physical layer block that performs baseband binary-phase shift-keyed spread-spectrum modulation and demodulation.

In the next two subsections we will present the GNU Radio flowgraphs for performing baseband DS-BPSK modulation and demodulation and also describe the functions of various native GNU Radio blocks and the custom blocks that are used to implement these flowgraphs.

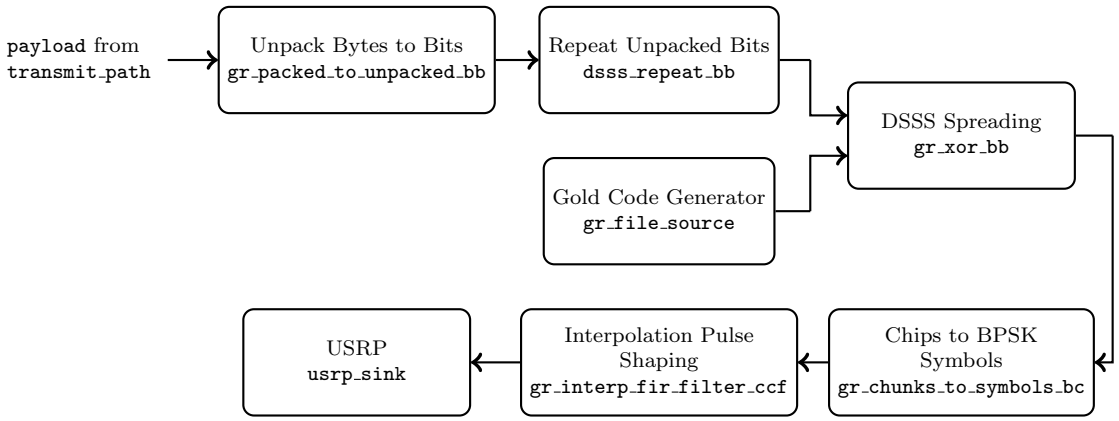

Figure 3.4. GNU Radio flow fraph of DS-BPSK modulator

3.3.1 GNU Radio flowgraph of a DS-BPSK Modulator

Figure 3.4 represents the flowgraph for performing DS-BPSK modulation in GNU Radio. The input to the flowgraph is the packet payload that is passed on by the class `transmit_path` by calling it's method `send_packet(payload)`, as seen in Line 41 of Listing 2.2. `payload` is a packet data structure of size `PACKET_SIZE` bytes or less (see Line 27 of Listing 2.2). The flowgraph unpacks these bytes and performs complex baseband modulation on the unpacked bits as described below.

- `gr_packed_to_unpacked_bb` - This is a native GNU Radio block [7] that converts a stream of data bytes into an output stream of unpacked bits where, each bit is represented by a `char` data type. For example, a data byte {`0xAA`} is converted to {`0x01 0x00 0x01 0x00 0x01 0x00 0x01 0x00`}. The output of this block is passed on to the next block named `dsss_repeat_bb`.

- `dsss_repeat_bb` - This is a GNU Radio compliant custom made block with a `C++` class constructor as `dsss_repeat_bb(int length_pn, int n_spread=1)`. This block repeats every input byte a "`length_pn*n_spread`" times and outputs them onto the output stream. This block is used prior to spreading using `gr_xor_bb` so that, the rate of both the inputs to `gr_xor_bb` is the same.

For example, if `length_pn=3` and `n_spread=2` and the input to the block is {`0x01`}, then the output is {`0x01`} repeated 6 times, i.e. {`0x01 0x01 0x01 0x01 0x01 0x01`}. The output of this block is passed as the first input to the next block `gr_xor_bb`.

- `gr_file_source` - This is a native GNU Radio block that reads a file repetitively and outputs the contents of the file onto its output stream [6]. In this flowgraph it repetitively reads a binary file containing the Gold-code sequence of length `pn_length`.

  For performance reasons we chose to save the Gold-code sequence in a binary file and read it using `gr_file_source`, instead of using the custom Gold-code sequence generator block that outputs a stream of Gold-code sequence chips.

- `gr_xor_bb` - This is a native GNU Radio block that performs exclusive-OR across all input streams [8]. In this flowgraph, the block performs exclusive-OR over the output stream of `dsss_repeat_bb`, which is the repeated bits of the packet payload, and the output stream of `gr_file_source` which is the Gold-code sequence bits or chips.

  The output is a stream of direct-sequence spread binary signal that is passed onto the next block `gr_chunks_to_symbols_bc`.

- `gr_chunks_to_symbols_bc` - This is a native GNU Radio block that converts an input stream of unpacked bytes into an output stream of complex constellation symbols. The constellation symbols are passed as a vector to the constructor of this class as shown in [**?**].

  As we are using BPSK modulation the constellation symbol vector passed to this block is {`1+0j,-1+0j`}. Each of the direct-sequence spread chips in the output stream of `gr_xor_bb` is converted to a BPSK symbol and passed onto the output stream of this block.

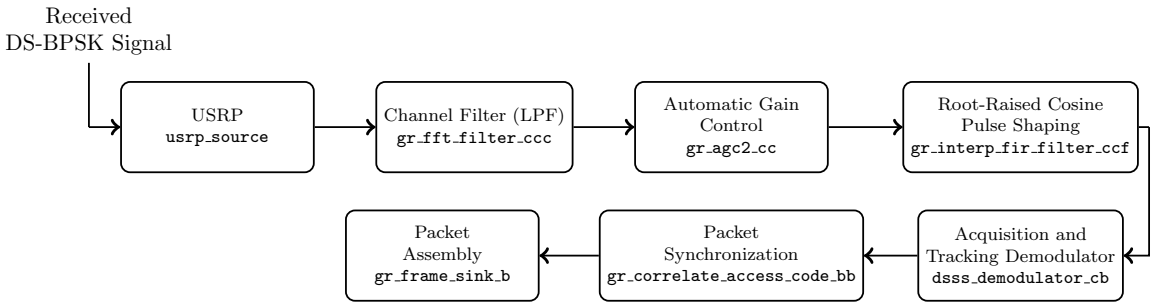

Figure 3.5. GNU Radio flowgraph of DS-BPSK demodulator

- `usrp_sink` This block is the GNU Radio's abstraction of the USRP's driver. It provides the interface to set the USRP parameters such as, frequency, interpolation and gain. As mentioned earlier, in transmit mode, the USRP performs upconversion of the low rate baseband signal coming over the USB bus to a higher rate discrete IF signal, before sending it to the daughterboard for the final RF conversion and transmission.

  The interpolation rate chosen for the USRP determines the rate of the data signal (r_b). As the USRP's DAC consumes samples at a rate of 128M samples per second (sps) the data rate is effectively controlled by the choice of the interpolation rate (i) as,

  $$r_b = \frac{128 \times 10^6}{i \times s} \text{ bps}, \tag{3.3.1}$$

  where s is the samples per symbol value that was used in the interpolation-pulse-shaping block.

Finally, the output of the USRP is the DS-BPSK transmission signal.

3.3.2 GNU Radio flowgraph of a DS-BPSK Demodulator

Figure 3.5 represents the flowgraph for performing DS-BPSK demodulation in GNU Radio. The various blocks and their function is described below.

- `usrp_source` - This block is GNU Radio's abstraction of USRP's driver when functioning in receive mode. It provides the interface to set the USRP parameters such as, receiver frequency, decimation rate and gain. In receive mode, the USRP performs downconversion of the received analog signal and transfers the downcoverted digital samples over the USB bus to be processed on the computer. The decimation rate chosen for the USRP determines the sampling rate of the signal entering the computer. As the USRP's ADC samples the analog signal at 64Msps, the sampling rate of the baseband signal available to be processed on the computer is effectively controlled by the choice of the decimation rate (d). If f_s is the sampling rate of the baseband signal,

$$f_s = \frac{64 \times 10^6}{d} sps, \tag{3.3.2}$$

represents the baseband signal's sampling rate. Depending upon the transmitted signal's data rate, d is chosen such that the Nyquist criterion is satisfied.

- `gr_fft_filter_ccc` - This is a native GNU Radio block that implements a fast FFT filter, with a choice of performing high pass, low pass or band pass filtering. The filter characteristics are determined through the choice of taps used. In this flowgraph, this filter functions as a low pass filter that blocks the unwanted frequencies beyond the baseband signal's bandwidth and effectively performs the function of a channel filter of a receiver.

- `gr_agc2_cc` - This is a native GNU Radio block that performs automatic gain control (AGC). Irrespective of the signal level, the signal coming in from the USRP is always applied with a constant gain and in order to maintain a relatively constant signal amplitude automatic gain controlling is necessary. This is done by sampling the output signal power and comparing it to a reference level. If the output signal level is too high, a negative signal is fed back to reduce the gain. Conversely, a positive signal is fed back to increase the gain if the output signal is low.

Preamble	Access Code	Length	Offset	Data	CRC	'\x55'

Figure 3.6. A typical GNU Radio frame

- `dsss_demodulator_cc` - This is the block where the actual demodulation of the baseband DS-BPSK signal takes place. It detects the presence of the signal through acquisition, tracks the signal, and demodulates the received signal into its constituent data bits.

  The input to the block is a stream of complex samples and the output is a stream of demodulated bits. We developed this block as a GNU Radio compatible block.

- `gr_correlate_access_code_bb` - This is a native GNU Radio block where frame synchronization takes place. Figure 3.6 represents the physical layer frame structure in GNU Radio. Immediately after the preamble, the frame consists of an access code which is a known vector consisting of bits `1` and `0`. For example, `"01010101011111000100"` is an access code that is used in some of the GNU Radio applications. This block uses correlation to determine the position of the access code in the demodulated stream of bits, coming from the previous block, and thus determines the start of a new frame.

- `gr_frame_sink_b` - This is a native GNU Radio block where the demodulated bits are assembled together to form the data packet and is sent to the application layer.

Ultimately, the application layer processes the data packets received from the physical layer.

3.4 Tracking

The carrier and code synchronization blocks of Figure 3.2 continuously track the offset carrier wave and code sequence in order to demodulate the received DS-BPSK signal.

Figure 3.7 represents the functional block diagram of the setup that implements these blocks. It consists of the following two sections:

- **Carrier Tracking Loop** - The carrier tracking loop is a phase-locked loop (PLL) that continuously tracks the carrier phase ϕ of $r(t)$ and generates a local quadrature carrier $e^{-j(2\pi\Delta f t+\widehat{\phi})}$, whose phase $\widehat{\phi}$ matches the carrier phase ϕ at all times. This loop consists of the *Carrier NCO* block, *Carrier Loop Filter* block and *Carrier Phase Discriminator* block.

- **Code Tracking Loop** - The code tracking loop is a PLL that continuously tracks the code phase T_d and generates the prompt sequence $c_p(t)$ of (3.4.1). that is matched in phase with the spreading-sequence $c(t)$ found in $r(t)$, at all times. $c_p(t)$ is represented The *prompt* sequence is a discrete sequence sampled at the same rate as $r(t)$ i.e. K samples per chip and is represented as,

$$c_p(t) = \sum_{m=0}^{\infty}\sum_{n=0}^{N-1}\sum_{k=0}^{K-1} c_n \Pi_{T_c}\left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d\right), \tag{3.4.1}$$

where $\widehat{T}_d$ is the code tracking loop's best estimate of T_d. Along with the prompt sequence, the code tracking loop generates two more sequences; the *early* sequence $c_e(t)$ and the *late* sequence $c_l(t)$. $c_e(t)$ is the replica code sequence, similar to $c_p(t)$, that has been advanced in time by half a spreading-sequence chip width $\frac{T_c}{2}$ and is represented as,

$$c_e(t) = c_p\left(t - \frac{T_c}{2}\right) = c\left(t - \widehat{T}_d - \frac{T_c}{2}\right). \tag{3.4.2}$$

Similary, $c_l(t)$ is the replica code sequence that has been delayed in time by half a spreading-sequence chip width of $\frac{T_c}{2}$ and is represented as,

$$c_l(t) = c_p\left(t + \frac{T_c}{2}\right) = c\left(t - \widehat{T}_d + \frac{T_c}{2}\right). \tag{3.4.3}$$

In Figure 3.7, the blocks, *Code NCO*, *Code Loop Filter*, and *Code Phase Discriminator* form the code tracking loop.

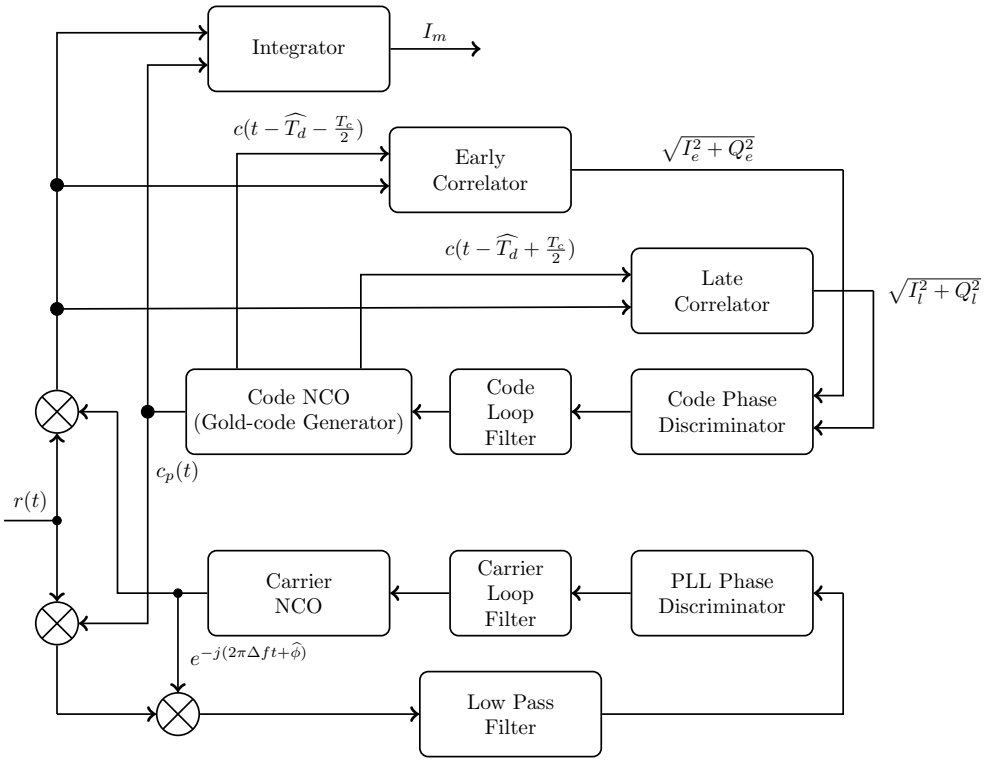

Figure 3.7. Functional block diagram of the tracking demodulator

- **Early-Prompt-Late Correlators** - This is a set of correlators that perform correlation between $r(t)$, after it has been corrected for carrier phase, and the early, prompt and late sequences respectively.

The tracking loops are initialized using the results from acquisition. The carrier NCO's initial frequency is set to Δf_{acq} and the prompt code's initial phase is set to T_d. In the carrier tracking loop, the despread signal $r_1(t)$ (3.2.10) is first multiplied with the local carrier and the product is fed into a low pass filter to produce $r_e(t)$, where

$$r_e(t) = d\,(t - T_d)\, e^{j\left(\phi - \widehat{\phi}\right)}. \tag{3.4.4}$$

The phase error "ϕ_e" between the received carrier and the local carrier is determined by the carrier phase discriminator. The discriminator uses the following equation to determine the phase error at sampling instance $t = mT_s$.

$$\phi_e[mT_s] = \tan^{-1}\left(\frac{\mathrm{Im}\,[r_e(mT_s)]}{\mathrm{Re}\,[r_e(mT_s)]}\right). \tag{3.4.5}$$

The phase error signal is passed on to the loop filter that produces a filtered value of the phase error that is used by the carrier NCO to adjust the phase of the local carrier signal such that, the local and the received carrier are in phase synchronization and

$$\phi_e(t) = \phi - \widehat{\phi} \approx 0. \tag{3.4.6}$$

In the code tracking loop, $r(t)$ is multiplied with the local carrier and the product is passed on to the early and late correlators. In the early correlator, $r_1(t)$ is multiplied with $c_e(t)$ and integrated over one code period (NT_c) to produce,

$$\begin{aligned} I_e &= d_m \cos\left(\phi - \widehat{\phi}\right) \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} \left[c_n \Pi \left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d \right) \right. \\ &\quad \left. c_n \Pi \left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d - \frac{T_c}{2} \right) \right], \end{aligned} \tag{3.4.7}$$

$$\begin{aligned} Q_e &= d_m \sin\left(\phi - \widehat{\phi}\right) \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} \left[c_n \Pi \left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d \right) \right. \\ &\quad \left. c_n \Pi \left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d - \frac{T_c}{2} \right) \right]. \end{aligned} \tag{3.4.8}$$

where, I_e and Q_e are the result of integration of the in-phase and quadrature components of $r_1(t)$. Similarly, the output of the late correlator is

$$\begin{aligned} I_l &= d_m \cos\left(\phi - \widehat{\phi}\right) \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} \left[c_n \Pi \left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d \right) \right. \\ &\quad \left. c_n \Pi \left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d + \frac{T_c}{2} \right) \right], \end{aligned} \tag{3.4.9}$$

$$\begin{aligned} Q_l &= d_m \sin\left(\phi - \widehat{\phi}\right) \sum_{n=0}^{N-1} \sum_{k=0}^{K-1} \left[c_n \Pi \left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d \right) \right. \\ &\quad \left. c_n \Pi \left(t - \widehat{T}_d - \frac{kT_c}{K} - nT_c - mT_d + \frac{T_c}{2} \right) \right]. \end{aligned} \tag{3.4.10}$$

The code phase discriminator uses the results from the integration to determine the phase difference between the prompt code sequence and the received signal's code sequence. The code phase error is found out as,

$$\psi_e = \frac{\sqrt{I_e^2 + Q_e^2} - \sqrt{I_l^2 + Q_l^2}}{\sqrt{I_e^2 + Q_e^2} + \sqrt{I_l^2 + Q_l^2}} \tag{3.4.11}$$

ψ_e is passed on to the code loop filter and the filtered output of the filter is used to adjust the phase of the code NCO. Under lock condition, the magnitudes of early correlation and late correlation are appromixately equal, i.e.

$$\sqrt{I_e^2 + Q_e^2} \approx \sqrt{I_l^2 + Q_l^2}, \tag{3.4.12}$$

If $0 < \psi_e < 1$, it means that the early sequence is *more* aligned in phase with the received code signal than the late sequence, and the prompt sequence's phase needs to be advanced. On the other hand, if $-1 < \psi_e < 0$, it means that the late sequence is *more* aligned in phase with the received code signal than the early sequence, and the prompt sequence's phase needs to be delayed.

3.5 Digital Phase Locked Loop Design

In this section we will describe the model of the PLL that we used to implement the tracking loops mentioned in the previous section. We will describe the design of this system using its closed loop transfer function and later develop difference equations that can be used to implement a software algorithm. Figure 3.8 shows the architecture of a second-order digital phase-locked loop (DPLL) system used to implement the tracking loops [20]. This system consists of the following components:

- Phase detector - The phase detector determines the difference between the input phase and the digitally controlled oscillator (DCO) phase. If $\phi_i[n]$ is the phase of

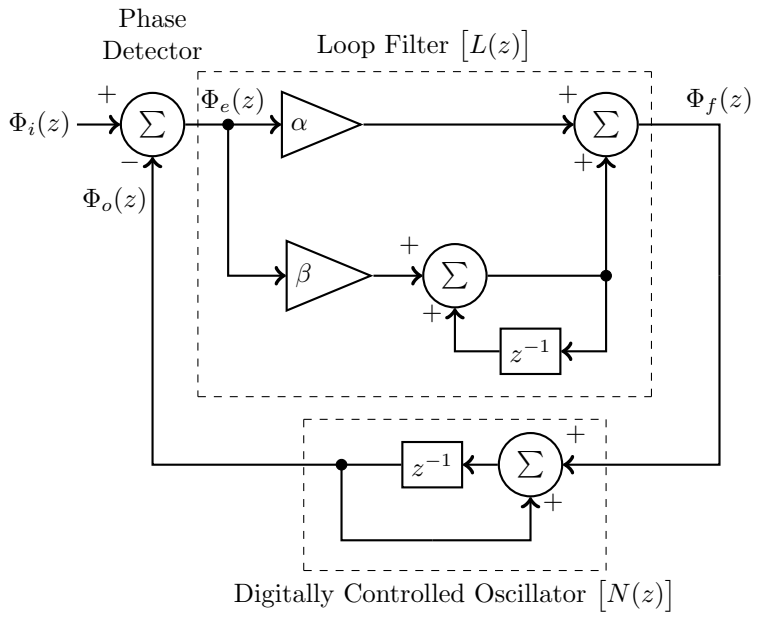

Figure 3.8. A second order digital phase locked Loop

the input sample at discrete time n and $\phi_o[n]$ is the output of the DCO the phase detector output $\phi_e[n]$ is,

$$\phi_e[n] = \phi_i[n] - \phi_o[n]. \tag{3.5.1}$$

- Loop Filter - A second-order DPLL consists of a proportional-integral (PI) filter with a proportional gain α and an integral gain β. With a PI loop filter, the PLL has a steady state error of zero for both phase and frequency steps, which makes this loop suitable for correcting phase and frequency offsets. The transfer function of this filter is,

$$L(z) = \frac{(\alpha+\beta)z - \alpha}{z-1}. \tag{3.5.2}$$

If $\phi_e[n]$ is the input to the PI filter at discrete time n, the resulting output of the filter can be described as

$$\phi_f[n] = \alpha\phi_e[n] + \sum_{m=0}^{n} \beta\phi_e[n-m], \tag{3.5.3}$$

where $\phi_f[n]$ is the output of the filter. The integral in the above equation results from the integral loop in the filter, where we accumulate the phase error from the

starting time of operation of the loop, i.e. 0, to the present time n. Using (3.5.1) we rewrite (3.5.3) as

$$\phi_f[n] = \alpha\{\phi_i[n] - \phi_o[n]\} + \sum_{m=0}^{n} \beta\{\phi_i[n] - \phi_o[n]\}. \tag{3.5.4}$$

- Delay element - z^{-1} is a delay element.

- Digitally Controlled Oscillator - The output of the filter is then passed on to the DCO. The DCO is an integrator that accumulates phase and produces an output that pushes the phase error at the output of the phase detector to zero. The transfer function of the DCO is,

$$N(z) = \frac{z}{z-1} \tag{3.5.5}$$

If $\phi_f[n]$ is the input to the DCO the following difference equation describes its output at discrete time n.

$$\phi_o[n] = \phi_o[n-1] + \phi_f[n]. \tag{3.5.6}$$

3.6 Closed Loop Transfer Function of a DPLL

With the block diagram and the transfer functions of the components available to us, the closed loop transfer function of the DPLL is then derived as,

$$H(z) = \frac{L(z)\, z^{-1} N(z)}{1 + L(z)\, z^{-1} N(z)}. \tag{3.6.1}$$

By substituting equations (3.5.2) and (3.5.5) into $H(z)$ and by reducing further, we get

$$H(z) = \frac{(\alpha+\beta)\left(z - \frac{\beta}{\alpha+\beta}\right)}{z^2 - 2\left(1 - \frac{\alpha+\beta}{2}\right)z + (1-\alpha)}. \tag{3.6.2}$$

In order to develop a software algorithm that functions as the DPLL represented by $H(z)$, we have to determine the values for α and β for which the loop is stable. In reference [16], the author outlines a procedure for designing analog phase locked loops. Deriving from

that, we convert the classical analog second-order loop transfer function $H_{\text{ref}}(s)$ (3.6.3) into an equivalent z-domain representation such that it has a similar form as $H(z)$ from (3.6.2).

$$H_{\text{ref}}(s) = \frac{2\zeta\omega_n s + {\omega_n}^2}{s^2 + 2\zeta\omega_n s + {\omega_n}^2}. \tag{3.6.3}$$

When bilinear transformation is applied to $H_{ref}(s)$, we get

$$H_{\text{ref}}(z) = H(s)\Big|_{s=\frac{2}{T_s}\frac{z-1}{z+1}} \tag{3.6.4}$$

where T_s is the time between successive samples. Upon substitution and rearranging, $H_{\text{ref}}(z)$ becomes

$$H_{\text{ref}}(z) = \frac{\frac{4\theta_n(\zeta+\theta_n)}{1+2\zeta\theta_n+{\theta_n}^2}\left(z - \frac{\zeta}{\zeta+\theta_n}\right)}{z^2 - 2\left(\frac{1+\theta_n^2}{1+2\zeta\theta_n+\theta_n^2}\right)z + \left(\frac{1-2\zeta\theta_n+\theta_n^2}{1-2\zeta\theta_n+\theta_n^2}\right)}, \tag{3.6.5}$$

where

$$\theta_n = \frac{\omega_n T_s}{2} = \frac{\omega_n}{\omega_s}\pi \tag{3.6.6}$$

θ_n is known as the normalized natural frequency. Comparing equations (3.6.2) and (3.6.5) and solving for α and β gives us, the loop gains in terms of the parameters of the analog transfer function, and

$$\alpha = \frac{4\zeta\theta_n}{1 + 2\zeta\theta_n + \theta_n^2}, \tag{3.6.7}$$

$$\beta = \frac{4\theta_n^2}{1 + 2\zeta\theta_n + \theta_n^2}. \tag{3.6.8}$$

3.7 DS-BPSK Signal Acquisition

Signal acquisition refers to the coarse synchronization of the code sequence present on the received signal and the locally generated sequence to within some fraction of the chip duration (T_c) of the code sequence. Once code acquistion has been accomplished, a code tracking loop is employed to achieve fine alignment. In addition to coarse code

synchronization, acquisition also coarsely determines the carrier frequency offset between the receiver and the transmitter [3].

As mentioned earlier, the received signal $r(t)$ suffers from delay (T_d) and carrier frequency offset (Δf). The tracking loops have a low bandwidth which requires them to be initialized such that the initial code phase of the locally generated code signal is within a fraction of the chip width and the frequency of the local carrier is within a few tens of hertz from the true carrier offset [2]. The goal of the acquisition is to perform cross-correlation of the received DS-BPSK signal and the code sequence used for spreading.

$$(s \star c)[n] = \sum_{m=0}^{N-1} s^*[m]\, c[n+m]. \tag{3.7.1}$$

If $s[n]$ and $c[n]$ are two sequences of length N, their cross-correlation is defined in (3.7.1) where $s^*[n]$ is the conjugate of $s[n]$. When $s(n)$ and $c(n)$ is the same, the above equation yields a maximum at $n = 0$ and this is the property we use for successful acquisition. During acquisition, if the code phase of the received signal matches the phase of the locally generated code sequence, the cross-correlation yeilds a maximum. We use the parallel search acquisition algorithm described in [2] to perform acquisition. This algorithm uses fast Fourier transform (FFT) to efficiently compute cross-correlation of the input signal and the locally generated code sequence.

The parallel acquisition algorithm involves performing circular cross-correlation between the sampled received signal and the sampled spreading sequence where the number of samples (`n`) chosen is a multiple of the product of the samples per chip (`samples_per_chip`) and the length (`pn_len`) of the spreading sequence. This ensures that we at least have one complete spreading sequence present in the received signal samples. The variables `s` and `c` in the pseudocode represent arrays of samples of signal and the spreading sequence.

The received signal is first multiplied with a locally generated exponential carrier such that the result of the multiplication yeilds a signal that removes the carrier from the received signal (line 24). Once the carrier is stripped, FFT based circular cross-correlation

is performed on `s` and `c` to yield the result of correlation `xcorr` (lines 26-30).

As the frequency of the carrier present on the received signal is unknown, the cross-correlation is repeated for various values of `f` in the range {`min_freq`, `max_freq`} incremented in steps of `delta_f`. Ultimately, the frequency `f` for which the cross-correlation is maximum and is greater than `THRESHOLD` is the estimated carrier frequency offset (`freq_offset`) and the index of the maximum is code phase offset (`code_offset`) in units of number of samples (lines 33-39).

```
1  # get 'n' signal and spreading sequence samples
2  n = k * samples_per_chip * pn_len
3  s = fetch_signal_samples(n)   # SIGNAL SAMPLES
4  c = fetch_code_samples(n) # CODE SAMPLES
5
6  # signal sample rate
7  fs    = chip_rate * samples_per_chip
8  ts    = 1.0/fs
9
10 # set frequency search range and resolution
11 min_freq  = MIN
12 max_freq  = MAX
13 delta_f   = fs / n
14
15 # function to generate exponential carrier of frequency 'x'
16 exp_carrier = lambda x: array([ 2*pi*x*ts*n for n
17                     in range(0,n-1) ])
18 curr_max = 0
19
20 fft_c = conjugate( fft(c) )
21 # find circular cross-correlation of signal_samples and code_samples
22 for f in range(min_freq, max_freq, delta_f):
23
24   # remove the carrier from the signal
25   s   = s * exp_carrier(f)
26   fft_s = fft(s)
27
28   # find cross-correlation
29   xcorr   = ifft(fft_s * fft_c)
30
31   # find cross-correlation maximum
32   _max, max_index = find_xcorr_max (xcorr)
33
34   if _max > THRESHOLD:
35     if _max > curr_max:
36       curr_max    = _max
37       freq_offset = f
38       code_offset = max_index
```

Listing 3.1. Parallel search algorithm for DSSS signal acquisition

This algorithm can be further improved for efficiency if we utilize the frequency shift

property of Fourier Transform [21]. This property states that if

$$f(t) \Leftrightarrow F(\omega) \tag{3.7.2}$$

then,

$$f(t)\, e^{j\omega_o t} \Leftrightarrow F(\omega - \omega_o)\,. \tag{3.7.3}$$

Using this, we calculate the FFT of the signal only once and rotate it in steps of one or more, around $f(0)$ component, to the same effect as multiplying a time domain signal with a complex carrier with a frequency that is a multiple of the frequency resolution of the FFT. Listing 3.2 shows the code that can replace lines 20-38 in Listing 3.1.

```
1  fft_c = conjugate( fft(c) )
2  fft_s   = fft(s)
3
4  shift_l = ceil(min_freq)/delta_f;
5  shift_r = ceil(max_freq)/delta_f;
6
7  # find circular cross-correlation of signal_samples and code_samples
8  for i in range(shift_l, shift_r, 1):
9
10   # find cross-correlation
11   xcorr   = ifft(rotate(fft_s, i) * fft_c)
12
13   # find cross-correlation maximum
14   _max, max_index = find_xcorr_max (xcorr)
15
16   if _max > THRESHOLD:
17     if _max > curr_max:
18       curr_max    = _max
19       freq_offset = i*delta_f;
20       code_offset = max_index
```

Listing 3.2. Improved parallel search algorithm for DSSS signal acquisition

Chapter 4

System Implementation

In this chapter, we discuss the implementation of the design we discussed in the Chapter 3. We begin by introducing to channel model block that we used for simualting a propagation medium. Later, we discuss the algorithms and software we developed to implement the various flowgraphs and signal processing blocks we discussed in Chapter 3. The results from testing and validation of the developed algorithms is presented at the end.

```
1 gr_channel_model(double noise_voltage,
2         double frequency_offset,
3         double epsilon,
4         const std::vector<gr_complex> &taps,
5         double noise_seed);
```

Listing 4.1. Channel model API

4.1 Channel Model

GNU Radio has a channel model block that can be used to simulate the effect of an additive white Gaussian noise (AWGN) channel on a propogating signal and the effect of frequency and timing offset caused due to Doppler effect and the unsynchronized clocks of the transmitter and the receiver. The channel model `gr_channel_model` exposes an API with which we can independently control the channel noise, frequency offset and the timing offset to simulate a received signal. `gr_channel_model` is implemented as a `C++`

class with the `API` shown in Listing 4.1. The arguments passed to this block are,

- `noise_voltage` (A_{n}) - This parameter is equivalent to the root mean square amplitude of noise in the signal and is calculated from the desired signal-to-noise ratio (S) of the simulated signal. (4.1.1) shows the relationship between S, the average signal power (P_{s}) and the average noise power (P_{n}).

$$S = 10 \log \frac{P_{\mathrm{s}}}{P_{\mathrm{n}}}. \tag{4.1.1}$$

  As the average power of the signal is a function of the square of the root mean square amplitude of the signal, (4.1.1) can be rewritten as

$$S = 20 \log \frac{A_{\mathrm{s}}}{A_{\mathrm{n}}}, \tag{4.1.2}$$

  where A_{s} is the root mean square amplitude of the simulated signal. Thus the `noise_voltage` parameter is evaluated as

$$A_{\mathrm{n}} = \frac{A_{\mathrm{s}}}{10^{\frac{S}{20}}}. \tag{4.1.3}$$

- `frequency_offset` (Δf_{n}) - This parameter is used to simulate the effect of a frequency offset (Δf) between the transmitted and received signals. Δf_{n} is Δf normalized to the sampling frequency (f_{s}) of the simulated signal.

$$\Delta f_{\mathrm{n}} = \frac{\Delta f}{f_{\mathrm{s}}}. \tag{4.1.4}$$

- `epsilon` (ϵ) - This parameter is used to simulate the effect of timing difference between the clocks of the transmitter and the receiver and signal delay. If t_{t} and t_{r} represent the pulse widths of the transmitter and receiver's clock, then

$$\epsilon = \frac{t_{\mathrm{t}}}{t_{\mathrm{r}}}. \tag{4.1.5}$$

4.2 Gold Codes

An important aspect of a spread-spectrum system design is to find a set of spreading codes or waveforms such that, multiple users can use the same frequency band without mutual interference. One important class of periodic sequences which provides a large set of sequences with good periodic cross-correlation property is the class of Gold sequences.

A Gold code set is generated through exclusive-or of two *preferred pair* maximum length sequences (m-sequences) [27] shifted by k bits, as shown in the following equation,

$$G_k[n] = m_1[n] \oplus m_2[n+k] \quad \text{for} \quad k \in \{0, 1, \ldots, 2^N - 1\}, \tag{4.2.1}$$

where m_1, and m_2 represent the preferred pair m-sequences of degree N, and G_k is the Gold code. Figure 4.1 represents a Gold code generator that uses two LFSR sequence generators and an ex-OR gate to generate Gold codes of length 63 . The two LFSR sequence generators shown in Figure 4.1 generate preferred pair m-sequences m_1, and m_2 that are represented by the following primitive polynomials of degree 6.

$$\begin{aligned} m_1 &= 1 + x^2 + x^3 + x^5 + x^6, \\ m_2 &= 1 + x + x^4 + x^5 + x^6. \end{aligned} \tag{4.2.2}$$

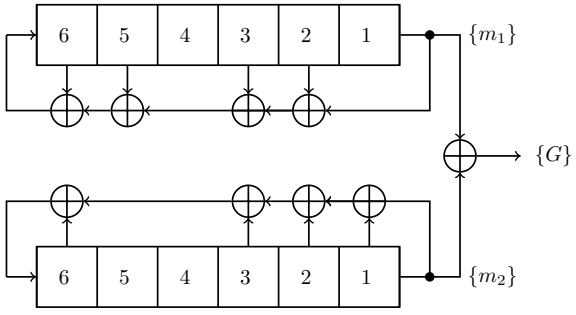

Figure 4.1. Gold code generator

One of the most important properties of Gold codes is their correlation results. In order to detect a weak signal in presence of other strong spread-spectrum signals, the

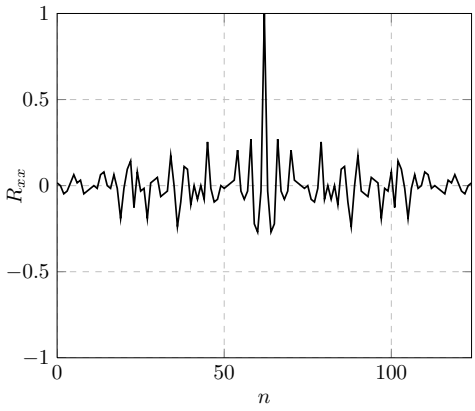

Figure 4.2. Autocorrelation of G_1

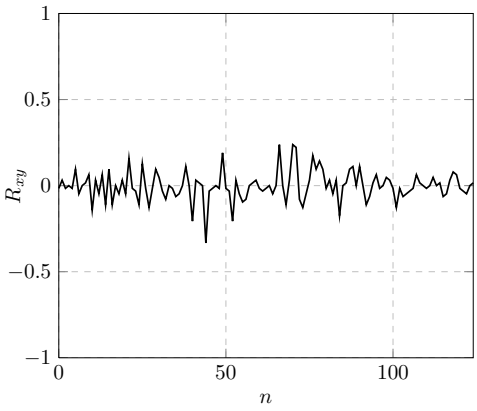

Figure 4.3. Cross correlation of G_1 with G_2

autocorrelation peak of the weak signal must be stronger than the cross correlation peaks from the strong signals. If the codes are orthogonal, the cross correlations will be zero. However, as Gold codes are not orthogonal but near orthogonal, their cross correlations are not zero but have small values and can be calculated using equations found in Table 5.4 of [27]. Figure 4.2 and 4.3 represent the correlation properites of Gold sequence G_1 with itself and another Gold code G_2. G_1 and G_2 were obtained using m-sequences from 4.2.2 with offsets k equal to 1 and 2 respectively.

In the discussions mentioned later, we will use two Gold codes of lengths 63 and 1023 to discuss the performance of the signal acquisition algorithm. The following table lists the primitive polynomials used to generate these two Gold codes.

Length	m_1	m_2
63	$1 + x^2 + x^3 + x^5 + x^6$	$1 + x + x^4 + x^5 + x^6$

Table 4.1. Primitive polynomials used

4.3 Testing the Acquistion and Carrier Synchronization blocks

To test and validate the algorithms developed for signal acquisition and tracking we generated test signals using the script `dsss_benchmark_loopback.py` that use the channel model block to simulate signal degradation during transmission. Figure 4.4 shows the position of the channel model block within the test flowgraph and Listing 4.2 shows an excerpt from the script. `dsss_benchmark_loopback.py` in Listing 4.2 instantiates an object each, of the `trasmit_path` flowgraph (line 21), `channel_model` block (line 22) and the `receive_path` flowgraph (line 23) and connects them together (line 25) to complete the flowgraph of Figure 4.4.

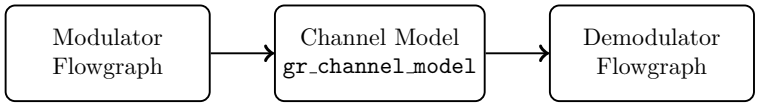

Figure 4.4. Test flowgraph with a channel model block

```
1  class my_top_block(gr.top_block):
2
3    def __init__(self, mod_class, demod_class, rx_callback, options):
4
5      gr.top_block.__init__(self)
6
7      # calculate noise_voltage
8      snr = snr_dB**(options.snr/10.0)
9      power_in_signal = abs(options.tx_amplitude)**2
10     noise_power = power_in_signal/snr
11     noise_voltage = math.sqrt(noise_power)
12
13     # frequency_offset
14     frequency_offset = options.frequency_offset
15
16     # samples_per_chip
17     if(options.samples_per_chip == None):
18       options.samples_per_chip = 4
19
20     # signal processing blocks
21     self.txpath = dsss_transmit_path.transmit_path(mod_class,
          options)
```

```
22      self.channel = gr.channel_model(noise_voltage, frequency_offset,
            1.0)
23      self.rxpath = dsss_receive_path.receive_path(demod_class,
           rx_callback, options)
24
25      self.connect(self.txpath, self.channel, self.rxpath)
26
27  def main():
28
29    global n_rcvd, n_right
30
31    n_rcvd = 0
32    n_right = 0
33
34    tb = my_top_block(dsss_modem.mod, dsss_modem.demod, rx_callback,
          options)
35    tb.start()
36
37    # generate and send packets
38    nbytes = int(1e6 * options.megabytes)
39    n = 0
40    pktno = 0
41    pkt_size = int(options.size)
42
43    while n < nbytes:
44      send_pkt(chr(0xFF))
45      n += pkt_size
46      pktno += 1
47      sys.stderr.write('.')
48
49    send_pkt(eof=True)
50    sys.stderr.write("\n");
51
52    tb.wait()
53
54  if __name__ == '__main__':
55    try:
56      main()
57    except KeyboardInterrupt:
58      pass
```

Listing 4.2. Excerpt from dsss_benchmark_loopback.py

4.3.1 Acquisition Tests

For the tests, we collected data by transmitting a sequence of the same character `0xFF` (Listing 4.2, lines 43-50) through the `transmit_path` flowgraph. As shown in Figure 3.4 the `transmit_path` flowgraph uses various blocks described in Section 3.3.1 to implement the DS-BPSK modulator. `class mod` shown in Listing 4.3 implements the DS-BPSK

modulator class in Python by importing the C++ classes we described in Section 3.3.1.

In lines 14-24, `class mod` instantiates these classes and connects them together (lines 26-29) to create the modulator flowgraph. `class mod` also sets the Gold code sequence used, it's length in chips and the number of samples per chip.

```
1  class mod(gr.hier_block2):
2
3    def __init__(self,samples_per_chip=_def_samples_per_chip,pn_id=
         _def_pn_id,verbose=_def_verbose,log=_def_log):
4      gr.hier_block2.__init__(self,"mod",gr.io_signature(1, 1, gr.
           sizeof_char),gr.io_signature(1, 1, gr.sizeof_gr_complex))
5
6      self._samples_per_chip = samples_per_chip
7      self._pn_id = pn_id
8      self._spreading = _def_spreading_factor
9      self._pn_len = _def_pn_len
10
11     self.excess_bw = _def_excess_bw
12     arity = 2
13
14     self.bytes2chunks = gr.packed_to_unpacked_bb(1,gr.GR_MSB_FIRST)
15     self.repeater     = dsss.repeat_bb(self._pn_len, 1)
16     self.pn_generator = gr.file_source(gr.sizeof_char, self._pn_file
           ,True)
17     self.xor          = gr.xor_bb()
18
19     bpsk_constellation    = [(1+0j), (-1+1.2246467991473532e-16j)]
20     self.chunks2symbols   = gr.chunks_to_symbols_bc(
           bpsk_constellation)
21
22     ntaps                   = 11 * self._samples_per_chip
23     self.rrc_taps           = gr.firdes.root_raised_cosine(self.
           _samples_per_chip, self._samples_per_chip, 1.0, self.
           excess_bw,ntaps)
24     self.rrc_interp_filter  = gr.interp_fir_filter_ccf(self.
           _samples_per_chip,self.rrc_taps)
25
26     self.connect(self, self.bytes2chunks, self.repeater)
27     self.connect(self.repeater, (self.xor,0))
28     self.connect(self.pn_generator, (self.xor,1))
29     self.connect(self.xor, self.chunks2symbols, self.
           rrc_interp_filter, self)
```

Listing 4.3. Excerpt from dsss_modem.py showing `class mod`

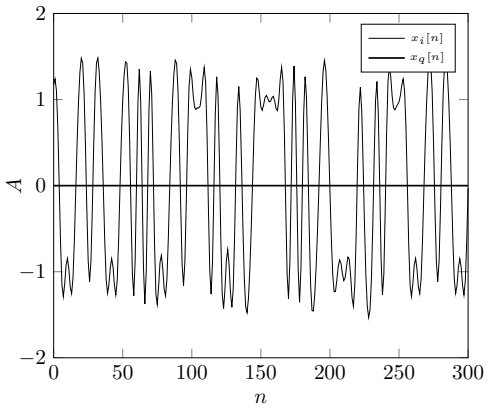

Figure 4.5. Modulator output from test 1

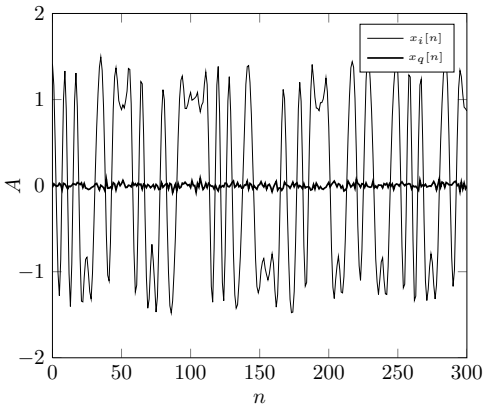

Figure 4.6. Channel output from test 1

```
class demod(gr.hier_block2):
  def __init__(self,
      samples_per_chip=_def_samples_per_chip,
      chip_rate=_def_chip_rate,
      pn_id=_def_pn_id,
      excess_bw=_def_excess_bw,
      verbose=False,
      log=False):

    gr.hier_block2.__init__(self, "demod",
        gr.io_signature(1, 1, gr.sizeof_gr_complex),
        gr.io_signature(1, 1, gr.sizeof_gr_complex))

    self._samples_per_chip = samples_per_chip
    self._chip_rate = chip_rate
    self._pn_id = pn_id
    self._excess_bw = excess_bw
    arity = 2

    self.agc = gr.agc2_cc(0.6e-1, 1e-3, 1, 1, 100)
    ntaps = 11 * samples_per_chip
    self.rrc_taps = gr.firdes.root_raised_cosine(
        1.0,                        # gain
        self._samples_per_chip,     # sampling rate
        1.0,                        # symbol rate
        self._excess_bw,            # excess bandwidth (roll-off
           factor)
        ntaps)

    self.rrc_filter=gr.interp_fir_filter_ccf(1, self.rrc_taps)
    # demodulator block performs acquisition and tracking
    self.demodulator = dsss.demodulator_cc(self._samples_per_chip,
       self._chip_rate, self._pn_id)

    # connect the blocks together to complete the flowgraph
    self.connect(self.agc, self.rrc_taps, self.demodulator, self)
```

Listing 4.4. Excerpt from dsss_modem.py showing `class demod`

Figure 4.5 shows the in-phase and the quadrature components of the signal output from end of the `transmit_path` flowgraph. The quadrature component of the signal is zero as we are performing BPSK modulation on the data bits. Figure 4.6 shows the signal output at the `channel_model` block. The effect of the channel is seen as a slight signal deterioration in both the in-phase and quadrature component.

The channel output is then fed into the `receive_path` flowgraph that consists of the demodulator class. Listing 4.4 shows the Python class that implements the demodulator flowgraph. `class demod` instantiates objects of the C++ classes we discussed in Section 3.3.2 and connects them together to complete the demodulator flowgraph.

We developed `dsss_demodulator_cc` as the C++ class that implements acquisition and tracking. It instantiates the class `dsss_acquisition` and uses its method to perform signal acquistion as discussed in Section 3.7. It also instantiates classes `code_nco` and `carr_nco` that generate the local code sequence and local carrier signal. Listing 4.6 shows the prototype of these classes and their methods.

```
1  class dsss_acquisition {
2
3    public:
4      dsss_acquisition(int samples_per_chip, double chip_rate,
5                       int pn_id, unsigned int nsamples);
6      ~dsss_acquisition();
7
8      bool perform_acquisition(gr_complex *signal, int start,
9                               double fft_resolution);
10     double  get_acq_freq();
11     unsigned int get_acq_code();
12   private:
13     void  sample_pn_seq(gr_complex *, gr_complex *);
14     void  fft_pn_seq(gr_complex *, gr_complex *);
15     void  find_max(gr_complex *,cross_corr *);
16     bool  compare_corr_bins(cross_corr *, int, double, acq_res *);
17     void  init();
18 };
```

Listing 4.5. Definition of `class dsss_acquisition`

With these settings, a complete Gold code sequence of length 63 is represented with 252 samples and thus we require at least 252 samples of the received signal to perform

acquisition. As the signal in our first test does not have a frequency offset, the acquisition algorithm performs cross-correlation on the signal sample set such that it searches only between {-delta_f, delta_f}, where delta_f is the frequency resolution of the FFT and is equal to

$$\texttt{delta\_f} = \frac{100000 * 4}{252} = 1587.3 \text{ Hz} \tag{4.3.1}$$

```
class dsss_demodulator_cc : public gr_block
{
  private:
    dsss_demodulator_cc(int samples_per_chip,
                        double chip_rate, int pn_id);
    friend dsss_demodulator_cc_sptr
    dsss_make_demodulator_cc(int samples_per_chip,
                             double chip_rate, int pn_id);

    class code_nco *d_code_nco;
    class carr_nco *d_carr_nco;

  public:
    int general_work(int noutput_items,
         gr_vector_int &ninput_items,
         gr_vector_const_void_star &input_items,
         gr_vector_void_star &output_items);
    void forecast(int noutput_items,
       gr_vector_int &ninput_items_required);
    ~dsss_demodulator_cc();
};
```

Listing 4.6. Definition of class dsss_modulator_cc

noise_voltage	0.1
frequency_offset	0
epsilon	1.0

Table 4.2. Settings for channel_model

SNR	30 dB
Δf	0
T_d	0

Table 4.3. Equivalent effect in signal

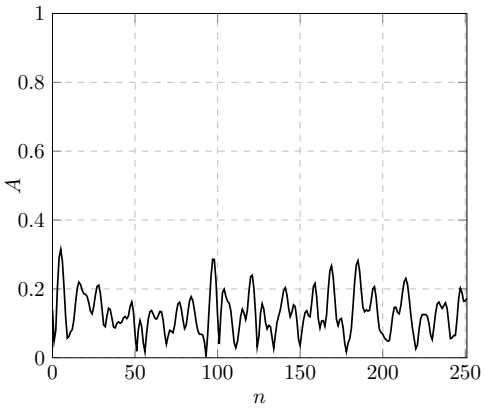

Figure 4.7. Acquisition results with f_L=-1587.3 Hz

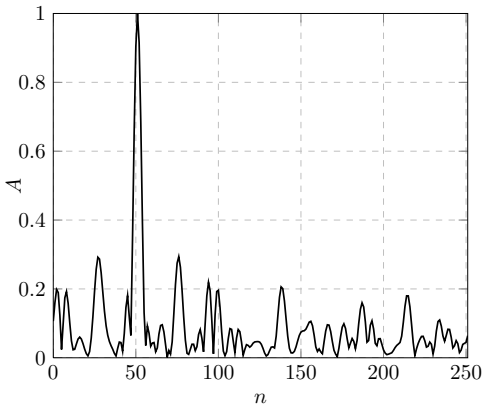

Figure 4.8. Acquisition results with f_L=0 Hz

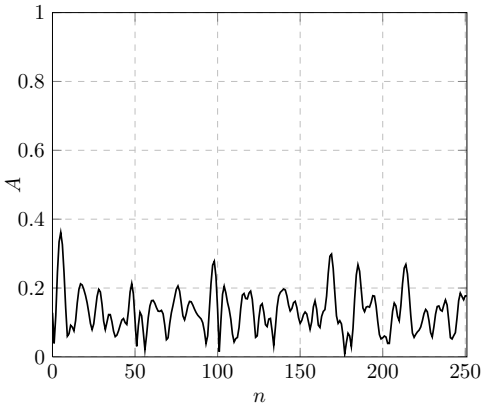

Figure 4.9. Acquisition results with f_L=1587.3 Hz

Figures 4.7 to 4.9 show the acquisition results from this test when local carrier signal frequencies of -1587.3 Hz, 0 Hz and 1587.3 Hz were used. As the signal didn't have any frequency offset, a distinct correlation peak is observed only in the 0 Hz set.

In our second test, we changed the frequency offset value in the channel model to 1000 Hz. Figure 4.10 to 4.13 shows the acquisition results when 252 samples were used and the frequency search was performed between {-3174.6 Hz, 1587.3 Hz} in steps of 1587.3 Hz. The highest peak is observed in the results from using f_{L} equal to -1587.3 Hz.

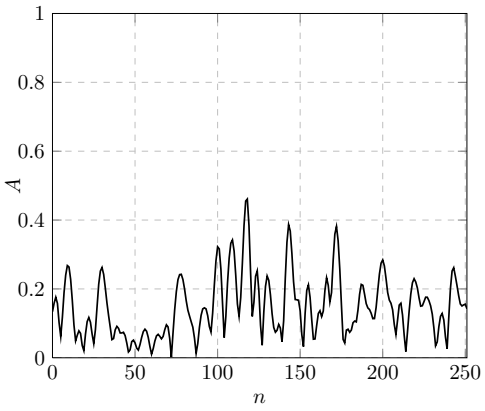

Figure 4.10. Acquisition results with f_L=-3174.6 Hz

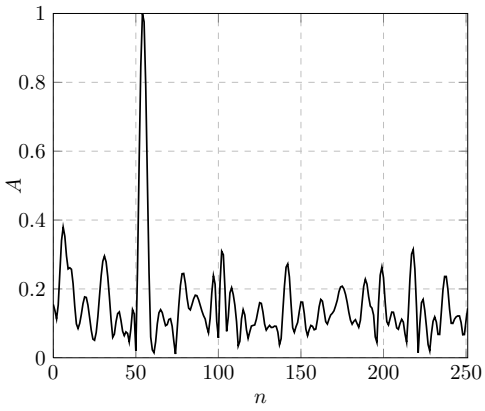

Figure 4.11. Acquisition results with f_L=-1587.3 Hz

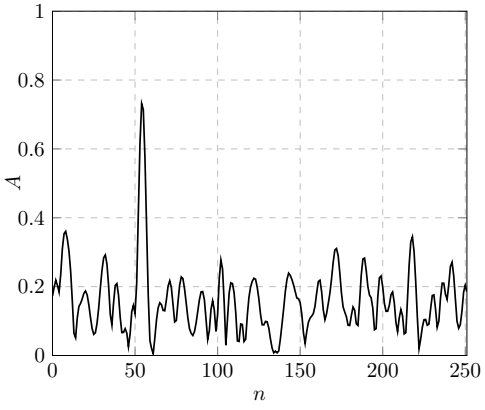

Figure 4.12. Acquisition results with f_L=0 Hz

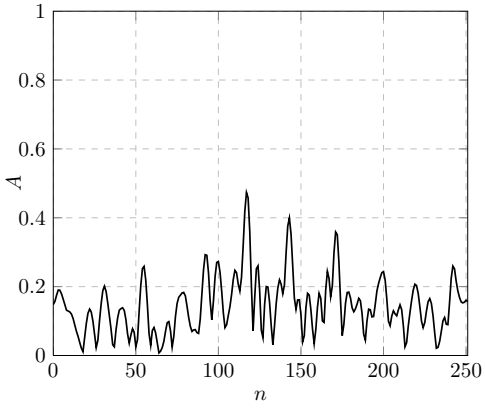

Figure 4.13. Acquisition results with f_L=1587.3 Hz

Next, we changed the noise voltage to 3.162 to introduce more noise into the signal such that the effective SNR was -10 dB. Figure 4.14 shows the acquisition result when 252 samples were used at f_c equal to -1587.3 Hz. Clearly, as there is no distinct peak in the signal we cannot determine the code and frequency offsets from this result.

From 3.7.1 ,we see that by increasing the number of samples (N) used for cross-correlation we increase the magnitude of cross-correlation. Thus, in order to determine the offsets we increased the number of samples to include at least 4 consecutive Gold

code sequences so that the noise can be averaged over a longer period of time resulting in higher correlation values. Figure 4.15 shows the result of acquisition when 1008 samples were used.

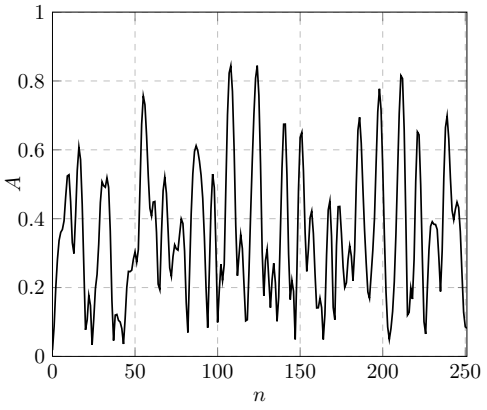

Figure 4.14. Acquisition result with 252 samples

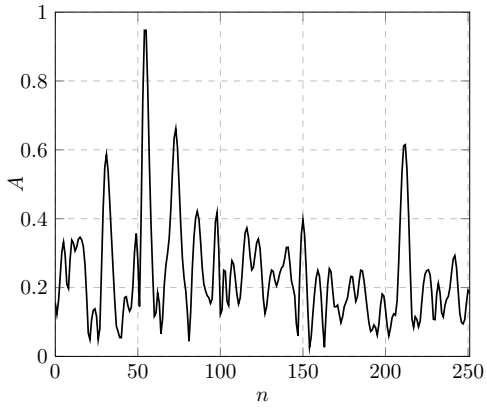

Figure 4.15. Acquisition result with 1008 samples

4.3.2 Carrier Tracking Tests

Once the code offset (T_{acq}) and carrier offset (Δf_{acq}) are determined, the information is passed on to the tracking algorithm of `dsss_demodulator_cc`. As described in Section 3.4 and in Figure 3.7, the tracking demodulator implements a carrier tracking loop and a code tracking loop. The carrier tracking loop algorithm shown in Listing 4.7 is based on the discussion of Section 3.5 and [14]. T_{acq} is used to adjust the code numerically controlled oscillator (NCO) such that the next code sequence it produces is aligned in phase with the code present on the received signal. Δf_{acq} is used to select an operating bandwidth for the carrier tracking loop. The frequency range {`d_min_freq, d_max_freq`} is chosen such that Δf_{acq} lies within.

Inside the for-loop, the state of the tracking algorithm changes with every input sample. Input sample `in[i]` is first multiplied with the prompt code sample (3.2.10) to obtain the despread sample `dptr` (line 18). Next, we find the carrier NCO output `nco_out` using the current phase output value `phi_o` and multiply it with `dptr` to obtain the code

tracking loop's output `pll_out` (3.2.11).

```
const gr_complex *in = (gr_complex *)input_items[0];
gr_complex *prompt_code_ptr;
float frequency_offset;
float carrier_frequency;

/* use the result from acquisition */
code_nco->adjust_code_nco(d_code_offset);
phi_o = get_freq_offset();
phi_f = (d_max_freq + d_min_freq) / 2.0;

phi_e = 0;


for(int i=0; i<noutput_items; i++) {

  if(d_state==TRACK) {

    gr_complex dptr = in[i] *  get_prompt_code_sample();
    nco_out = gr_expj(-phi_o);
    pll_out = dptr * nco_out;
    dll_out = in[i] *nco_out;

    phi_e = pll.imag() * pll.real();

    /* Limit phi_e between {-1, 1} */

    if (phi_e > 1)
      phi_e = 1;
    else if (phi_e < -1)
      phi_e = -1;

    phi_f = phi_f + beta * phi_e;
    phi_o = phi_o + (alpha * phi_e) + phi_f;

    /* limit phi_f_b between {min_freq, max_freq} */

    if(phi_f > d_max_freq)
      phi_f = d_max_freq;
    if(phi_f < d_min_freq)
      phi_f = d_min_freq;

    /* Limit phi_o between {-2*PI, 2*PI} */

    while(phi_o > M_TWOPI)
      phi_o -= M_TWOPI;

    while(phi_o < -M_TWOPI)
      phi_o += M_TWOPI;


  }

}
```

Listing 4.7. Carrier tracking algorithm

α	0.1
`max_freq`	0.0187
`min_freq`	-0.0187

Table 4.4. Settings for test 3

In line 20, we multiply `nco_out` with input sample `in[i]` to obtain `dll_out` which becomes the input to the code tracking loop. From `pll_out` we evaluate the carrier phase error `phi_e` in line 23. Using `phi_e`, we estimate the input (`phi_f`) and the output (`phi_o`) of the loop filter, where `phi_f` forms $\phi_{\mathrm{f}}[n]$ of (3.5.3) and `phi_o` is $\phi_{\mathrm{o}}[n]$ of (3.5.4).

In our first test, we generated a signal using the channel model the same way as in Section 4.3.1. Figure 4.16 and 4.17 show a sample of the inphase and quadrature components of this signal with a simulated SNR of 20 dB and a frequency offset of 1000 Hz. From acquisition results, this signal had a code offset value of 114 and frequency offset value of 1190.48 Hz.

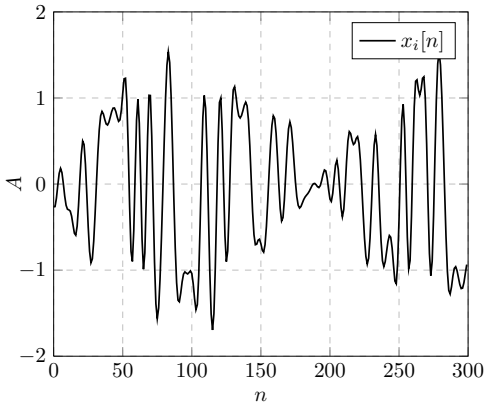

Figure 4.16. Inphase component of received signal for Test 3 with Δf=1000 Hz and SNR=20 dB

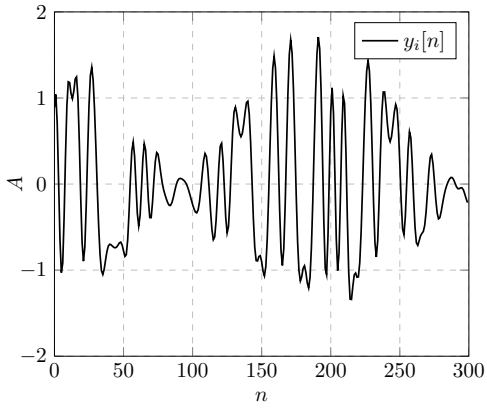

Figure 4.17. Quadrature component of received signal for Test 3 with Δf=1000 Hz and SNR=20 dB

Figure 4.18 and 4.19 show the carrier tracking loop's output when settings from Table 4.4 were used. Figure 4.20 plots the phase error values for the first 300 samples processed

by the tracking loop. From these figures we see that, as the phase error stabilizes around zero, the quadrature component of the carrier tracking loop output settles down around zero in accordance with (3.2.12).

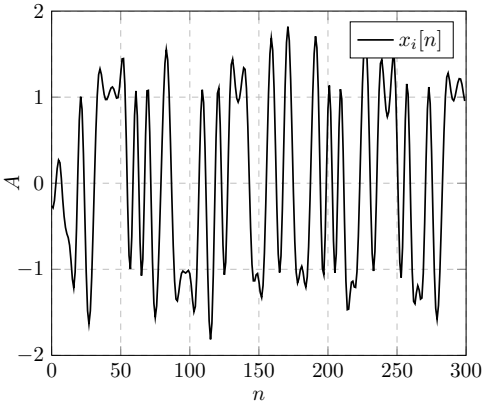

Figure 4.18. Inphase component of the output signal from tracking loop

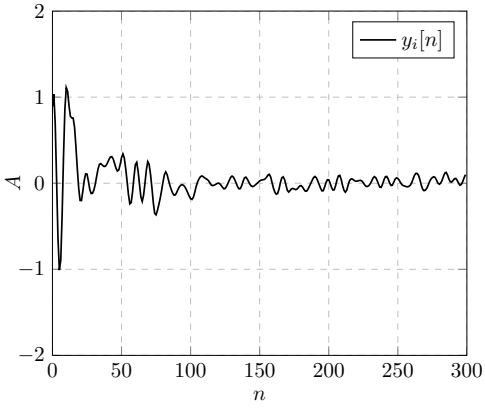

Figure 4.19. Quadrature component of the output signal from tracking loop

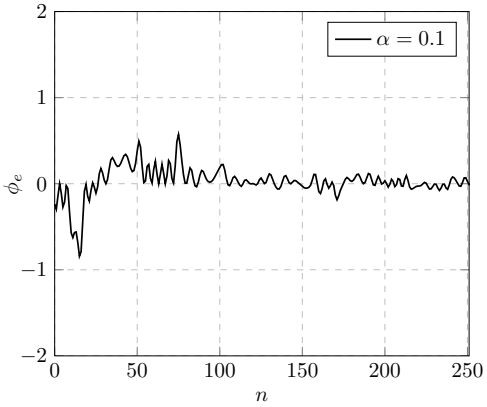

Figure 4.20. Phase error plot for test 3

Chapter 5

Conclusions and Future Research

In this thesis, we evaluated an SDR design to prototype a DS-BPSK transceiver. In this chapter, we summarize the major results obtained through tests and simulations and propse further research directions.

First, the achitecture of GNU Radio and USRP was discussed. We thoroughly understood the design procedure and proposed the design for a GNU Radio compliant transmitter flowgraph and a receiver flowgraph along with discussing the blocks that are needed to implement these flowgraphs. We also presented a detailed analysis of DSSS and discussed the algorithms required to detect and demodulate a DSSS signal.

We then discussed in detail the algorithms and code we developed and performed tests to validate our software. We showed that our developed software radio blocks for acquisition and carrier tracking function well for high signal to noise ratio (SNR) signals. The software we developed is highly configurable, extendable and maintains compatibility with GNU Radio design.

From this study, we believe that software radio and GNU Radio in particular is suitable for developing prototypes for validating protocol design for ad hoc networks. Although the framework we developed requires development of a code tracking loop, we believe in its current state it provides a good base to develop DSSS based transceivers for ad hoc networking.

BIBLIOGRAPHY

Bibliography

[1] Unknown Author. Direct Sequence Spread Spectrum (DSSS) Modem Reference Design [Online]. Technical report, Altera Corporation, San Jose, CA, Sep 2001. Available: `http://www.altera.com/literature/fs/fs14_dsss.pdf`.

[2] Kai Borre, Dennis M. Akos, Nicolaj Bertelsen, and Peter Rinderand Soren Holdt Jensen. *A Software-Defined GPS and Galileo Receiver - A Single-Freqeuncy Approach*, chapter 6-7, pages 75–125. Birkhäuser, Boston, MA, 1st edition, Mar 2007.

[3] K.K. Chawla and D.V. Sarwate. Parallel acquisition of PN sequences in DS/SS systems. *IEEE Transactions on Communications*, 42(5):2155 –2164, May 1994.

[4] John N. Daigle. private communication, Apr 2008.

[5] Mark Ettus. Transceiver Daughterboards for the USRP Software Radio System [Online]. Available: `www.ettus.com/downloads/ettus_ds_transceiver_dbrds_v6c.pdf`.

[6] Free Software Foundation. gr_file_source [Online]. Available: `http://gnuradio.org/doc/doxygen/classgr__file__source.html`.

[7] Free Software Foundation. gr_packed_to_unpacked_bb [Online]. Available: `http://gnuradio.org/doc/doxygen/classgr__packed__to__unpacked__bb.html`.

[8] Free Software Foundation. gr_xor_to_bb [Online]. Available: `http://gnuradio.org/doc/doxygen/classgr__xor__bb.html`.

[9] Free Software Foundation. OSSIE - SCA-Based Open Source Software Defined Radio [Online]. Available: `http://ossie.wireless.vt.edu/`.

[10] Free Software Foundation. tunnel.py [Online]. Available: http://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[11] Free Software Foundation. Welcome to GNU Radio [Online]. Available: http://gnuradio.org/redmine/projects/gnuradio/wiki.

[12] Free Software Foundation. Welcome to SWIG [Online]. Available: http://www.swig.org/.

[13] Python Software Foundation. Callback Functions [Online]. Available: http://docs.python.org/release/2.5.2/lib/ctypes-callback-functions.html.

[14] Eric Hagemann. The Costas Loop [Online]. Technical report, Company Unknown. Available: [www.dsp-book.narod.ru/costas/DSP010419F1.pdf], [www.dsp-book.narod.ru/costas/DSP010419F1.pdf], [www.dsp-book.narod.ru/costas/DSP010628F1.pdf], [www.dsp-book.narod.ru/costas/DSP010315F1.pdf].

[15] Firas Abbas Hamza. The USRP Under 1.5X Magnifying Lens! [Online]. Technical report, Available: http://microembedded.googlecode.com/files/USRP_Documentation.pdf, Jun 2008.

[16] F. Harris and B. Farhang-Boroujeny. On the stability of DSP based P-I phase-locked loops containing matched filter delays. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 958 –962, Nov 2011.

[17] Fredric J. Harris. *Multirate Signal Processing for Communication Systems*, chapter 4, pages 89–90. Prentice Hall, Upper Saddle River, NJ, 1st edition, May 2004.

[18] Rodger H. Hosking. *Software Defined Radio Handbook*, pages 7–10. Pentek, Inc, Upper Saddle River, NJ, 8th edition, Jan 2010.

[19] Peter B. Kenington. *RF and Baseband Techniques for Software Defined Radio*, chapter 2, pages 25–33. Artech House, Norwood, MA, 1st edition, Jun 2005.

[20] Wen Li and Jason Meiners. Introduction to Phase-Locked Loop System Modeling [Online]. *Analog Applications Journal*, pages 5–10, May 2000. Available: `www.ti.com/lit/an/slyt169/slyt169.pdf`.

[21] Richard G. Lyons. *Understanding Digital Signal Processing*, chapter 8, pages 335–358. Prentice Hall, Upper Saddle River, NJ, 2nd edition, Mar 2004.

[22] J. Mitola. The Software Radio Architecture. *Communications Magazine, IEEE*, 33(5):26–38, May 1995.

[23] Roger L. Peterson, Rodger E. Ziemer, and David E. Borth. *Introduction to Spread Spectrum Communications*, chapter 2, pages 47–48. Prentice Hall, Upper Saddle River, NJ, Apr 1995.

[24] Roger L. Peterson, Rodger E. Ziemer, and David E. Borth. *Introduction to Spread Spectrum Communications*, chapter 2, pages 64–75. Prentice Hall, Upper Saddle River, NJ, Apr 1995.

[25] Don Torrieri. *Principles of Spread-Spectrum Communication Systems*, chapter 2,3, pages 55,129. Springer Inc, New York, NY, 1st edition, Jul 2005.

[26] James Bao-Yen Tsui. *Fundamentals of Global Positioning System Receivers - A Software Approach*, chapter 7-8, pages 129–171. Wiley Interscience, Hoboken, NJ, 2nd edition, Mar 2005.

[27] James Bao-Yen Tsui. *Fundamentals of Global Positioning System Receivers - A Software Approach*, chapter 5, pages 68–79. Wiley Interscience, Hoboken, NJ, 2nd edition, Mar 2005.

[28] Unknown Author. Software Defined Radio Measurement Solutions [Online]. Technical report, Agilent Technologies, Santa Clara, CA, Jul 2007. Available: `http://www.home.agilent.com/agilent/application.jspx?nid=-34052.0.00&lc=eng&cc=US`.

VITA

Mir Murtuza Ali received his B.E in Electrical and Electronics Engineering at Osmania University, India in 2004. He joined the Masters in Engineernig Science program in Electrical Engineering at University of Mississippi in 2006. During his Masters program he worked as a research assistant at National Center for Physical Acoustics and Center for Wireless Communications. Currently, he is working as a software engineer at Barracuda Networks in Campbell, CA. Mir's technical interests include digital signal processing, computer networking and web development. When not pursuing these interests, Mir can often be found reading and has a particular fascination for US history and politics.