

Exploration into the Use of a Software Defined Radio
as a Low-Cost Radar Front-End

Andrew Michael Monk

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of
Master of Science

David G. Long, Chair
Philip Lundrigan
Willie Harrison

Department of Electrical and Computer Engineering
Brigham Young University

Copyright © 2020 Andrew Michael Monk
All Rights Reserved

ABSTRACT

Exploration into the Use of a Software Defined Radio as a Low-Cost Radar Front-End

Andrew Michael Monk

Department of Electrical and Computer Engineering, BYU
Master of Science

Inspection methods for satellites post-launch are currently expensive and/or dangerous. To address this, BYU, in conjunction with NASA, is designing a series of small satellites called CubeSATS. These small satellites are designed to be launched from a satellite and to visually inspect the launching body. The current satellite revision passively tumbles through space and is appropriately named the passive inspection cube satellite (PICS). The next revision actively maintains translation and rotation relative to the launching satellite and is named the translation, rotation inspection cube satellite (TRICS). One of the necessary sensors aboard this next revision is the means to detect distance. This work explores the feasibility of using a software defined radio as a small, low-cost front end for a ranging radar to fulfill this need.

For this work, the LimeSDR-Mini is selected due to its low-cost, small form factor, full duplex operation, and open-source hardware/software. Additionally, due to the the channel characteristics of space, the linear frequency modulated continuous-wave (LFMCW) radar is selected as the radar architecture due to its ranging capabilities and simplicity. The LFMCW radar theory and simulation are presented.

Two programming methods for the LimeSDR-Mini are considered: GNU Radio Companion and the pyLMS7002Soapy API. GNU Radio Companion is used for initial exploration of the LimeSDR-Mini and confirms its data streaming (RX and TX) and full duplex capabilities. The pyLMS7002Soapy API demonstrates further refined control over the LimeSDR-Mini while providing platform independence and deployability.

This work concludes that the LimeSDR-Mini is capable of acting as the front end for a ranging radar aboard a small satellite provided the pyLMS7002Soapy API is used for configuration and control. GNU Radio Companion is not recommended as a programming platform for the LimeSDR-Mini and the pyLMS7002Soapy API requires further research to fine tune the SDR's performance.

Keywords: software defined radio (SDR), radar, linear frequency modulated continuous wave (LFMCW), application product interface (API)

ACKNOWLEDGMENTS

I would like to thank first and foremost my wife, Kate. Without her, this work would not be a thing and I would likely not even be pursuing a graduate degree. Also, I would like to thank my children, Andie and Killian, who reminded me over and over to take breaks and enjoy what really matters. Lastly, I would like to thank Dr. Long for the time and endless words of encouragement that he used to keep me going when the road seemed too *long*.

TABLE OF CONTENTS

TABLE OF CONTENTS	iv
LIST OF TABLES	vi
LIST OF FIGURES	vii
Chapter 1 Introduction and Background	1
1.1 Digital Signal Processing	2
1.2 Hardware	5
1.3 Takeaway	8
Chapter 2 LFMCW Radar and Simulation	10
2.1 Simulation Requirements	10
2.2 Signal Analysis and Generation	11
2.3 IQ Modulation	12
2.4 Range Detection	17
2.5 Nonidealities	23
2.5.1 Maximum Detectable Range	23
2.5.2 Channel Noise	24
2.5.3 Antenna Crosstalk	25
2.6 Simulation Contribution	26
Chapter 3 GNU Radio and gr-radar	29
3.1 GNU Radio Companion	29
3.1.1 Toolboxes and Blocks	30
3.1.2 Toolbar Functions	34
3.2 Flow Graph Topology	36
3.3 Pros and Cons	37
3.4 GNU Radio Contribution	39
Chapter 4 LimeSuite GUI and Python3 API	41
4.1 LimeSDR-Mini Hardware	41
4.2 Programming the LMS7002M FPRF	44
4.2.1 LimeSuite GUI	45
4.2.2 LimeSDR Python3 API	47
4.3 LimeSuite GUI and API Contribution	50
Chapter 5 Conclusion	58
5.1 Work Summary	58
5.2 Conclusion	59
5.3 Future Work	59

REFERENCES	61
Appendix A SDR Comparison and Selection	62
A.1 Considerations and Selection	62
A.2 SDR Comparisons	63
Appendix B LFM CW Simulation MatLab Script	65
B.1 Main Code	65
B.2 Helper Function	71
Appendix C GNU Radio and gr-radar Installation	73
C.1 Preface	73
C.2 Installation	73
C.2.1 Basic Installation	73
C.2.2 Additional Steps for VirtualBox	75
C.2.3 Installation Verification	76
C.3 Resources	77
Appendix D GNU Radio Exploration through Flow Graphs	79
D.1 Flow Graphs	79
Appendix E PiSDR Installation	84
E.1 Required Hardware	84
E.2 PiSDR Installation	84
E.3 pyLMS7002Soapy API Installation	86
E.4 Verification	86
E.5 Recommendations	87
E.5.1 ipython3	87
E.5.2 PuTTY	87
E.5.3 GUI Applications through SSH	88
E.5.4 Raspberry Pi IP Emailer	89
E.6 Resources	89
Appendix F LimeSuite GUI and API	90
F.1 LimeSuite GUI Tabs	90
F.2 API Libraries	91
F.3 Unused API Libraries	93
F.4 Other Useful Acronyms	94
Appendix G Generate LFM CW Waveform MatLab Script	95
Appendix H Example API Script	99

LIST OF TABLES

A.1	A comparison of six different software defined radios. Green squares indicate best or good, uncolored squares are acceptable, and red squares indicate bad or outside project restrictions.	64
-----	---	----

LIST OF FIGURES

2.1	Convolution example shown in frequency domain. Input is a 500 MHz signal, the local oscillator frequency is 3 kHz, and the output signal is the input signal centered around each local oscillator delta.	13
2.2	Block diagram of a general IQ modulator with two input data streams output to a single antenna.	14
2.3	Block diagram of general IQ demodulator with input antenna and resultant data streams.	15
2.4	Example FFTs of sinusoidal waves. The small spikes below and above the deltas are artifacts from actualizing the FFTs with a finite number of samples. Left: FFT of $\cos(2\pi f)$; Right: FFT of $j\sin(2\pi f)$	16
2.5	Example of a simple demodulation using the output signal from Fig. 2.1 with the same local oscillator frequency of 3 kHz.	17
2.6	General DSP process for LFM CW radar range detection.	18
2.7	Example doppler time plot of arbitrary transmit and receive (LFM CW) signals with frequency and time deltas marked (δ and τ respectively).	19
2.8	Example complex input signals (real portions) over one second. Left: complex sinusoid with a frequency of 10 Hz. Right: complex sinusoid with a frequency of 7 Hz.	20
2.9	Resulting signal (real portion) of the conjugate multiplication of the signals in Fig. 2.8.	21
2.10	Example of constant wave resulting from transmitted and received signal conjugate multiplication. The high frequency noise (located just right of $7 \mu s$) is a result of the chirp restart.	22
2.11	Example PSD of a given signal generated from the conjugate multiplication between transmit and receive signals.	23
2.12	Example conjugate product of transmit receive pair with added uniform channel noise fifty times greater than transmit signal.	25
2.13	Example PSD of a given signal generated from the complex product between transmit and receive signals with uniform white channel noise at fifty times the transmit amplitude.	26
2.14	Comparison of non-cooperative (left) and cooperative (right) radar topologies. The color of the arrow indicates the carrier frequency, while the color of the box around TX or RX indicates the frequency the radio is tuned to transmit/receive.	27
2.15	Example conjugate product with antenna crosstalk at a 4:1 crosstalk to signal ratio. . .	27
2.16	Example PSD of the same data from Fig. 2.15.	28
3.1	Full window view of GNU Radio Companion with simple flow graph example. Key areas of the GUI are outlined and labeled.	30
3.2	Visualization of the transmit block from the LimeSuite toolbox. Here it is depicted without any connections and none of the parameters, save the default ones, are provided. As such, the block is throwing an error, which is visible from the red title. . . .	31
3.3	The data types and their corresponding colors used within GNU Radio.	32
3.4	Properties dialog for the LimeSuite transmit block. The options tabs include general, CH_A, CH_B, and advanced. The documentation tabs include documentation and generated code.	33

3.5	View of special GNU Radio toolbar functions found at the top of the main program window.	34
3.6	Example flow graph of an FSK radar provided by the gr-radar toolbox with the simulated transmit and receive pair replaced with the LimeSuite transmit and receive blocks, respectively.	40
4.1	High level block diagram of the LimeSDR-Mini hardware. The notable components are the FPGA (Intel MAX 10) and RF transceiver (LMS7002M).	42
4.2	Block diagram of MyriadRF LMS7002M FPRF.	51
4.3	Focused view of the MyriadRF LMS7002M receive block diagram. Data flow is from left to right, from antenna to ADC output.	52
4.4	Focused view of the MyriadRF LMS7002M transmit block diagram. Data flow is from right to left, from DAC input to antenna.	52
4.5	Screenshot of the LimeSuite GUI tabs with half of the tabs wrapped around for easier viewing. Tabs outlined with a yellow rectangle correspond directly to an API library and the orange tabs loosely correspond to one or more API libraries.	52
4.6	Block diagram of the API include structure. Each of the rectangular libraries within the dotted outline are individually imported by the LMS7002 library. The yellow libraries directly correspond to a LimeSuite GUI tab and the orange libraries loosely correspond to a LimeSuite GUI tab.	53
4.7	View of the LimeSuite GUI opened to the calibrations tab (default upon launch). The bottom console currently displays information about the newly connected SDR.	54
4.8	LimeSuite GUI radio selection dialog to connect/disconnect an SDR.	54
4.9	Subsection of the SXR (SXT has similar options) tab indicating the carrier frequency selection textbox and tuning button.	55
4.10	LimeSuite GUI FPGA controls dialog window used to feed data into the transmit portion of the SDR.	55
4.11	LimeSuite GUI FFT viewer currently running showing transient, constellation, and FFT representations of the received data.	56
4.12	Example terminal for a successful SDR connection using the pyLMS7002Soapy API.	56
4.13	Example of traversing the API using the ipython3 tab completion feature to show all available functions and classes from the LMS7002 library.	57
D.1	Small flow graph to explore the gr-radar toolbox's CW signal generator and target/channel simulator.	79
D.2	Small flow graph to explore the gr-radar toolbox's FMCW signal generator.	80
D.3	Simple transmit flow graph using FM modulation to explore using the LimeSuite transmit block as well as a few other blocks.	80
D.4	Simple receive flow graph using FM modulation to explore using the LimeSuite receive block as well as a number of other blocks. Note that this flow graph has quite a few extra blocks for GUI sink control.	81
D.5	This flow graph attempted to combine the transmit and receive flow graphs (Figs. D.3 and D.4). Operation did not throw any errors, however the data that did come through had issues. The most likely issue is sample rate mismatching.	82

D.6	Full FSK radar simulation. The original flow graph is provided in the gr-radar examples folder. The flow graph was modified to use the LimeSuite's transmit and receive blocks instead of a channel simulator block (located in the gr-radar toolbox).	83
E.1	Example of successful verification terminal outputs.	87

CHAPTER 1. INTRODUCTION AND BACKGROUND

Visually inspecting satellites to ensure that they do not have any structural or exterior disrepair is quite difficult once they are in orbit. Currently people are sent on space walks or in the case of the ISS, a large mechanical arm is maneuvered to take pictures of the outside. In both cases, the procedure can be costly and/or dangerous [1]. In conjunction with NASA, BYU is developing a series of small satellites called CubeSATs that are 10 x 10 x 10 cm in size. These CubeSATs are designed to inspect the exterior of the launching body. The CubeSAT currently in development is called the PICS (passive inspection cube satellite). As the name suggests, it passively tumbles through space after being ejected. Cameras are located on each face of the PICS to ensure that no matter the rotation or translation it has view of the launching satellite. The images can be stitched together using minimal post-processing to obtain the necessary views.

The next version of CubeSAT, which is still in the proposal phase, is called the TRICS (translation, rotation inspection cube satellite). The TRICS has control over its translation and rotation with respect to the launching satellite. This is achieved through magnetorquers and small thrusters. This reduces the number of cameras to one, which leaves more space, power, and money for other systems. TRICS requires information about the deploying satellite including, orientation and distance. Orientation is not explored in this work; however, solving the distance problem is. This work explores the feasibility of using a software defined radio (SDR) to create a small, low-cost, ranging radar suitable for use in a small spacecraft. Such a radar would need to have short range detection (between 50 to 2000 meters), fine range resolution (at most 20 meters), while being space efficient, cheap, and low power. In order to better understand and fulfill these requirements, background information concerning digital signal processing (DSP) and radar hardware is necessary.

1.1 Digital Signal Processing

The understanding and selection of different radar system architectures is crucial to this work as it determines how the SDR is programmed. In the case of radar systems, the architecture can vary from simple continuous frequency transmission to modulated signals [2].

Ranging radars can detect distance by transmitting a signal and waiting for the signal to come back also known as time of flight (TOF). The TOF can be measured since the waves travel at the speed of light, providing a relationship between distance and time. The distance d to the detected object can be calculated according to $d = \frac{ct}{2}$ where $c = 2.998 * 10^8$ m/s (the speed of light) and t is the measured TOF. The result is divided by two since the signal has to travel the distance twice (to and from). Due to the fact that the speed of light is so fast, measuring this time can be difficult with low-cost hardware and many techniques have been developed to address this problem. Here we consider a number of DSP aspects that play a part in radars.

1. **Channel characteristics** - The channel is the space that the signals travel through. On Earth, it is polluted with physical objects (clutter) and other electromagnetic signals (interference), which can have negative effects on the signal propagation, fidelity, and amplitude. Some physical objects can be opaque to the signal and absorb the signal, causing the radar to detect nothing. Other electromagnetic signals can be detected by the radar receiver and give false positives, constructively or destructively couple to the signal fundamentally altering the signal. In addition to these obstacles, each channel has an intrinsic attenuation based on the contents of the channel. As an electromagnetic signal passes through the channel, there is calculable attenuation that depends on frequency and travelled distance [3]. For example, on earth the channel is filled with air, which is nitrogen and oxygen gas. In space the channel is different due to the lack of obstacles and atmosphere. Without these factors, a signal can propagate in free space with negligible attenuation and interference and targets are easier to detect since there is no clutter.
2. **Signal characteristics** - The capabilities of a radar are largely determined by the kind of signal it transmits. For example, the simplest radar technique is to transmit a pure sinusoid. Since the signal itself does not have any special features other than repeating every period, it is difficult to extract timing information from the signal [4]. It is tempting to think that

it is possible to tell distance if the received signal is less than one half period delayed; the distance data could be calculated since the instantaneous differential voltage would be non-zero, but this is not true since the possibly complex nature of the channel can introduce phase attenuation and skew.

However a simple, single frequency radar is most useful for detecting velocity according to the Doppler effect. If the detected object is moving fast relative to the radar, then the return signal has a higher frequency. This difference in frequency from transmit to receive is processed to detect velocity. The concept of the Doppler radar, called a continuous wave (CW) radar, is logically expanded to the linear frequency modulated continuous wave (LFMCW) radar.

The signal produced by an LFMCW radar changes steadily from an initial to target frequency (typically low to high) and repeats. The unique characteristic in this signal is that the frequency difference versus time is related to the target distance. This can be used to detect the distance to an object since the radar is transmitting one frequency and receiving another frequency simultaneously. The difference in these frequencies is correlated to the TOF. Other more complicated signals are often used in other applications where signal integrity is necessary or where there are other special needs.

3. **Maximum range** - Channel attenuation and signal processing constraints both have limiting affects on a ranging radar's maximum detectable range. Channel attenuation affects maximum range by swallowing the signal in noise if the object is too far away. The strength of a signal drops off quadratically because the transmit energy is spread over an quadratically increasing area as the radius, or distance of travel, increases linearly according to r^{-2} [3]. As the energy spreads out thinner and thinner, so too does the return signal.

The transmit signal also affects the maximum usable range. As previously noted, range is often detected by measuring the time difference between the transmit and receive signals, for example with frequency modulation. The frequency modulation needs to restart as the trend cannot continue indefinitely, as that would require infinite memory or bandwidth/mixed signal IC resolution respectively. When the pattern repeats, the time defining characteristic is also lost. Often the maximum range can also be increased, by increasing the time duration

of the frequency modulation or the length of the bitstream. This phenomenon of not knowing the the actual distance due to chirp or bitstream repeats is known as range ambiguity [5].

4. **Range resolution** - Lastly, range resolution is essentially the countable distances that the radar can detect. Each increment in range resolution can be called a bin, a quantized value. If a detected object is eleven meters away, but the radar's range resolution is ten meters, the radar reports that it is ten meters away. On the other hand, if the object is sixteen meters away and its range resolution is still ten meters, the reported distance is twenty meters. Therefore necessary resolution needs to be considered. If the radar is expected to detect objects ten kilometers away, then a range resolution of ten to even one hundred meters may not be so bad, but if it is expected to detect objects within one hundred meters, then a finer range resolution is necessary. Range resolution is determined by the bandwidth of transmitted data. Higher bandwidth results in finer range resolution. There are techniques involving overlapping bins, but their complexity and applications are outside the scope of this work.

Because this work focuses primarily on space applications, many of these considerations can be simplified. Channel characteristics are simplified as there is nearly zero channel attenuation and interference is small since other electromagnetic sources are minimal. Due to the lack of interfering communications, much simpler transmission schemes can be employed. As such, an LFMCW radar is considered a good choice due to its simplicity and ability to detect range, fitting the requirements for the TRICS. Maximum range and range resolution in LFMCW radars are considered in the simulation chapter, but it is sufficient to say that in LFMCW radars the chirp/modulation duration and bandwidth of the signal need to be balanced. As one increases, the slope of the corresponding Doppler time plot (frequency vs time) of the chirp changes. A steep slope necessitates higher resolution mixed signal hardware, while a shallower slope requires a higher bandwidth radio. Both of these can become quite expensive. In order to keep the cost down, range resolution and maximum range need to be carefully considered in tandem with hardware constraints.

1.2 Hardware

Radar hardware can be simplified to a radio transmitter and receiver operating together connected by a processor for control and calculations. The core building blocks are signal generators, oscillators, mixers, analog to digital and digital to analog converters (ADCs and DACs), filters, and amplifiers. Transmitters and receivers both use these components but vary slightly in underlying architecture and use. In addition to using the basic radio hardware requirements, a radar needs to perform DSP on the transmit and receive signals in order to determine velocity, distance, or both. To do this, typically a digital processor of some sort is also available in the system. Some of the fundamentals components are discussed here.

1. **Signal generators/Synthesizers** - In CW radars, the signal generator may seem fairly trivial as a simple sinusoid wave is all that is needed. However, even this simple signal can be difficult to generate. There are many different techniques for generating simple sinusoids from analog oscillators to digitization. When more complex radar transmit signals are needed, the signal generation design problem similarly becomes more complex.

Regardless of the signal generation method, there are a few metrics, most notably stability and spectral purity, which are distinct yet related [6]. The stability of a sinusoid is essentially how much the signal deviates from the signal's formula, excluding noise. For example, a CW signal is defined as $V = e^{j\omega t}$ (note that in this thesis, signals are written in complex form for simplicity). If the signal's amplitude varies, but still has the expected period, the signal is considered to have poor signal stability even though it is technically the correct frequency. Spectral purity, is fundamental. Ideally the FFT of a CW signal ($V = e^{j\omega t}$) is a delta function at ω . When this signal is generated and measured using a spectrum analyzer or the FFT function on an oscilloscope, we see a spike at ω surrounded by a skirt. A skirt is the sloping off from the point of interest off to the noise floor. This means that the power in the signal is not purely concentrated in the desired frequency, but instead has some power leakage to the surrounding frequencies.

2. **Amplifiers** - Amplifiers in radars are needed to boost signal strength for transmission and processing. Typically there are two notable amplifiers. On the transmit side, a power amplifier (PA) is used to boost the power of the signal before feeding it into the antenna. It is

important to emphasize that a power amplifier boosts power instead of voltage since many amplifiers only boost voltage with relatively little current behind it. Antennas need power to transmit a strong signal and voltage alone does not afford this as $P = VI$. Conversely, the receive side of the radar uses a low noise amplifier (LNA). These are specially designed amplifiers that increase the signal with minimal noise. This increases the signal to noise ratio (SNR) which in turn makes it easier for the DSP to correctly process the desired signal.

3. **Mixed-signal hardware** - The choice of analog to digital converters (ADCs) and digital to analog converters (DACs) within a radar system is crucial as it largely defines the bandwidth of the transmit and receive signals. This is due to the Nyquist theorem. Nyquist states that it is only possible to deterministically reconstruct a signal from periodic sampling if the sample rate exceeds the maximum signal frequency by a factor of two ($f_{\text{samp}} > 2f_{\text{max}}$) [4]. ADCs tend to be the slower of the two mixed signal circuits, since they often operate by sequentially generating the corresponding bits detailing the input analog voltage. Architectures exist to address this problem such as pipeline and flash [7]. These architectures have the problem that as the bit resolution becomes higher, the circuits become non-linearly larger and power hungry. As such, modest bit resolution (12 to 16 bits) are usually selected as they provide sufficient resolution while running at speeds that afford acceptable frequency bandwidth.
4. **Filters** - One of the unfortunate side effects of modern transceivers is the generation of aliased signals. One special case is called an image frequency. These are generated when downmixing a signal from its carrier frequency to baseband [8]. Fortunately, if the bandwidth and carrier frequencies are chosen carefully, the image frequency can be removed with a filter. Filters find place in a number of places along the signal pipeline, both to and from the channel. Filters can be either analog or digital. The simplest analog filter is an RC filter consisting of a resistor and capacitor [9]. Depending on the desired configuration, it can be either a high or low pass. A simple digital filter architecture is a finite impulse response (FIR) filter [10]. Both analog and digital filters have their place and are used at different points in a radar.
5. **Processor** - Choosing a processor is a crucial decision. The processor needs to meet the demands of the rest of the system while simultaneously having all the other hardware interface

and correctly work in tandem. There are three main groups of contenders to fill this need: microcontrollers, microprocessors, and computers.

Microcontrollers are the simplest of the three choices. They are typically categorized as a chip that has general purpose input/output (GPIO) pins that can be controlled through code. The code that is programmed to the chip then runs on loop indefinitely as long as it is powered. Another attractive aspect is that they are typically self-contained chips, needing very few external components such as memory or additional specialized compute units. Typically, the only external necessary components are quartz crystals that provide tight oscillation for the internal clock. While this does provide ease and flexibility in design and budget, they tend to be on the lower end of compute capability as well as on-the-fly reconfigurability.

Microprocessors encompass a large gray area between microcontrollers and computers as they are usually small, low power, and sport GPIO akin to microcontrollers, but boast higher compute performance and complexity like computers. Perhaps the most prolific microprocessor is the Raspberry Pi. Microprocessors tend to be more difficult to use in systems, when not packaged as a single board computer (SBC) like the Raspberry Pi, since they require more external hardware as it is not provided on the die. This means that external memory and IO ports (aside from GPIO) such as USB and ethernet often need dedicated controller ICs. In addition to the hardware difficulties, many microprocessors are capable of running full operating systems effectively turning them into mini computers. The power that this provides is staggering as a scheduler can be run as part of the kernel allowing for multiple processes to run simultaneously. Unfortunately, this can add additional overhead since the operating system must be suited for the instruction set of the processor, such as x86 (Intel proprietary), ARM (ARM proprietary), RISC-V (open-source), etc.

The final processor option is to use a computer. The advantage that a computer has over microcontrollers and microprocessors is most notably its instructions per clock (IPC) and clock rate. Computer processor designers maximize their IPC and push clock rates since these two metrics correlate to the amount of data it can process in a given amount of time. Clock rates on computer processing units (CPUs) are significantly higher than that of microcontrollers and microprocessors. Further widening the performance gap, computers often

have additional hardware that can substantially increase data throughput such as graphics cards, for parallelized duplicate operations. All of this comes with a price - computers have significantly higher cost, power consumption, and lack of GPIO.

1.3 Takeaway

Since this work is focused on determining whether an SDR can cover the hardware needs of a radar, nearly all of the hardware considerations can be abstracted into the SDR with the exception of the processor. This does not mean that knowledge of these pieces is unnecessary; in actuality such knowledge is quite useful. SDRs are not magic boxes that produce and receive signals mysteriously. Instead they have all of the components previously described, but are designed in a manner to generalize operation and make said operation programmable. As such, the understanding of these core components supports an intelligent SDR selection and details which “knobs and buttons” to push with code.

In addition to hardware abstraction, the electromagnetics involved in typical radar designs are not considered as they fall outside the scope of this work. Antenna design and signal conditioning, outside the SDR, fall within the realm of radar actualization. As such, with the DSP and hardware in mind, a number of selections can be set to guide our exploration.

1. The SDR selected for this work is the LimeSDR-Mini by MyriadRF. For details why this SDR was selected, refer to Appendix A.
2. Any processor can be used since the processor is not pertinent to the proposed work. As such, the processor changes a few times through the work and is discussed as encountered.
3. The LFMCW radar architecture is used since it is the simplest ranging radar [11] and interferers (physical and electromagnetic) are not problematic in space.

With that in mind, the following chapters explore the feasibility of using the LimeSDR-Mini as an LFMCW radar. Chapter 2 covers radar principles and LFMCW radar techniques through MatLab simulation for DSP understanding; Chapter 3 examines GNU Radio Companion

and its suitability as a programming platform for the SDR; and Chapter 4 explores the LimeSDR-Mini hardware more closely and how to use the Python API provided by MyriadRF. Supplemental information and scripts/code are found in the appendices.

CHAPTER 2. LFM CW RADAR AND SIMULATION

Radars work on the same principles as radio communication, such as AM or FM. They differ from the traditional concept of radio communication by receiving the transmitted signal back as an echo. The echo that is received can be processed and analyzed, which can then determine the distance and velocity of the object off which the signal reflected. The simplest way to conceptualize this operation is that the signal takes time to travel through space, reflect off the object, and return to the radar. This time can be measured and used to determine the distance of flight. Determining the distance is not actually this simple. Designing and building a timer that runs fast enough with sufficient resolution for a short range radar is quite difficult and expensive. Instead, clever signal design and digital signal processing is typically used to determine actual range and velocity.

2.1 Simulation Requirements

In order to understand where to begin designing a radar using a software defined radio, we first turn our attention to the underlying concepts and signal processing. To fully grasp these concepts a MatLab simulation was written to provide a sandbox wherein variables could be changed and the corresponding output observed. To ensure that it is useful, the simulation follows a few requirements outside typical radar characteristics.

The simulation generates and transmits a linear frequency modulated continuous wave (LFMCW). The LFMCW is selected for this application as it is one of the simplest forms of radar capable of detecting distance. Such radars can have issues with the signal getting corrupted due to its simple nature, but these effects are not a problem as this work is intended for space application where interferers and channel attenuation are minimal.

In addition to the LFMCW, the script modulates and demodulates the signal for radio communication using an inphase-quadrature (IQ) modulator. IQ modulators are ubiquitously used in

SDRs since they can generate any modulation scheme by changing a few parameters. This enables the SDR to be configurable through code with fixed hardware.

2.2 Signal Analysis and Generation

The starting point of any radar is generation of the transmit signal. For this work, we use an LFM CW (often called a chirp). Its mathematical model is

$$V_{\text{LFMCW}}(t) = e^{j(\alpha t^2 + \beta t + \phi_0)} \quad (2.1)$$

where α , β , and ϕ_0 are the chirp bandwidth, the base frequency, and initial phase, respectively. To understand this formula and decide what the parameters are, consider

$$V(t) = A \cos(2\pi f t) \quad (2.2)$$

$$= A \cos(\phi(t)). \quad (2.3)$$

Eq. 2.2 is a sinusoid continuing from $t = -\infty$ to $t = \infty$ at the constant frequency f . Eq. 2.3 states that the sinusoid can also be described by the phase function $\phi(t)$ described as

$$\phi(t) = \phi_0 + 2\pi \int_0^t f(\tau) d\tau \quad (2.4)$$

where $f(\tau)$ is the desired frequency, which is not necessarily constant. To have the signal change linearly, we use $f(\tau) = c\tau + f_0$. Ultimately, after working through the math, the final formula comes out as

$$V(t) = \cos\left(\frac{f_1 - f_0}{T} \pi t^2 + 2\pi f_0 t + \phi_0\right) \quad (2.5)$$

where f_0 is the chirp starting/base frequency, f_1 is the chirp ending frequency, T is the chirp duration in seconds, and ϕ_0 is the initial phase. Comparing Eqs. 2.1 and 2.5, we define α , β , and ϕ_0 . The difference between Eqs. 2.1 and 2.5 becomes apparent when comparing the two equations after using Euler's identity on Eq. 2.1,

$$V_{\text{LFMCW}}(t) = \cos(\alpha t^2 + \beta t + \phi) + j \sin(\alpha t^2 + \beta t + \phi). \quad (2.6)$$

Eq. 2.5 is a purely real signal whereas Eq. 2.1 has both a real and imaginary part. In addition, the real and imaginary portions of Eq. 2.6 are 90° out of phase from each other. This means that the real and imaginary parts are orthogonal or in quadrature. The fact that there is a real and imaginary portion to Eq. 2.1 is one of the reasons we choose to use Eq. 2.1 over Eq. 2.5, as discussed at the end of the chapter.

```

1 alpha = fbw*pi/T;
2 beta = 2*pi*f0;
3 phi = 0;
4 bb_lfmcw = exp(1j*(alpha.*t_part.^2 + beta.*t_part + phi));

```

Listing 2.1: Simulation Chirp Generation

Listing 2.1 gives a MatLab code snippet that generates the LFM CW. Note that $f_1 - f_0$ from Eq. 2.5 is replaced with $f_{bw} = f_1 - f_0$ since it is the bandwidth of the chirp signal. In the case of an LFM CW radar, as the bandwidth increases, the range resolution of the radar also increases according to

$$S_r = \frac{c}{f_{bw}}. \quad (2.7)$$

In the case of the LimeSDR-Mini, the maximum possible bandwidth is 30.62 MHz. Since a finer range resolution is desirable, the maximum bandwidth of the LimeSDR-Mini is used in our simulation.

Note that on line 3 of Listing 2.1, $\phi_0 = 0$. This is because the receive signal is processed in relation to the transmitted signal. Any phase offset in the transmit signal is the same in the return signal, thus the initial phase is irrelevant.

2.3 IQ Modulation

As stated previously, SDRs are able to generate any radio signal thanks to IQ modulation. This enables any modulation scheme through easily changed parameters and fixed hardware. To understand IQ modulation, an understanding of mixing is necessary.

At its core, mixing is simply multiplying one signal by another in the time domain. We know from digital signal processing that if two signals are multiplied in the time domain then they

are convolved in the frequency domain. If one of the signals is a sinusoidal, convolution essentially copies one signal centered around the other (instead of the origin).

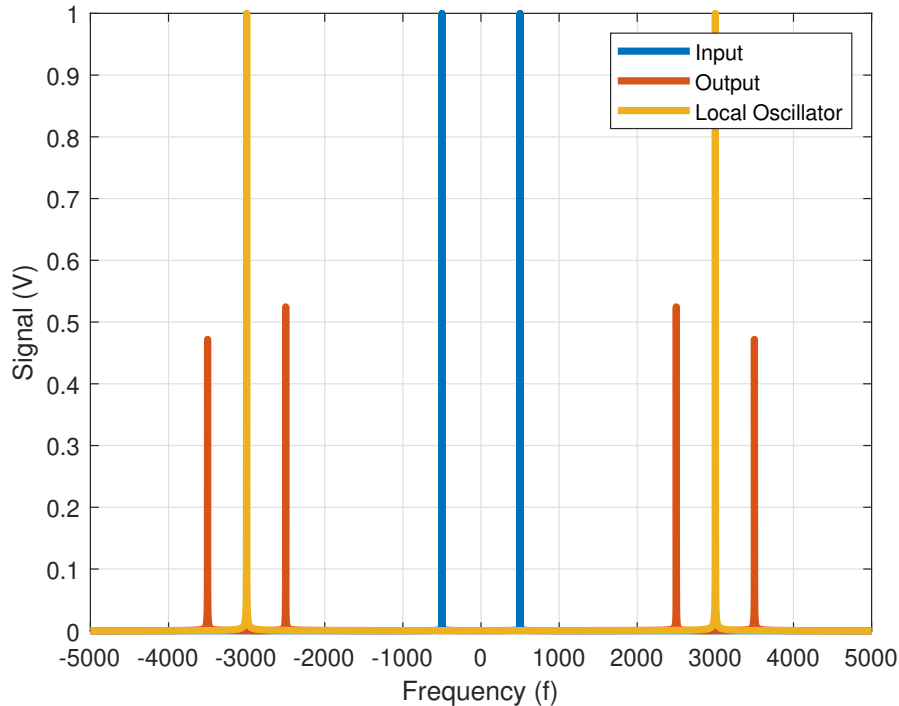


Figure 2.1: Convolution example shown in frequency domain. Input is a 500 MHz signal, the local oscillator frequency is 3 kHz, and the output signal is the input signal centered around each local oscillator delta.

Fig. 2.1 shows the fast Fourier transform (FFT) of an input, mixing, and output signal. The input signal is a 500 Hz cosine, the mixing signal is a 3 kHz cosine, and the resulting signal is a copy of the input cosine copied about the two deltas of the mixing signal. The four deltas of the upmixed signal have a lower amplitude than the original signal since the input power of the signal split between four deltas, instead of the original two deltas. The difference in power levels can be even more drastic in a realized system since there is no such thing as a perfectly efficient system where no power is lost.

It is very rare that a pure sinusoid is transmitted in an actual communication system. In an actual communication or system, the transmitted or input signal to the mixer has some bandwidth. In this case, each blue delta shown in Fig. 2.1 is replaced with the signal spectrum, e.g. a rectan-

gle. The width of the rectangle shows the bandwidth of the signal. When mixing a signal with bandwidth, the resulting FFT shows the original rectangles (one positive and one negative), copied about the mixing signal, just the same as if the input were a single frequency.

IQ Modulation is similar to the mixing method discussed above, but instead of mixing with a single oscillator frequency, the signal is mixed with one oscillator and another oscillator 90° out of phase from the first. These two oscillators are called inphase and quadrature, respectively. The basic IQ transmitter is shown in Fig. 2.2 and the basic IQ demodulator is shown in Fig. 2.3.

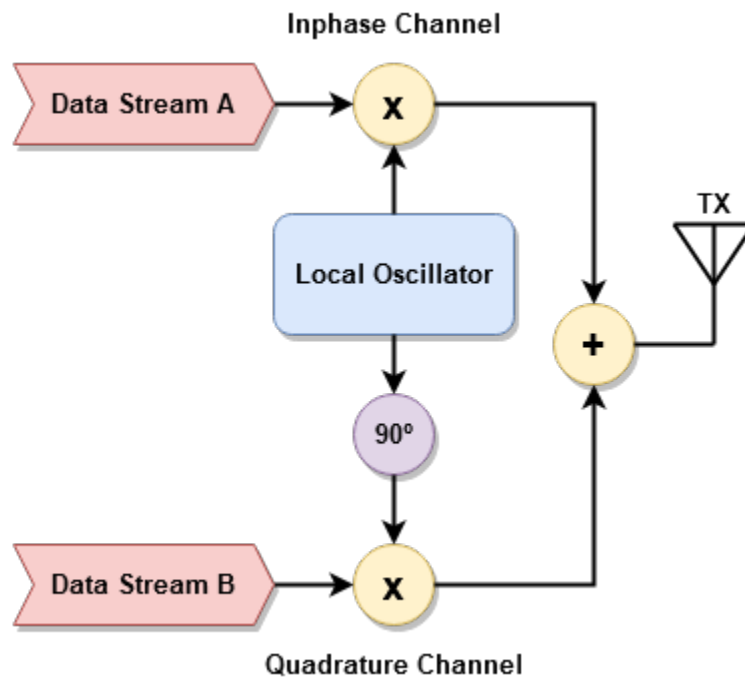


Figure 2.2: Block diagram of a general IQ modulator with two input data streams output to a single antenna.

If the local oscillator is outputting a cosine wave, then the same signal with 90° offset is a sine wave. Looking at the FFT for a cosine and sine wave in Fig. 2.4, we can see that both have a positive delta in the positive frequency space, but in the negative frequency space, they have opposite signs. If these two signals are added together, the negative frequency deltas cancel, leaving

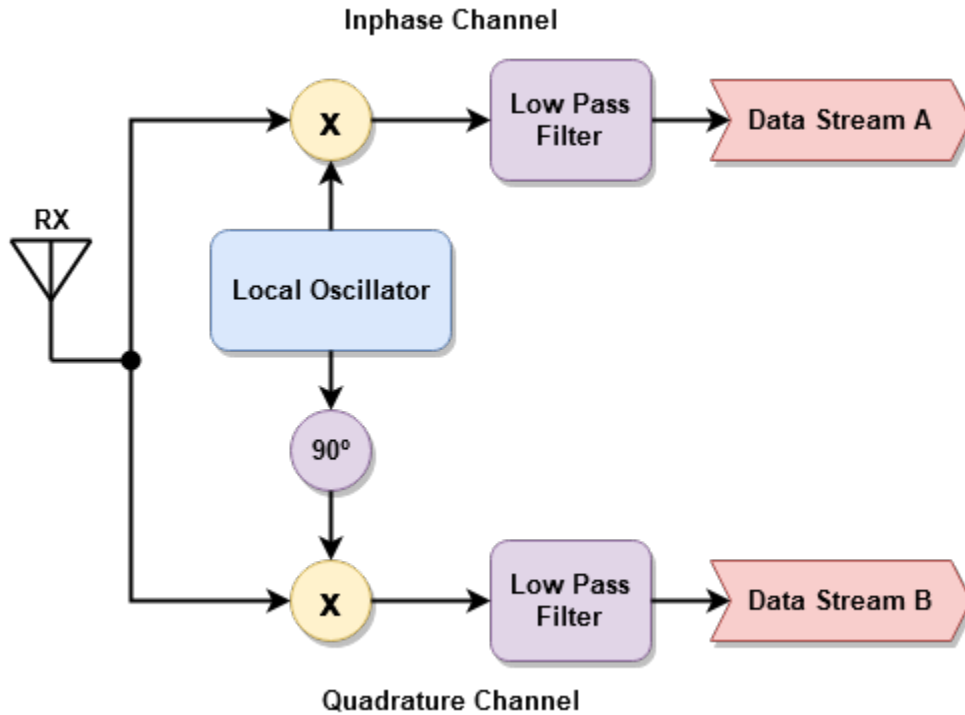


Figure 2.3: Block diagram of general IQ demodulator with input antenna and resultant data streams.

only a positive frequency delta. This resultant signal can be described using Euler's Identity:

$$V = \cos(\omega t) \pm j \sin(\omega t) \quad (2.8)$$

$$= e^{\pm j\omega t} \quad (2.9)$$

```

1 tx_inph = real(bb_full).*cos(2*pi*flo.*t_full);
2 tx_quad = imag(bb_full).*sin(2*pi*flo.*t_full);
3 tx = tx_inph + tx_quad;

```

Listing 2.2: LFMCW upmixing using inphase-quadrature modulator as depicted in Fig. 2.2.

Listing 2.2 provides a MatLab script to produce an upmixed complex signal using an IQ modulator. The complex signal is split by using the `real` and `imag` functions. These two parts are treated as the inphase and quadrature data streams into the IQ modulator. Each are multiplied with the respective portion of the complex modulator and added together.

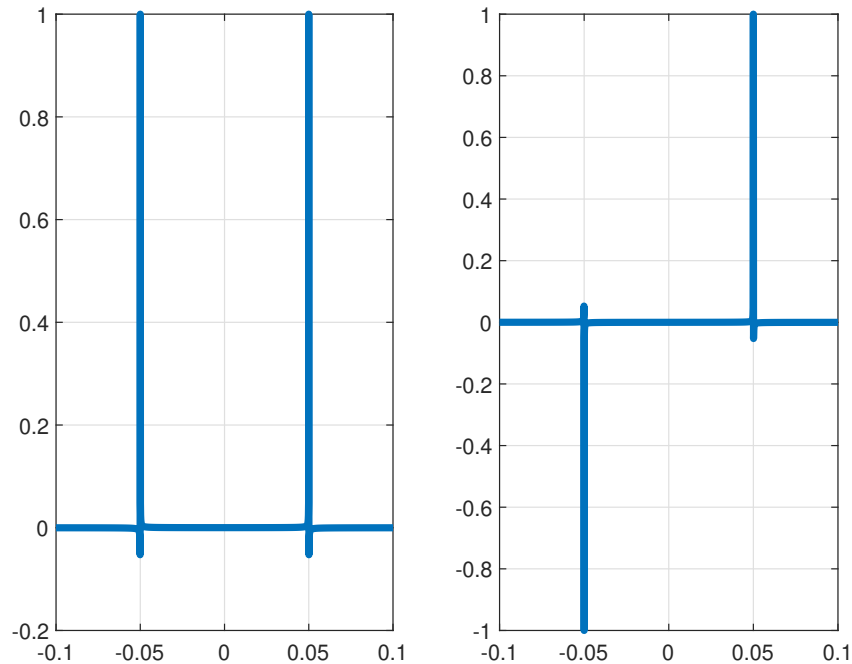


Figure 2.4: Example FFTs of sinusoidal waves. The small spikes below and above the deltas are artifacts from actualizing the FFTs with a finite number of samples. Left: FFT of $\cos(2\pi f)$; Right: FFT of $j\sin(2\pi f)$.

Similar to how the signal is modulated using an IQ oscillator, the demodulator, or receiver, works through the same principles. When the signal is received at the antenna, the signal is fed into both the inphase and quadrature channels of the receiver. With careful design, the signals can be recovered without ambiguity.

Demodulating is done through the same process as modulating. This is helpful because it can utilize the same hardware and techniques used for modulating. Unfortunately it also generates unwanted high frequency signals. Fig. 2.1 shows the FFT of a modulated signal. Fig. 2.5 shows that when a modulated signal is mixed with the same local oscillator again, for demodulation, extra signals are produced at high frequency. These frequencies are called image frequencies. To remove the unwanted signals, the signal is put through a low pass filter.

In the case of the IQ modulator, the process of mixing with the local oscillator and filtering is done twice, once for the inphase and quadrature channels, as shown in Fig. 2.3. Once the signal is demodulated and filtered, the original data stream is restored at baseband, with some attenuation.

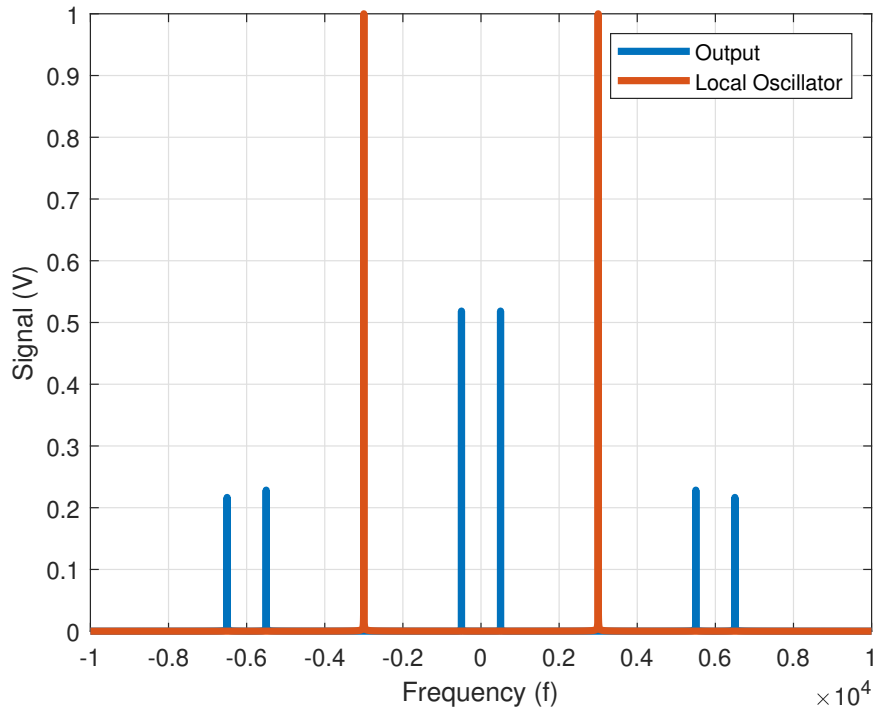


Figure 2.5: Example of a simple demodulation using the output signal from Fig. 2.1 with the same local oscillator frequency of 3 kHz.

```

1 rx_inph = rx.*cos(2*pi*flo.*t_full);
2 rx_quad = rx.*sin(2*pi*flo.*t_full);
3 bb_inph = lowpass(rx_inph, 2*fbw, fmax/T);
4 bb_quad = lowpass(rx_quad, 2*fbw, fmax/T);

```

Listing 2.3: LFMCW demodulation using an IQ modulator and per channel low pass filtering.

Listing 2.3 shows the signal `rx` being demodulated using the inphase mixer on line one and the quadrature mixer on line two. Then lines three and four show the signal being filtered to remove the image frequencies, resulting in the baseband inphase and quadrature signals.

2.4 Range Detection

As stated in the background to this chapter, by employing clever signal design, range estimation can be done through digital signal processing. The overall range detection process is shown in Fig. 2.6.

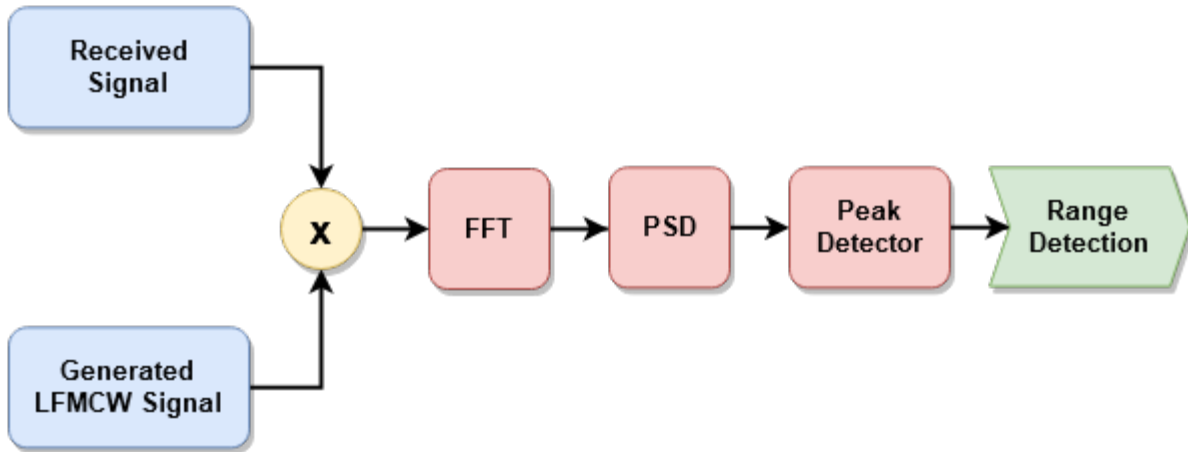


Figure 2.6: General DSP process for LFM CW radar range detection.

It is possible to detect the range of an object by observing the power spectral density (PSD) of the product between the transmitted and received signals due to the changing frequency characteristic of the chirp. When the signal travels and reflects off an object, it returns with some delay. Due to this delay and the changing frequency of the chirp, the received instantaneous frequency is a fixed difference from the current transmit frequency. This frequency difference, as well as the corresponding time delta, is visible in the doppler time plot of an arbitrary chirp in Fig. 2.7 as δ and τ , respectively.

Noting that

$$\frac{\delta}{\tau} = \frac{f_{bw}}{T} \quad (2.10)$$

it is possible to convert between δ and τ .

The first step in finding the range of the target is multiplying the received signal with the transmitted signal. Since they are complex, it is a conjugate multiplication. This is the point that the characteristics of the LFM CW are important. The fact that the transmit and receive signals are the same differing only by a sample delay (which directly correlates to a time delay), if the two signals are multiplied together, the result is a constant frequency equaling the difference of the two input frequencies.

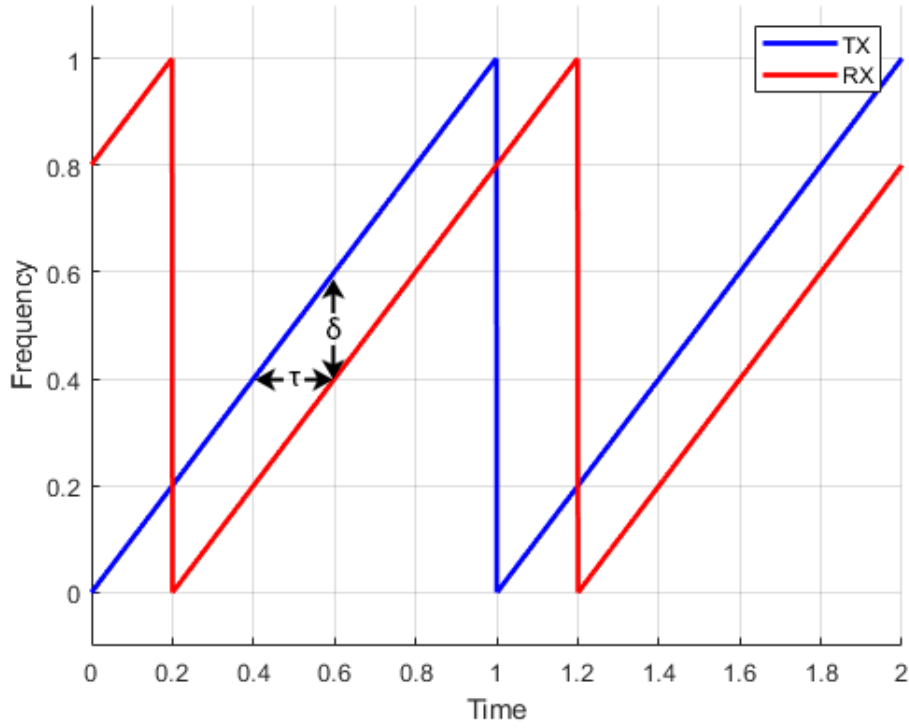


Figure 2.7: Example doppler time plot of arbitrary transmit and receive (LFMCW) signals with frequency and time deltas marked (δ and τ respectively).

To help visualize this point, consider the following equations.

$$x(t) = e^{j2\pi 10t} \quad (2.11)$$

$$y(t) = e^{j2\pi 7t} \quad (2.12)$$

$$z(t) = x(t)\bar{y}(t) \quad (2.13)$$

where $x(t)$ and $y(t)$ are the input signals to our conjugate multiplication and $z(t)$ is the output. Note that signal $x(t)$ has a frequency of 10 Hz and signal $y(t)$ has a frequency of 7 Hz. Both input signals are shown in Fig. 2.8 and the resulting signal $z(t)$ is shown in Fig. 2.9. As expected, signal $z(t)$ has a frequency of 3 Hz, which is the difference between signals $x(t)$ and $y(t)$.

When this principle is extended to the LFM, despite being a non-constant frequency, the result is the same since the frequency delta between the signals is constant, as depicted in Fig. 2.7, and the constant frequency output from our simulation is shown in Fig. 2.10.

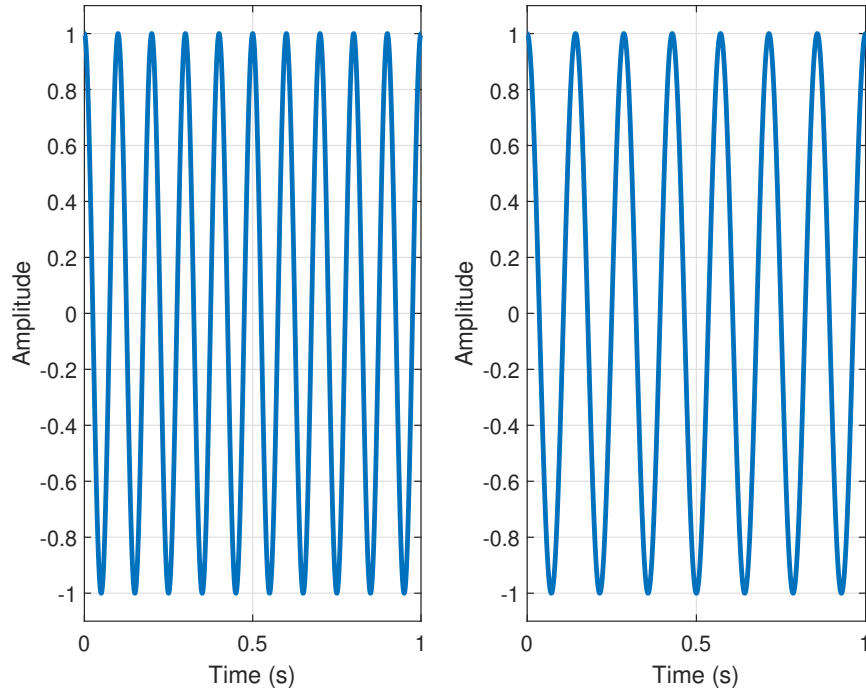


Figure 2.8: Example complex input signals (real portions) over one second. Left: complex sinusoid with a frequency of 10 Hz. Right: complex sinusoid with a frequency of 7 Hz.

The fact that the conjugate multiplication between two complex signals differing by a constant δ is the reason that the LFMCW is considered a clever signal. As the distance that the signal travels increases, the time delay increases as well. The delay in time results in a greater frequency difference, which then translates to a higher constant frequency.

The next step in range detection is to find the PSD of the signal. The PSD, or power spectral density, is the amount of signal power/energy there is for a given frequency. Theoretically, the frequency with the most energy is the detected range of the target in terms of frequency.

To find the PSD, first the FFT of the constant frequency from the conjugate multiplication is taken. The output from the FFT is a complex signal and to see the energy in a given frequency, the magnitude of the signal is taken at the frequency in question. In other words, the PSD is the magnitude of the FFT which is given by

$$\text{PSD} = \sqrt{\text{real}(\text{FFT})^2 + \text{imag}(\text{FFT})^2}. \quad (2.14)$$

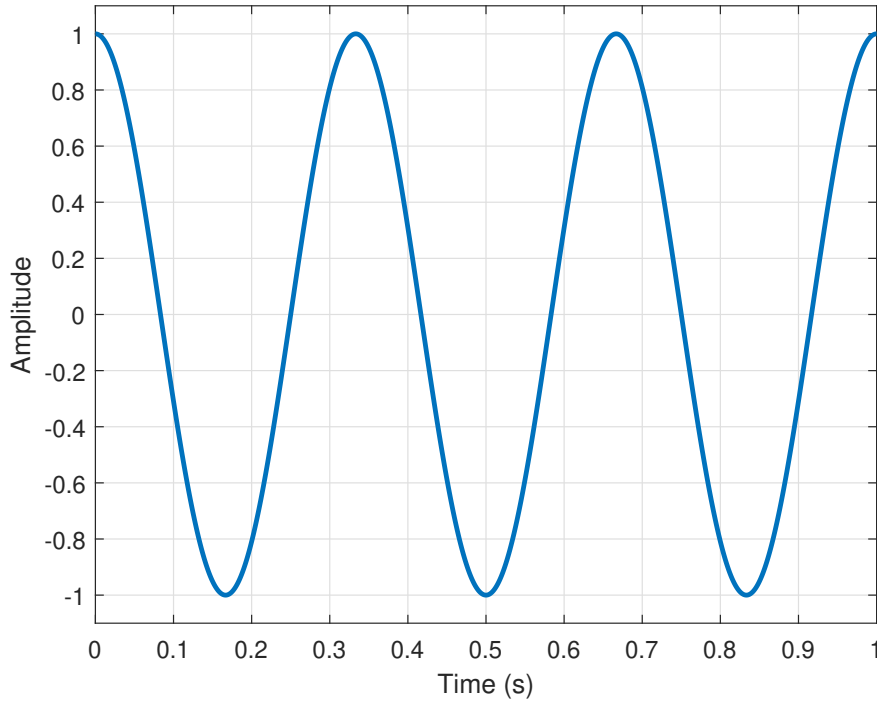


Figure 2.9: Resulting signal (real portion) of the conjugate multiplication of the signals in Fig. 2.8.

The PSD of the constant frequency depicted in Fig. 2.10 is given in Fig. 2.11.

It may seem counterintuitive that the detected frequency is negative; however, remember that a negative delta is indicative of a complex sinusoid with a negative in the exponent such as $e^{-1j\omega t}$. Using Euler's Identity, we know that this just means that the imaginary sine portion is negated. As such, it is a 180° phase rotation in the imaginary portion.

From here, to detect the range, it is just a matter of detecting the peak frequency and converting it to distance, rather than frequency. Detecting the strongest frequency through code is trivial, so it is not discussed here. To find the detected range is just a matter of manipulating units given our chirp rate and the speed of light. Using the following equation results in the detected range

$$\text{Range} = -f_{\text{target}} \frac{T}{f_{\text{bw}}} c \quad (2.15)$$

where f_{target} is the maximum power frequency detected by using the PSD, $\frac{T}{f_{\text{bw}}}$ is the chirp rate of the LFM CW, and c is the speed of light. The negative is to account for our negative detected frequency due to our phase rotated imaginary signal.

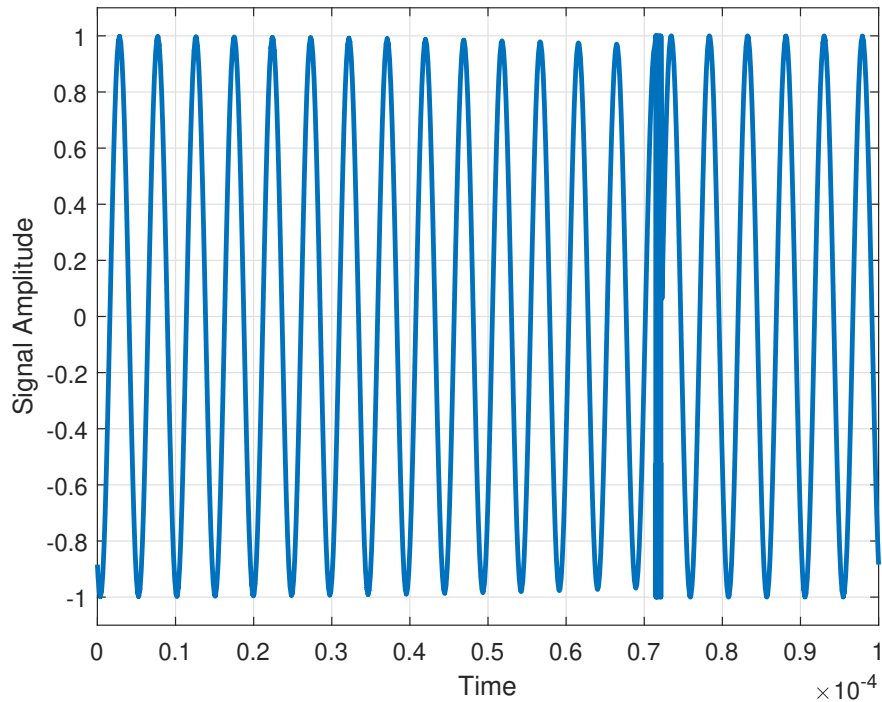


Figure 2.10: Example of constant wave resulting from transmitted and received signal conjugate multiplication. The high frequency noise (located just right of $7 \mu\text{s}$) is a result of the chirp restart.

The following code snippet is from the MatLab simulation to detect the simulated range.

```

1 delta_r = c/fbw;
2 tx_sampled = bb_full(index+1:index+sample_window);
3 rx_sampled = bb_reconst(index+ch_delay+1:index+sample_window+
   ch_delay);
4
5 range_data = tx_sampled.*conj(rx_sampled);
6 [f, fft] = plottableFFT(range_data, T, 0);
7 fft_power = abs(fft)/max(abs(fft));
8
9 max_power = max(fft_power);
10 tgt_bin = find(fft_power == max_power);
11 tgt_freq = f(tgt_bin);

```

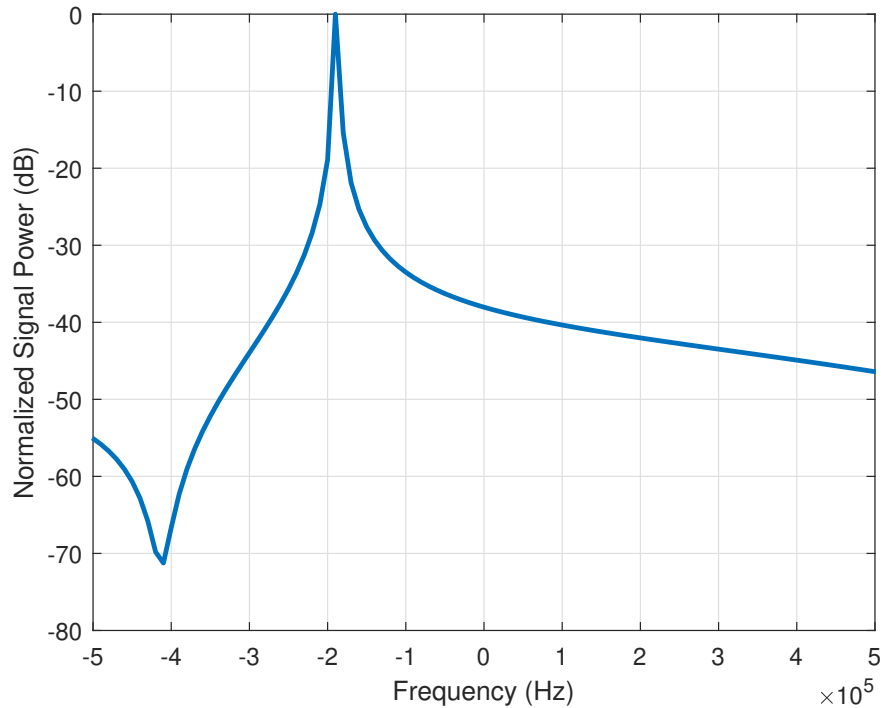


Figure 2.11: Example PSD of a given signal generated from the conjugate multiplication between transmit and receive signals.

```
12 range = -tgt_freq*T*delta_r;
```

Listing 2.4: Matlab simulation code to implement LFMCW range detection.

2.5 Nonidealities

There are a number of things to consider when designing an LFMCW outside the theoretical aspects; however, only two are discussed here due to the thesis focus: the maximum detectable range and channel noise.

2.5.1 Maximum Detectable Range

It is tempting to say, when observing Fig. 2.7, that the maximum detectable range is when the receive signal returns just before the next chirp is about to transmit. If the receive signal were to arrive just after the following chirp started, then the signal processing would not be able to tell

the difference between a close object and a far object. If this were the case, the maximum range would be given by

$$\text{Range}_{\max} = \frac{cT}{2}. \quad (2.16)$$

This, unfortunately, is not quite the case. The system has to be actualizable, and as such, the receiver of the system needs to digitize the signal coming back in with an analog to digital converter (ADC). The ADC has a maximum sample rate, and due to Nyquist. This adds another stipulation on the maximum range. The actual maximum range equation is given by

$$\text{Range}_{\max} = \frac{cT}{2} \frac{f_{\text{adc}}}{f_{\text{bw}}} \quad (2.17)$$

where f_{adc} is the maximum sample rate of the ADC and f_{bw} is the bandwidth of the transmit signal. In almost all cases, the ratio of the bandwidth to ADC sample rate reduces the actual maximum detectable range. In unusual cases where the ratio is greater than one, Eq. 2.16 is used.

2.5.2 Channel Noise

When the signal is transmitted from the radar, the signal travels through the channel to the target where it is reflected back to the radar. The received signal includes noise from the channel. In this MatLab simulation, three cases can be considered: 1) no noise, 2) gaussian white noise, and 3) uniform white noise. In addition to the noise, the amplitude of the noise is configurable. Fig. 2.10 shows the conjugate multiplication of a transmit receive pair without noise, whereas Fig. 2.12 shows the same but with added uniform white noise with an amplitude fifty times greater than the transmitted signal.

Note that for each of these scenarios, the signal amplitude is the same. As the noise power increases the signal is completely drowned in the noise. The method described in this chapter accounts for noise and handles it without additional processing. When the signal is received, it is put through a low pass filter. This attenuates the noise power outside the desired signal bandwidth, since the noise is white and uniform across all frequencies. Fig. 2.6 shows the PSD for range detection without channel noise, and Fig. 2.13 shows the PSD for range detection with uniform white channel noise at fifty times the transmit amplitude.

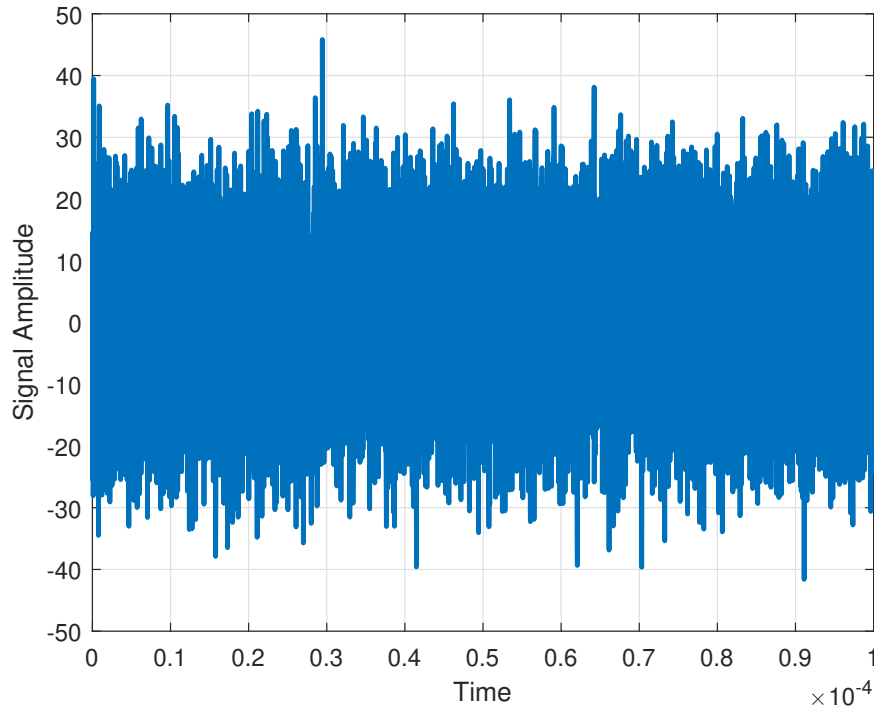


Figure 2.12: Example conjugate product of transmit receive pair with added uniform channel noise fifty times greater than transmit signal.

2.5.3 Antenna Crosstalk

The radar simulated in this chapter is a non-cooperative. All of a non-cooperative radar's hardware is located on a single body. As such, the radar body both transmits and detects the same carrier frequency. Consider a non-cooperative radar that utilizes isotropic antennas. As the signal travel distance increases linearly, the signal power decreases quadratically, according to r^{-2} . This means that the signal travelling directly from one antenna to the other, due to the isotropic nature of the antennas, has significantly higher strength than the signal that reflects off the target.

There are a few ways to resolve this issue such as the use of a cooperative radar. Cooperative radars have transceivers on both the main body as well as the target body, as shown in Fig. 2.14. In this way, one transceiver can transmit one frequency and detect another while the target body receives the one and transmits the second. This work explores only non-cooperative radar as it reduces the amount of hardware needed and also allows the radar to detect any object.

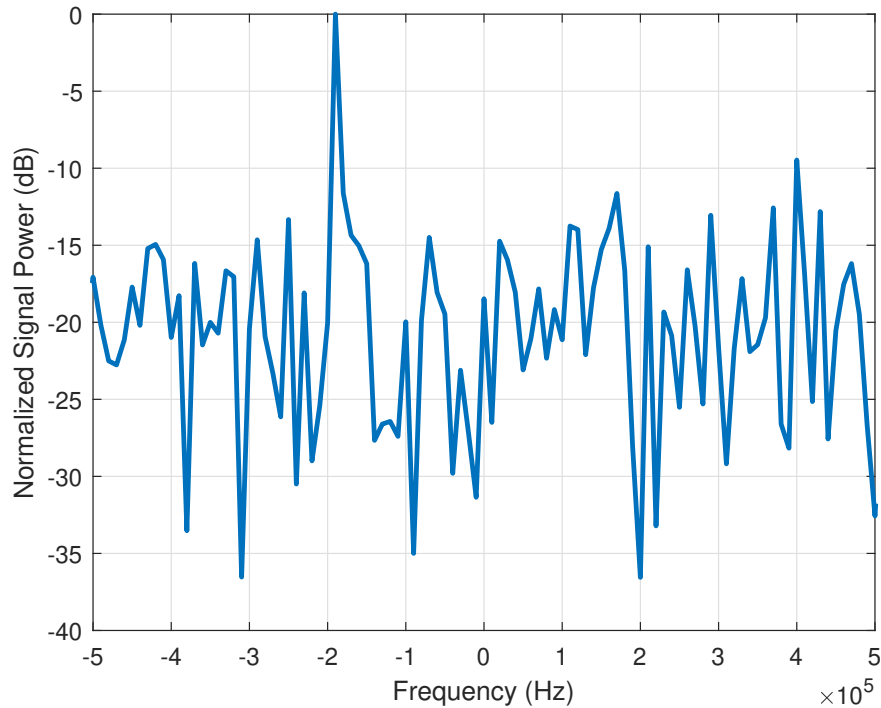


Figure 2.13: Example PSD of a given signal generated from the complex product between transmit and receive signals with uniform white channel noise at fifty times the transmit amplitude.

This flexibility is desirable for a CubeSAT as it can be deployed from any satellite without the satellite needing any modifications.

The other way to remove the effects of crosstalk is through high pass filtering. This is because crosstalk manifests in the DSP as a DC signal. This is apparent in Fig. 2.15 as a DC offset and in Fig. 2.16 as a delta at zero hertz. By understanding the general distance of object and having a high-order high pass filter (HPF), these effects can be removed.

2.6 Simulation Contribution

The LFMCW simulation detailed in this chapter, despite being idealized in many regards, provides this work many important insights including, but not limited to, the effects of RF bandwidth, chirp duration, and noise on the operation of an LFMCW radar. By consulting this simulation and playing with its parameters, a more intelligent configuration of the LimeSDR-Mini is achievable.

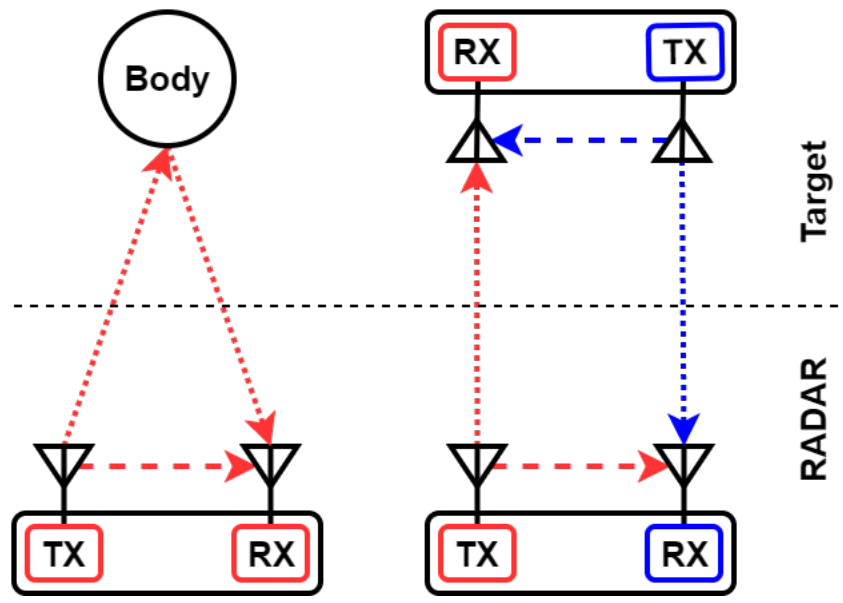


Figure 2.14: Comparison of non-cooperative (left) and cooperative (right) radar topologies. The color of the arrow indicates the carrier frequency, while the color of the box around TX or RX indicates the frequency the radio is tuned to transmit/receive.

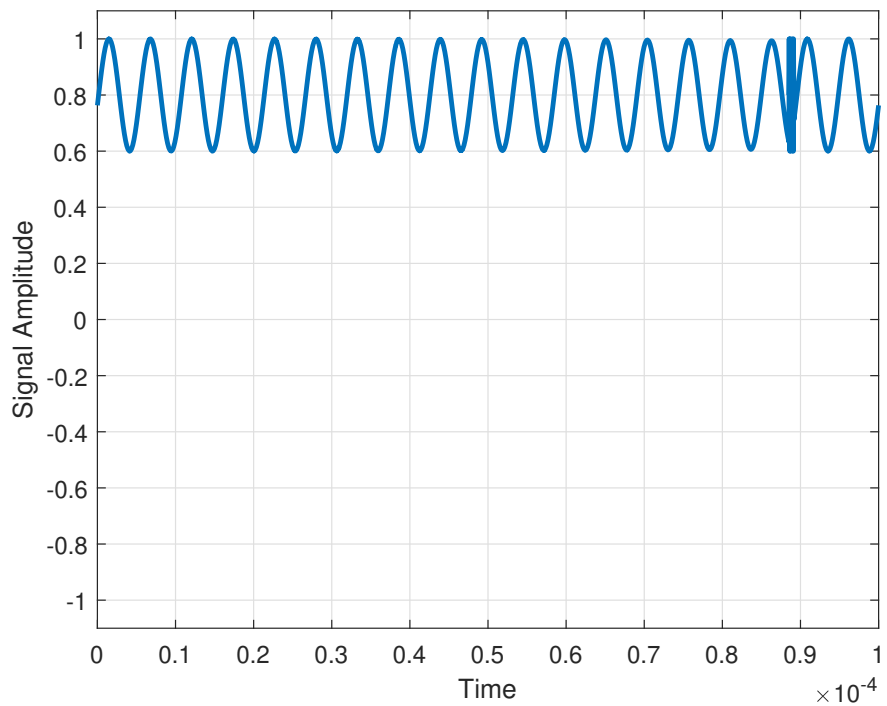


Figure 2.15: Example conjugate product with antenna crosstalk at a 4:1 crosstalk to signal ratio.

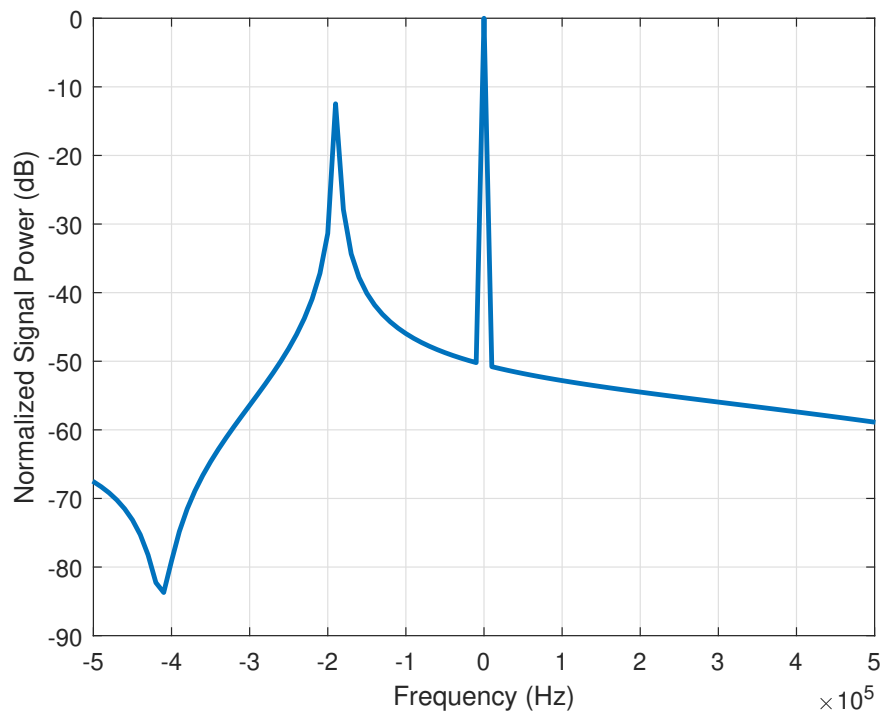


Figure 2.16: Example PSD of the same data from Fig. 2.15.

CHAPTER 3. GNU RADIO AND GR-RADAR

With the understanding of how the LFM CW radar works, thanks to the theory and simulation detailed in the previous chapter, the SDR is now considered. To program the SDR to function as the front end for a radar, it needs to transmit and receive simultaneously at a given carrier frequency. As such, the SDR programming method needs to be capable of configuring the SDR, controlling both the transmit and receive data streams simultaneously, and provide a way to execute the DSP. Within the HAM radio and amateur scene, the most common way to program and control SDRs is through GNU Radio Companion.

GNU Radio, formally known as GNU Radio Companion, is an open-source framework that allows users to program a wide variety of SDRs. This is accomplished through highly abstracted, interconnecting blocks that piece together to make a program. These blocks enable SDR programming without the need to know the underlying code. This way, the programmer can focus on the overall data flow without the need to worry about code level issues.

Within the GNU Radio community, many different toolboxes have been implemented over the years to fill various needs. Toolboxes, as discussed in more detail, contain many blocks for the user to use. The LimeSDR and gr-radar toolboxes are among these and provide the necessary building blocks to actualize the LFM CW radar as discussed in the previous chapter on the LimeSDR-Mini. Step-by-step instructions on how to install GNU Radio and the necessary toolboxes are detailed in Appendix C.

3.1 GNU Radio Companion

Since GNU Radio is designed to support a wide variety of SDRs, the flow is quite generalized, only varying in the complexity of the individual blocks that are used. A program within GNU Radio is called a flow graph. The flow graph is situated within the canvas which is represented by a large white rectangle in the main viewing window as shown in Fig. 3.1. Above the canvas

is a toolbar containing useful buttons and to the right are the currently installed toolboxes with their corresponding blocks. Below the canvas is either a variable viewing window for convenience (default) or the GNU Radio output terminal. If GNU Radio was launched using the `pybombs run gnuradio-companion` command, the launching terminal mirrors the GNU Radio output terminal.

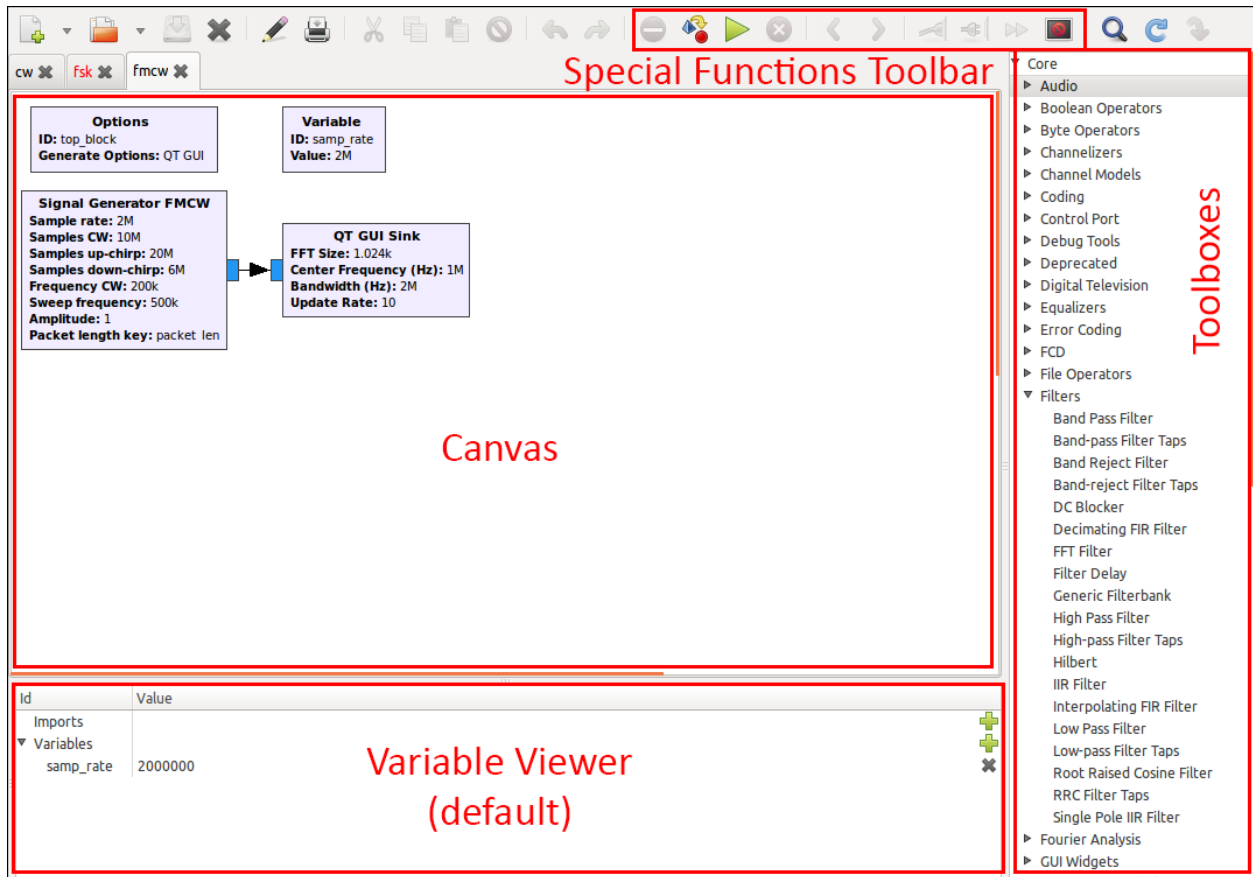


Figure 3.1: Full window view of GNU Radio Companion with simple flow graph example. Key areas of the GUI are outlined and labeled.

3.1.1 Toolboxes and Blocks

All of the blocks for creating a flow graph are located within various toolboxes. The default toolbox, otherwise known as the core toolbox, houses the majority of the blocks used in any flow graph. These blocks range from data converters and decimators to GUI based data viewers. Additional toolboxes can be installed and used within GNU Radio to hone and specialize a flow

graph's capabilities through specialized blocks such as the one shown in Fig. 3.2. The LimeSuite and gr-radar toolboxes are additional toolboxes added to supplement this work.

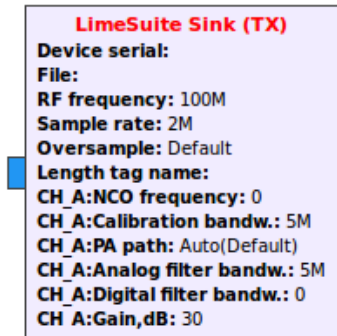


Figure 3.2: Visualization of the transmit block from the LimeSuite toolbox. Here it is depicted without any connections and none of the parameters, save the default ones, are provided. As such, the block is throwing an error, which is visible from the red title.

When placed on the canvas, a block by default has its inputs on the left and outputs on the right. Additionally, the color of the input or output node communicates the data type the block is expecting. The colors and corresponding data types are shown in Fig. 3.3. Note that the complex data types are interleaved real and complex values. For example, a complex 32 bit float has the first real sample (32 bits), followed by the first imaginary sample (32 bits), and so on. Some blocks, where flexibility is necessary, the inputs and outputs can support multiple data types. To do this, the block's properties need to be changed accordingly.

The properties of any block can be accessed by double clicking said block or right clicking it and then selecting properties at the bottom of the action menu. Within the properties dialog, there are multiple tabs with at least one tab for options and another for documentation as shown in Fig. 3.4. Unfortunately, often times the documentation tab is not helpful and shows relatively cryptic pseudo-code of what the function does.

LimeSuite

The LimeSuite toolbox is provided by MyriadRF to enable users to interact with their SDRs (LimeSDR-Mini and LimeSDR-USB) through GNU Radio. The toolbox consists of only two blocks which are the receive and transmit functions. These allow the user to either push data

Complex Float 64
Complex Float 32
Complex Integer 64
Complex Integer 32
Complex Integer 16
Complex Integer 8
Float 64
Float 32
Integer 64
Integer 32
Integer 16
Integer 8
Message Queue
Async Message
Bus Connection
Wildcard

Figure 3.3: The data types and their corresponding colors used within GNU Radio.

into, or read data from, the SDR. Since all MyriadRF SDRs are full-duplexed, both blocks can be used simultaneously in a single flow graph.

Which SDR is being used is important as there are some key differences between the LimeSDR-Mini and LimeSDR-USB. One notable difference is that the LimeSDR-Mini does not support multiple in multiple out (MIMO) operation so only channel A is usable. With the LimeSDR-USB, however, either channel A or B can be used for single in single out (SISO) operation, or both can be used for MIMO applications. Further explanation of the differences between the LimeSDR-Mini and -USB are not discussed here as this work utilizes the LimeSDR-Mini. For the details concerning why the LimeSDR-Mini is used over the LimeSDR-USB, refer to Appendix A.

gr-radar

The gr-radar toolbox was developed by Stefan Wunsch at the Karlsruhe Institute of Technology over six years ago. The age of this toolbox and the fact that it does not seem to be in active support anymore (last push to the git repository was March 8, 2018) is the reason that Ubuntu 16.04 LTS is needed to successfully install this toolbox, as stated in Appendix C. Despite its age, this toolbox affords a number of useful radar related blocks for GNU Radio applications. Within the toolbox, there are five subsections.

1. **Estimators** - Estimators are blocks that handle the majority of the DSP necessary to calculate the velocity and/or distance of an object based on the transmitted and received signals of the

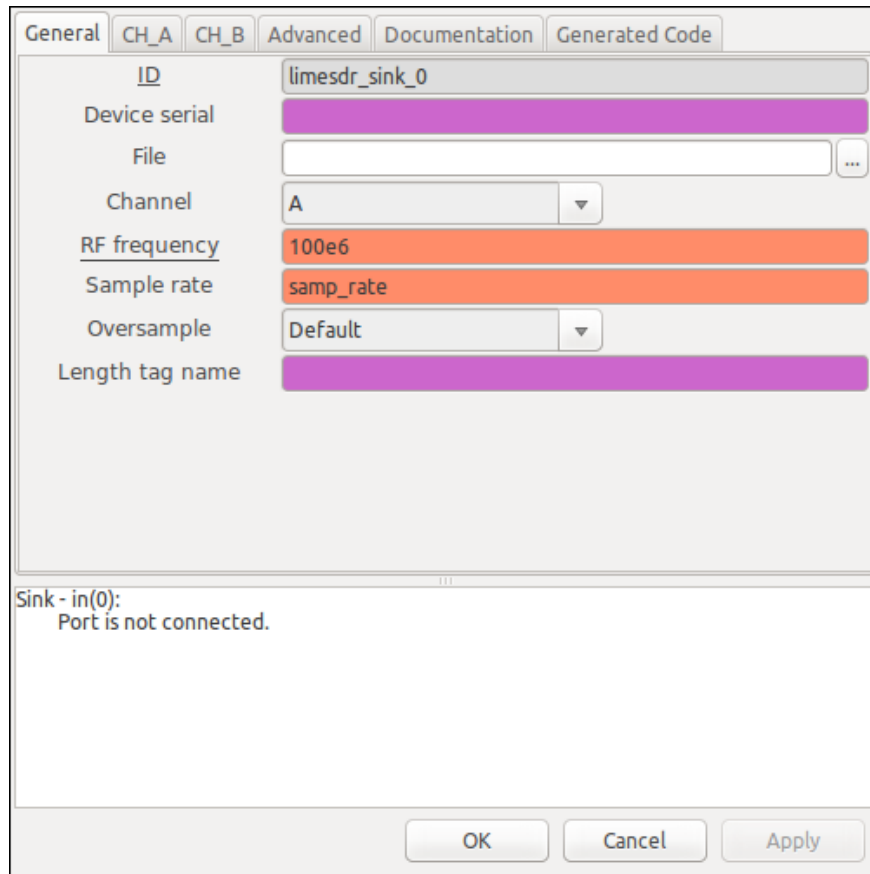


Figure 3.4: Properties dialog for the LimeSuite transmit block. The options tabs include general, CH.A, CH.B, and advanced. The documentation tabs include documentation and generated code.

flow graph. There are different estimators for different radar techniques, such as CW and FMCW.

2. **Generators** - It can be difficult to generate the necessary signals for radar transmission using the default blocks afforded by GNU Radio. As such, the gr-radar toolbox provides blocks to generate the exact signals needed for a few typical radars. One such generator is the FMCW, or as we have been calling it in this work, the LFMCW explored in this thesis.
3. **GUI** - This toolbox provides a few data visualization blocks. I found them less than useful as the built in GNU Radio GUI data sinks were more than adequate.
4. **RADAR** - Surveillance radars require more sophisticated DSP methods such as the ordered-statistic constant false-alarm rate (OS-CFAR) detector. The blocks in this subsection of

the toolbox were not explored as they pertain primarily to radars more complex than the LFM CW radar.

5. **Tools** - Various other tools are provided such as frequency-shift key (FSK) signal splitting and maximum peak search. In general the blocks provided here are not used since they are often implemented in the higher level blocks such as the estimators.

By utilizing the blocks provided by the core, LimeSuite, and gr-radar toolboxes, a full radar flow graph can be implemented. This has been demonstrated by Luigi Freitas who built a CW/Doppler radar (the least complicated radar architecture) using the LimeSDR-Mini and GNU Radio [12].

3.1.2 Toolbar Functions

The toolbar at the top of the GNU Radio window affords a number of useful functions, aside from the typical buttons such as new file and save. The buttons are discussed from left to right, according to Fig. 3.5.



Figure 3.5: View of special GNU Radio toolbar functions found at the top of the main program window.

1. **View errors** - Blocks that currently have errors are indicated with the name of the block in red text. Often this is not very helpful as it does not inform the user what the error is. This button on the toolbar compiles the errors on the whole flow graph and reports details about the corresponding errors. Warnings, such as sample rate discrepancies, are not included in this function.
2. **Generate code** - Theoretically when a flow graph has proven that it performs all of the necessary tasks, the flow graph is converted into code that can be refined and set to run outside of GNU Radio. The generate code button does just that. When pressed, a Python

script is compiled from the existing flow graph. The export location for the file is the same location of the flow graph file. If the location is unknown, the console printout from this operation details the file location.

3. **Execute** - The execute button allows the user to run the flow graph, performing all of the functions detailed by the blocks. This includes GUI data viewers as well as transmitting and/or receiving through physical or simulated transceivers.
4. **Kill** - Since flow graphs are different from typical code and scripts in that they do not have a concluding statement, the kill button is used to cease execution. This also works when the flow graph has logical errors and the execution locks up. It does not work, however, when GNU Radio itself becomes unresponsive.
5. **Rotate (CCW/CW)** - Often times when laying out blocks for a flow graph the sequence of blocks can be quite long, necessitating a wrap around. This is due to the fact that the canvas for a flow graph does not extend infinitely in the horizontal or vertical axes. In order to have a cleaner flow graph, visually, rotating the blocks is sometimes desirable.
6. **Enable/disable block** - Akin to commenting out blocks of code in C++ or other languages, blocks can be disabled. When a block becomes disabled, it retains its connections to other blocks, but turns gray indicating that it is no longer part of the signal pipeline. This can cause blocks downstream to throw errors as there is now a portion of the flow missing. Enabling a block causes it to return to the active use within the flow graph.
7. **Bypass block** - Similarly to enabling or disabling a block within the flow graph, a block can be bypassed. A bypassed block is shown as yellow. This is similar to disabling the block, but instead of simply deactivating it, it connects its input to its output, effectively sending the data around the block. Either enabling or disabling the block changes the block's working state.
8. **Toggle visibility** - When many blocks are disabled, it can often be beneficial to hide them to see more clearly which blocks are currently functioning as part of the flow graph. This button allows the user to toggle the visibility of these blocks. Bypassed blocks are not hidden.

3.2 Flow Graph Topology

After creating a new flow graph in GNU Radio, the user is presented with a pristine canvas with two blocks. The first is titled “Options” and the second “Variable.” The options block contains the parameters for the flow graph options and meta data and is the only block that cannot be deleted. Here the title and author of the flow graph can be recorded as well as changing the size of the canvas. In addition to these, the rendering engine is selected here. In general, QT is used for Linux systems and WX is used for Windows. The different renderers determine which GUI sink must be used (e.g., QT or WX variant).

The second block present on the new flow graph is a variable block. Since GNU Radio uses blocks instead of code, variables are represented as blocks. This, just like in typical programming languages, allows the user to use a value in multiple places by using the variable instead of a number. When the number of variables grows, it can be difficult to find a singular variable block to change its value. For this reason, the variable viewer at the bottom of the GNU Radio window is useful. In the viewer, it is possible to change values and even add new variable blocks.

To add more blocks to a custom flow graph, blocks can be dragged from or double clicked in the right-hand window. Since there are so many blocks, it can be difficult to find a specific one. To remedy this, hitting CTRL+F opens a search bar that searches all of the installed toolboxes. After a few blocks are added to the canvas, they can be connected by clicking the output of one block and then the input of the next. This creates a connection denoted by an arrow. Note that GNU Radio allows connections between ports of non-matching data types though this does not work. In this case a red arrow indicates the error.

Fig. 3.6 shows an example flow graph that works as a frequency shift keying (FSK) radar. The FSK radar is more complicated than the CW and LFM CW radars but is explored to detail the theoretical complexity GNU Radio can provide. The blocks are arranged to cleanly show the data flow from signal generator to estimator. In the figure, below the options block, the signal is generated and then moves into a throttle block. Following the arrows, the general flow of a radar is apparent. The signal passes into the channel, which is the LimeSuite transmit followed by receive blocks, as well as passing straight to the conjugate multiplication block. The conjugate multiplication block receives both the signal from the generator as well as the signal from the channel. The remainder of the blocks and data path is for GNU Radio sample rate regulation as

well as other DSP techniques specific to FSK. In addition to these blocks, there are a number of gray blocks which are disabled.

3.3 Pros and Cons

GNU Radio as a platform for programming the LimeSDR-Mini provides a number of key benefits but has some significant limitations.

The pros are:

- **Flow graphs** - The block structured nature of GNU Radio creates an environment that enables quick development and testing. This is done through removing much of the guesswork and minor logical/syntactic errors that often accompanies programming. These two factors alone often bloat code-based development time. In addition to this, it is easy to see the data path in a given flow graph since the arrows move through the canvas in the same direction as the data.
- **Community open-source** - Since GNU Radio is a community, it is constantly being updated and help is often readily available on the forums when needed. Older blocks and toolboxes may lack support. It is also possible to open the source code for any given block and modify it if necessary. The underlying blocks are all coded in C++, despite the flow graphs exporting in Python.
- **Deployment** - When all design and programming is done, it is very simple to export the flow graph as a standalone Python script. That script can then be deployed to any device that has the necessary libraries. This means that it can be set to run automatically on the CubeSAT without human intervention.

Unfortunately, GNU Radio has quite a few shortcomings as well.

- **Flow graphs** - Since GNU Radio attempts to generalize all SDR operation into blocks that can be picked and matched to fit any need, there are quite a few blocks. Of course this can be remedied with time, but becoming familiar with even a number of useful blocks can become quite cumbersome since the documentation tab within most blocks are poorly

maintained/written. Instead, to learn what a block does, it is often necessary to find the block on the GNU Radio or corresponding toolbox website and read the full documentation, which is often inadequate.

- **Community open-source** - Within the GNU Radio core toolbox, multiple coding standards are present as well as varying levels of documentation. This makes it quite difficult to learn how the different blocks can fit together. Then from there, adding toolboxes from other sources, such as gr-radar, introduce the problem of upkeep and compatibility. Since the gr-radar toolbox has not been updated since 2018, it is easy to say that the future of any project that builds on it is similarly unable to be updated beyond a certain point. In the case of this work, being limited to Ubuntu 16.04 LTS is a large problem for future expandability.
- **Deployment** - As stated above, deployment is simple as long as the target platform has the necessary libraries. Unfortunately, many of the libraries that are needed to run a script generated by GNU Radio are only available through the installation of GNU Radio. This places higher processor and memory requirements on the target platform, often increasing price and power consumption.
- **Timing control** - The block structure obfuscates much of the underlying operation of the program. Often when programs are built to be generalized, the code tends to bloat with if/else and try/catch statements. These additional lines of code can increase the data path time to and from the channel. In typical SDR applications this is not an issue; if the data is transmit a microsecond later, the receiver on the other end knows no differently. In radars, however, one microsecond translates to approximately 150 meters difference, according to $d = \frac{ct}{2}$.
- **Block modification/addition** - The block methodology of GNU Radio heavily implements class inheritance and hierarchies. In the case that a block needs to be modified/created, said inheritances and hierarchies need to be understood to write the new block such that it fulfills the requirements of a child class as well as not duplicating any existing functions. Doing so can cause the flow graph to not function correctly or crash GNU Radio altogether.

3.4 GNU Radio Contribution

GNU Radio provides a useful starting platform to understand the software needs of this work. Even though it has many drawbacks, it provides a means to test the LimeSDR-Mini in many capacities, most notably its full-duplex operation. Based on the needs of this work, GNU Radio is not recommended as a topic of further research due to its many drawbacks. The fact that this work found the benefits to prove detrimental when inspected more closely, GNU Radio is found to be inadequate for this work.

The majority of this information was gleaned from trial and error over a long period of time through many different flow graphs. Appendix D documents a number of the more successful and insightful flow graphs.

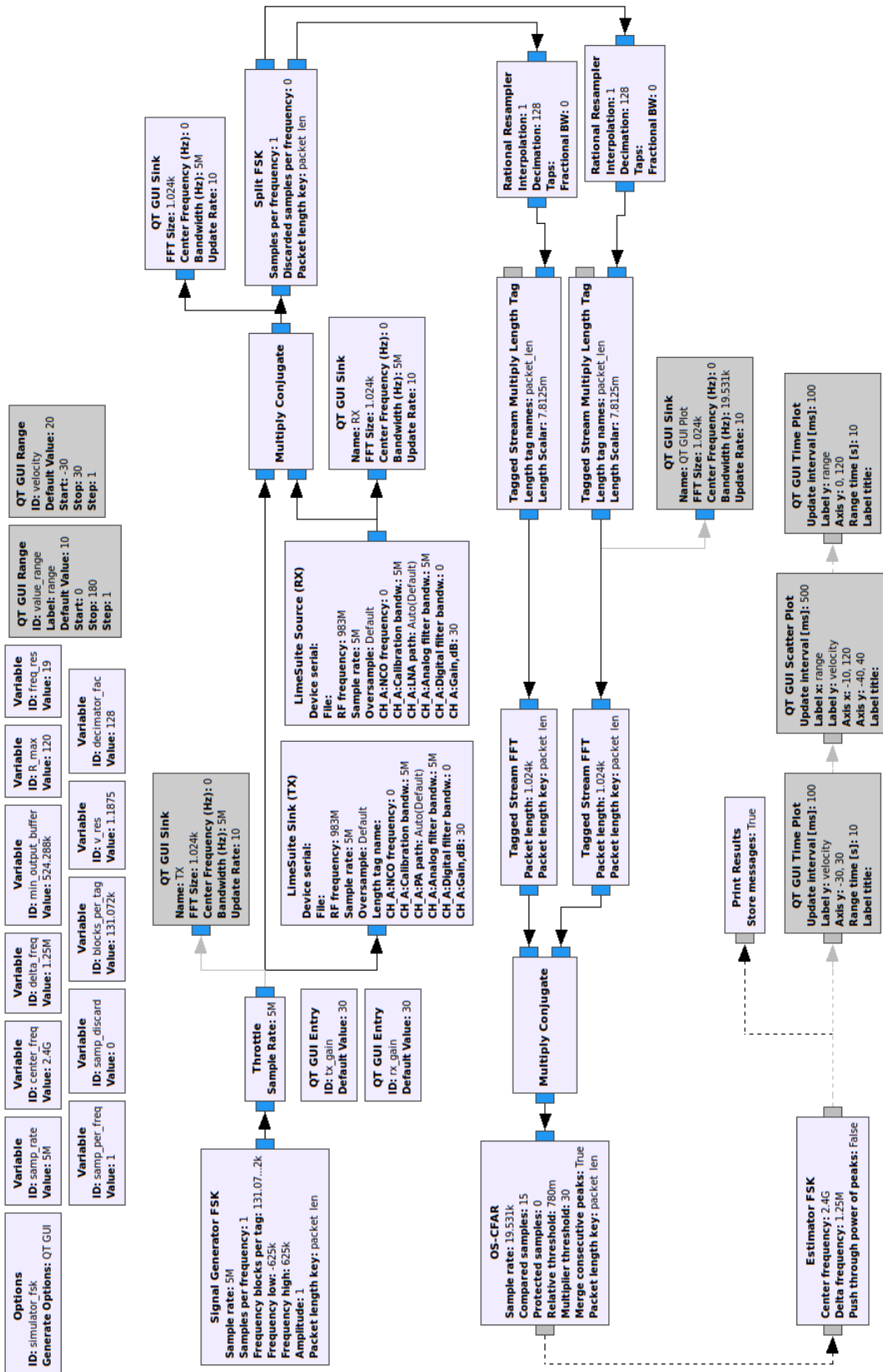


Figure 3.6: Example flow graph of an FSK radar provided by the gr-radar toolbox with the simulated transmit and receive pair replaced with the LimeSuite transmit and receive blocks, respectively.

CHAPTER 4. LIMESUITE GUI AND PYTHON3 API

In this chapter, another approach to programming the LimeSDR-Mini is explored since GNU Radio Companion was found inadequate for this work. MyriadRF, as part of their open-source framework, provide an application programming interface (API) alongside their hardware. An API is a core component of many systems as it allows the code to access hardware level features with little overhead compared to other generalized solutions. As stated in the previous chapter, the data path within a GNU Radio flow graph can introduce significant latency due to the generalized and abstracted nature of the software. An API has the distinct advantage in this regard as it is closely designed to the hardware for which it is intended. This allows the code to run quickly and efficiently by excluding unnecessary checks often found in generalized solutions.

4.1 LimeSDR-Mini Hardware

When GNU Radio is used in a system, a deeper understanding of the underlying hardware and components is often unneeded as the software, with its prebuilt blocks, abstracts much of the details. An API, on the other hand, requires the user to understand the hardware to achieve the desired outcomes, but provides better flexibility in dealing with the hardware. The understanding of the LimeSDR-Mini begins with the board level design. This design is represented by a high level block diagram in Fig. 4.1.

Among the blocks in Fig. 4.1, the one of most interest to this work is the LMS7002M (RF transceiver). This is because the LMS7002M is responsible for controlling radio transmit and receive aspects of the SDR. This operation is quite similar to microcontroller functionality. In microcontrollers, code is loaded into memory within the chip and when the code is executed, it modifies different register blocks. The registers are buffered to different GPIO which control whatever external circuitry designed by the engineer. In the LMS7002M, these registers are connected to RF circuitry also contained within the chip which modify the different aspects of the transmit

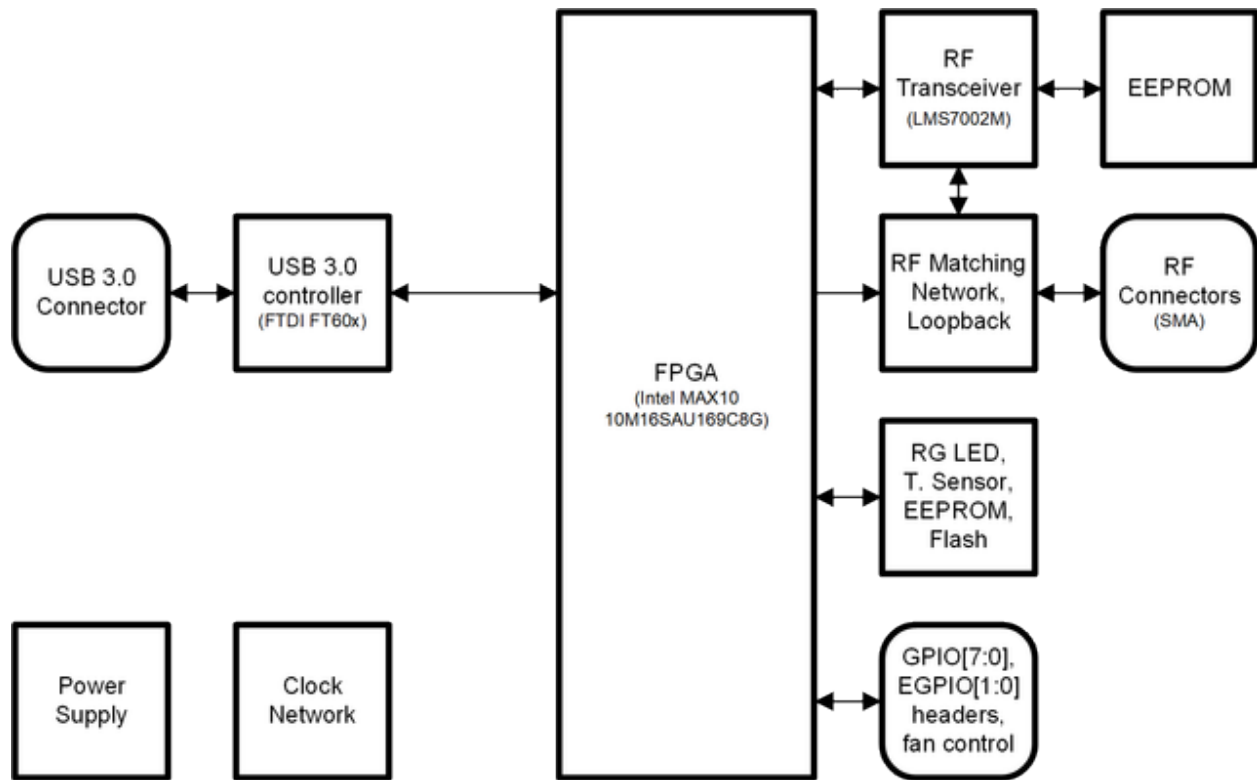


Figure 4.1: High level block diagram of the LimeSDR-Mini hardware. The notable components are the FPGA (Intel MAX 10) and RF transceiver (LMS7002M).

or receive behavior. Even though many of the other blocks in Fig. 4.1 are configurable, they are not explored in this work as they have little effect on the SDR's transceiver operation and are not supported directly by the API.

By looking at the block diagram of the inner workings of the LMS7002M, shown in Fig. 4.2, we glean a better understanding of what aspects of the chip are programmable. Every box and symbol in the block diagram is configurable. Upon inspection it is clear that the diagram is split into a receive chain (top) and transmit chain (bottom). These are split into separate figures, Fig. 4.3 and Fig. 4.4 respectively. Within the receive and transmit sections, it is clear that the same blocks/hardware are duplicated. This is to enable multi-channel or MIMO operation. In addition to these sections, there are three large rectangles on the right-hand side of the diagram labelled transceiver signal processor and LimeLight digital IQ interface ports 1 and 2. The transceiver signal processor (TSP) is responsible for conditioning the data according to user requirements. These options include sample interpolation/decimation, digital filtering, phase and gain correction, and

stage bypassing. The LimeLight digital IQ interface is a data streaming interface between the user and the LMS7002M which complies with JEDEC industry standards. The JEDEC standards used here are not explored.

Within a single data receive path, as shown in Fig. 4.3, many of the same aspects described in the simulation chapter can be observed. Each of the components in Fig. 2.3 can also be found in Fig. 4.3. These include the local oscillator/synthesizer, IQ mixer, filtering, and data output/ADC. The LMS7002M diagram differs in its addition the low noise amplifier (LNA), trans-impedance amplifier (TIA), and programmable gain amplifier (PGA). These three amplifiers are included to resolve signal degradation issues, such as noise and attenuation, that accompany real systems. The “RF RSSI” block is the receive signal strength indicator (RSSI) and is used for automatic gain control to maximize the receive signal’s signal to noise ratio (SNR).

The transmit data path, depicted in Fig. 4.4, can similarly be compared to Fig. 2.2 in chapter 2. Again, each of the components in Fig. 4.4 are visible in Fig. 2.2 with the addition of a low pass filter (LPF) and power amplifier driver (PAD). The LPF is to remove undesirable harmonics and aliasing that may arise from the TSP and sampling. The PAD is included to amplify the signal power preparatory to driving an external antenna. In an ideal system, such as the one explored in chapter 2, these components are not required.

Throughout the block diagram there are various arrows that come off the receive and transmit data paths. They include SDR digital interface bypasses and three loopback paths. The digital interface bypasses are located just after the DACs and before the ADCs. With proper configuration, it is possible to transmit and receive custom analog IQ signals to and from these ports instead of using the TSP and LimeLight interfaces. This work does not explore this option as doing so would require more hardware and development which has already been done and placed on the board by MyriadRF. The loopback paths allow signal verification at different stages within data path. The different loopbacks are as follows:

1. **Digital loopback (DLB)** - The transmit signal is fed from the LimeLight transmit port (port 1 or 2 depending on configuration) directly to the LimeLight receive port (port 2 or 1). At this point, the signal is a pair of digital signals (IQ).

2. **Basband loopback (BBLB)** - The transmit signal after the LPF is fed back to the input of the LPF on the receive side. The signal at this stage is a pair of analog signals (IQ) at baseband.
3. **RF loopback (RFLB)** - The transmit signal is fed from the output of the PAD to the input of the receiver LNA. The signal at this point is similar to the signal in the BB loopback, but has been mixed to the configured carrier frequency.

The microcontroller, which is found between the transmit and receive portions of the block diagram, is responsible for controlling many of the fine tuning. Fortunately, through the LimeSuite GUI and API, an internal working of the registers and microcontroller are not necessary as there are higher level buttons/function calls to automatically configure these options.

4.2 Programming the LMS7002M FPRF

A basic understanding of the different components within the LMS7002M FPRF is necessary as non GNU Radio programming methods require deeper familiarity with the hardware in order to achieve desired functionality. MyriadRF supports two different methods of programming the LMS7002M, and by extension the LimeSDR-Mini, besides GNU Radio. One method is to use the LimeSuite GUI and the other is to use the pyLMS7002Soapy API. Comparing the different configuration tabs from the LimeSuite GUI, shown in Fig. 4.5 and libraries within the pyLMS7002Soapy API, shown in Fig. 4.6. Since the majority, if not all, of the SDR's configurability comes from setting the LMS7002M FPRF, programming/configuration methods discussed here extend not only to the LimeSDR-Mini, but to any SDR that utilizes the LMS7002M.

By comparing the tabs from the LimeSuite GUI and libraries within the pyLMS7002Soapy API, it is quickly apparent that the two are closely related. As such, it is assumed within this work that the LimeSuite GUI is programmed to merely be a graphical interface for the API. In this way the LimeSuite GUI differs from GNU Radio as GNU Radio implements more functionality and checks in order to support a wider variety of SDRs, whereas the LimeSuite GUI is built on top of an API targeted toward a single FPRF. This keeps the code base smaller and execution time faster.

This work explores non GNU Radio programming methods generally by experimenting with configurations within the LimeSuite GUI and then implementing them in a Python script using the pyLMS7002Soapy API. The LimeSuite GUI and pyLMS7002Soapy API are both running on a

Raspberry Pi running the custom PiSDR image. Installation and verification instructions are found in Appendix E.

4.2.1 LimeSuite GUI

After launching the LimeSuite GUI, the user is presented with a window similar to Fig. 4.7. a number of basic functions of the LimeSuite GUI need to be understood in order to explore the different capabilities of the LimeSDR-Mini as a radar. The basic functions include connecting an SDR to the software, setting the RF (transmit and receive) frequencies, calibrating the receive and transmit hardware, and handling receive and transmit streams. The LimeSuite GUI offers significantly more options than these, but only these have been explored thus far.

Connecting a Radio

To connect the radio to the LimeSuite GUI, select “Connection Settings” from the options drop down menu. The following dialog window displays all of the currently connected compatible radios, as shown in Fig. 4.8. Select the desired radio and click “Connect.” Afterward, the console at the bottom of the main window displays information about the newly connected SDR.

Once a radio is connected, it is not possible to determine which radio is connected through the GUI, nor is it possible to see which radio (if multiple are available) is connected or disconnected. For this reason it is recommended to use only one radio at a time.

Setting RF Frequency

The SXR and SXT tabs in the LimeSuite GUI correspond to the receive and transmit synthesizers, as outlined by Appendix F. These synthesizers are responsible for generating the frequencies used for the receive and transmit IQ mixers, as shown in Figs. 4.3 and 4.4. There are two different synthesizers to control each mixer in the case that the receive and transmit hardware are operating at different carrier frequencies.

Within both the SXR and SXT tabs, the subsection indicated by Fig. 4.9 allows the user to modify the receive or transmit carrier frequency. To set, type the desired carrier frequency into

the text box and then click the calculate button. The calculate button calculates the necessary configurations to achieve the target frequency and then sets the hardware accordingly.

Both the receive and transmit synthesizer frequencies need to be set using the described method. In theory, there is a way to set the SDR to TDD mode so that the receive mixer uses the carrier signal generated by the transmit synthesizer, however, as of yet it is not clear where to set this in the GUI.

Calibrating RX and TX Hardware

To calibrate, the SDR uses calibration parameters set in the on-chip microcontroller to remove non-idealities in the transmit and receive signals. This process fine tunes a number of parameters throughout the SDR according to newly updated settings, such as synthesizer frequencies and configurations. Calibration is done from the calibration tab by pressing either the receive or transmit calibration button, or both by pressing the calibrate all button to the right.

This process often fails as some little configuration goes awry. The resulting error message indicates which part of the SDR failed allowing the user to navigate to the corresponding tab and fix the error. Another solution is to press the “default” button at the top of the GUI and then entering the desired configurations again. Often is the case that some parameter is automatically configured causing the calibration error. By reverting to default configurations, such parameters should be corrected.

More advanced calibration techniques are available through the MR3 tab (microcontroller calibration).

RX and TX Streams

The last essential aspect to exploring the LimeSuite GUI is streaming custom data to and from the SDR. The onboard FPGA is responsible for storing the transmit data and feeding it into the LimeLight digital IQ interface. It is also responsible for retrieving the receive data from the LimeLight digital IQ interface and routing it to the connected device through the USB bridge controller.

First, a custom waveform needs to be created to comply with the hardware and LimeLight data requirements. A MatLab script to create a custom LFMCW signal according to the requirements is provided in Appendix G. To insert the waveform file into the SDR, select the “FPGA Controls” option from the modules drop down. This opens the FPGA controls dialog shown in Fig. 4.10. From this dialog, it is possible to source data from a stream and a waveform file. Streaming data is currently unexplored.

From here, use the folder button to navigate to and select the custom waveform file created by the MatLab script. Pressing the “Custom” button starts transmitting IQ samples from the selected waveform file. During operation, the play button is not selectable and instead the stop button is.

Currently it does not seem like it is possible to stream the received data samples out of the LimeSuite GUI. As such, the built in FFT viewer is needed to see the data being received. To open, select the FFT viewer from the modules drop down menu. An example of the FFT viewer running is shown in Fig. 4.11. After starting a waveform from the FPGA controls, press the start button in the FFT viewer to view the signal.

A spectrum analyzer is recommended for transmit signal inspection from the SDR and the FFT viewer to analyze receive signals. This way the user can be certain that the transmit signal is not being distorted by the receive hardware in some way.

4.2.2 LimeSDR Python3 API

While the LimeSuite GUI provides the capabilities desired for this work, it is not deployable. This is because the configurations within the LimeSDR-Mini are volatile; when the SDR power cycles, the configurations are reset. To deploy a system using the LimeSuite GUI, the system would need to launch the software on boot and somehow program in all of the configurations. As seen from the previous section, this would be quite difficult as programming takes significant menu traversal. The pyLMS7002Soapy API provided by MyriadRF is the solution as it is lightweight, provides full control over the SDR, and is deployable through script execution on boot.

The first task to writing code utilizing the API is understanding how it is organized and written. The API organization is made clear through the import tree is shown in Fig. 4.6. It also

shows the general level of complexity. Each library within the API is written as a class and as one library imports another, it includes the imported class. The `pyLMS7002Soapy.py` library is the highest library and contains the highest level functions. Calls to functions here trickle down through the other necessary libraries and variables. As the user access libraries deeper into the include tree, the functions become similarly lower level. By inspecting the `pyLMS7002Soapy.py` script (the library imported by a script to use the API), there are a few things to note that highlight how the API is written.

One, the library imports a library called `SoapySDR`. `SoapySDR` is a platform independent SDR support library intended to simplify many of the general aspects of interacting with an SDR. This may sound very similar to GNU Radio Companion, but it differs in that when `SoapySDR` is instantiated within the `pyLMS7002Soapy` class, an argument is passed saying that the connected SDR is a lime device. Within the codebase of `SoapySDR`, this is used to sidestep many of the generalized checks done by GNU Radio.

Second, the API is written using the `@property`. Due to the way Python functions, all variables in a class are public and can be called at any time. This can lead to users or other libraries/functions inadvertently changing a variable. In an environment as complex as the LimeSDR-Mini, this can be catastrophic if the variable is changed without updating other related variables. In languages such as Java and C++, such variables are made private and then supplementary getter and setter functions are implemented. In Python, the equivalent is the `@property` tag. The `@property` tag, when placed before a function, defines the function as a getter for the variable where the function name is the variable. The setter analog to this is the `@<var>.setter` tag. The benefit to using these tags is that whenever these variables are accessed, either through writing or reading, the corresponding getter or setter is called rather than directly modifying the variable. At the script level, to access the getter and setter functions, simply read or set them as if they are variables. An example script is found in Appendix H.

Connecting the LimeSDR

The API needs to be imported into the script first to connect the LimeSDR-Mini or -USB. This is done by typing `from pyLMS7002Soapy import pyLMS7002Soapy as pylmss`. This tells the script that it is importing the `pyLMS7002Soapy` class from the `pyLMS7002Soapy.py`

file and to give it the pseudonym pylmss. At this point the API is available and the SDR can be connected by typing `<desired-var-name>= pylmss.pyLMS7002Soapy()`. A successful connection is shown in Fig. 4.12. This line instantiates the class within the script for the first SDR connected by serial number's alphanumeric order. As long as the script is running, the connected SDR remains connected and cannot be accessed by another script or application.

Accessing Specific Libraries

In the case that configurations/functions deeper within the API are needed, they can be accessed using the dot operator. Fig. 4.13 shows an example of library traversal using the dot operator. In the example, all of the libraries shown in the dotted box in Fig. 4.6 are visible as well as the functions unique to the LMS7002 class.

Notable Features

As discussed in the LimeSuite GUI section, there are a few notable functions needed to fulfill this work's requirements. Aside from setting the receive and transmit carrier frequencies, bandwidths, gains, etc, which are all accessible through high level calls (i.e. from the pyLMS7002Soapy library), a few are mentioned here. In each of the of the items below, `lime` is the instantiated pyLMS7002Soapy class as shown in Fig. 4.12.

- **TDD Mode** - TDD mode allows the receive IQ mixer to use the signal from the transmit synthesizer, as discussed in the LimeSuite GUI section. Through the API, this is easily changed directly from the pyLMS7002Soapy. The code to do so is:

```
1 lime.tddMode = 1
```

- **Calibration** - The functions to calibrate the SDR can be called by accessing the calibration library. This can be done with:

```
1 lime.LMS7002.calibration.<function>
```

- **Data Streaming** - Since streaming data to and from the SDR is an interfacing operation, it is handled by the SoapySDR. The pyLMS7002Soapy script shows that the SoapySDR library is instantiated as sdr. To setup a data stream, the code is:

```
stream = lime.sdr.setupStream(pylmss.SOAPY_SDR_<RX or TX>,
                               pylmss.SOAPY_SDR_CF32, [0])
```

The first argument sets the stream as a receive or transmit stream and the second argument defines the expected datatype. In this case, SOAPY_SDR_CF32 is a 32 bit complex float.

4.3 LimeSuite GUI and API Contribution

The LimeSuite GUI provides a useful sandbox environment where different parameters and configurations can be explored. Once desirable a configuration is determined, it can be encoded in a Python script using the pyLMS7002Soapy API. This affords a lightweight (time-wise and computationally) approach to controlling the LimeSDR-Mini.

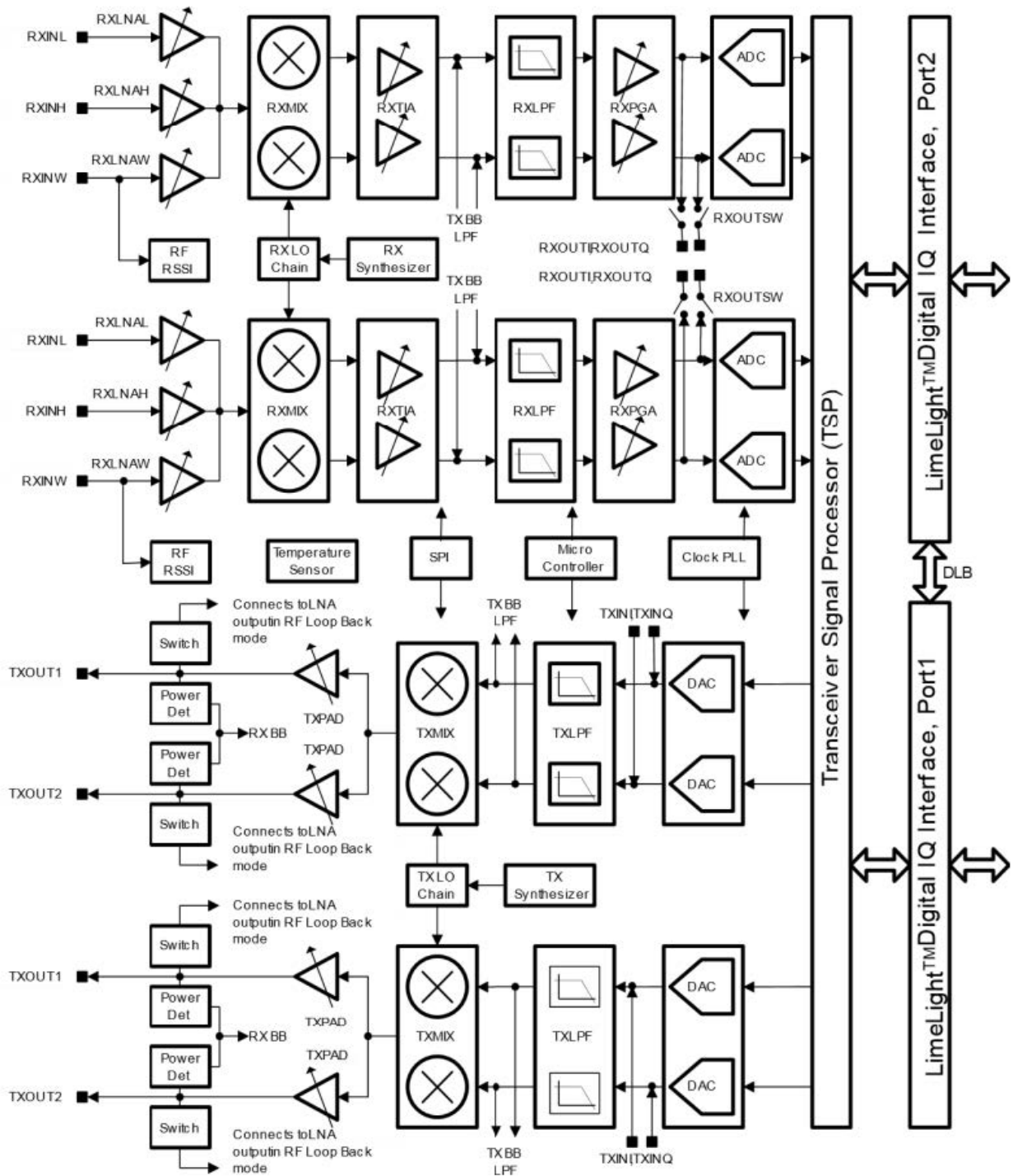


Figure 4.2: Block diagram of MyriadRF LMS7002M FPRF.

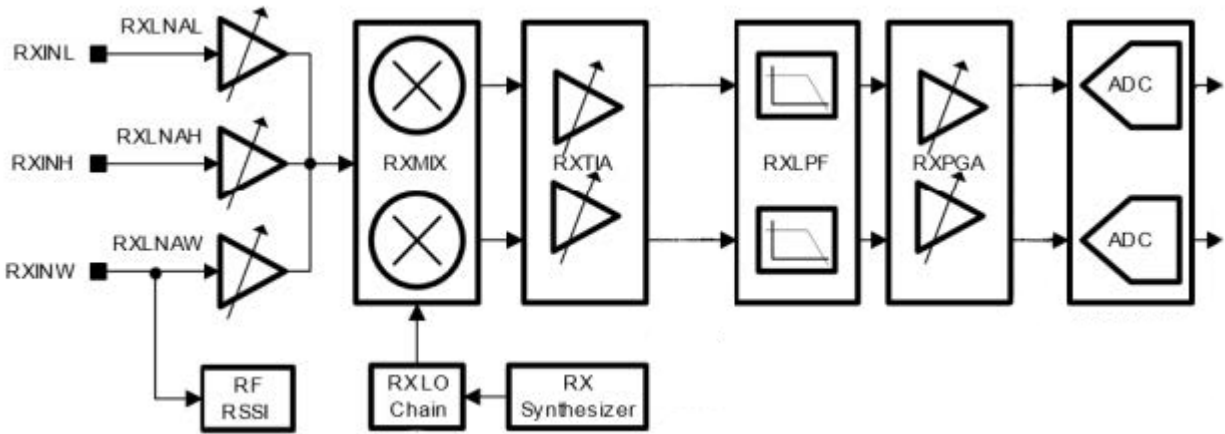


Figure 4.3: Focused view of the MyriadRF LMS7002M receive block diagram. Data flow is from left to right, from antenna to ADC output.

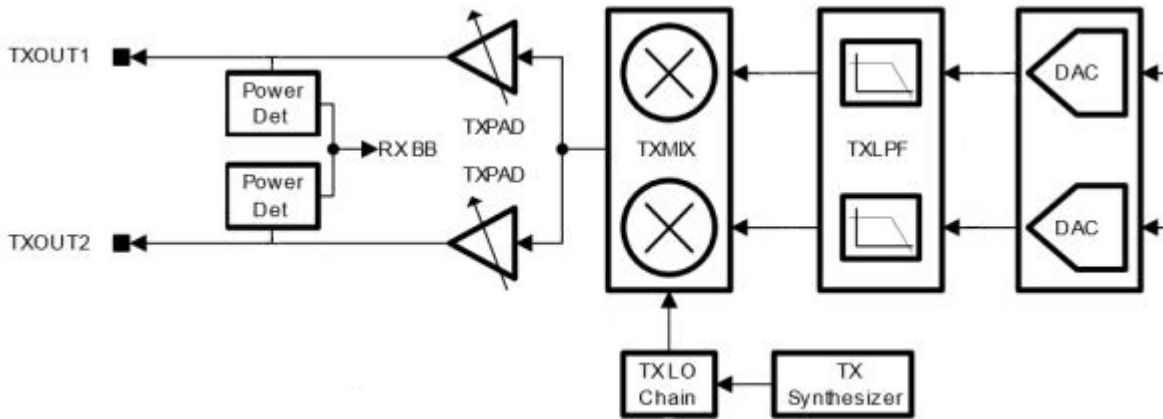


Figure 4.4: Focused view of the MyriadRF LMS7002M transmit block diagram. Data flow is from right to left, from DAC input to antenna.

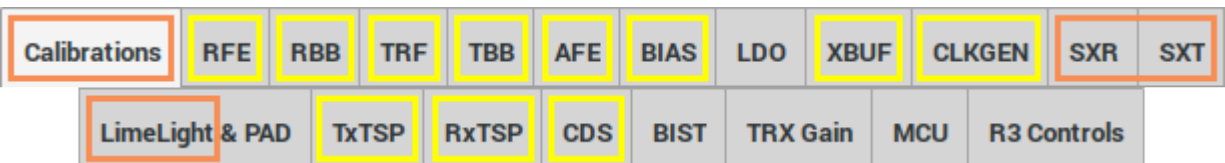


Figure 4.5: Screenshot of the LimeSuite GUI tabs with half of the tabs wrapped around for easier viewing. Tabs outlined with a yellow rectangle correspond directly to an API library and the orange tabs loosely correspond to one or more API libraries.

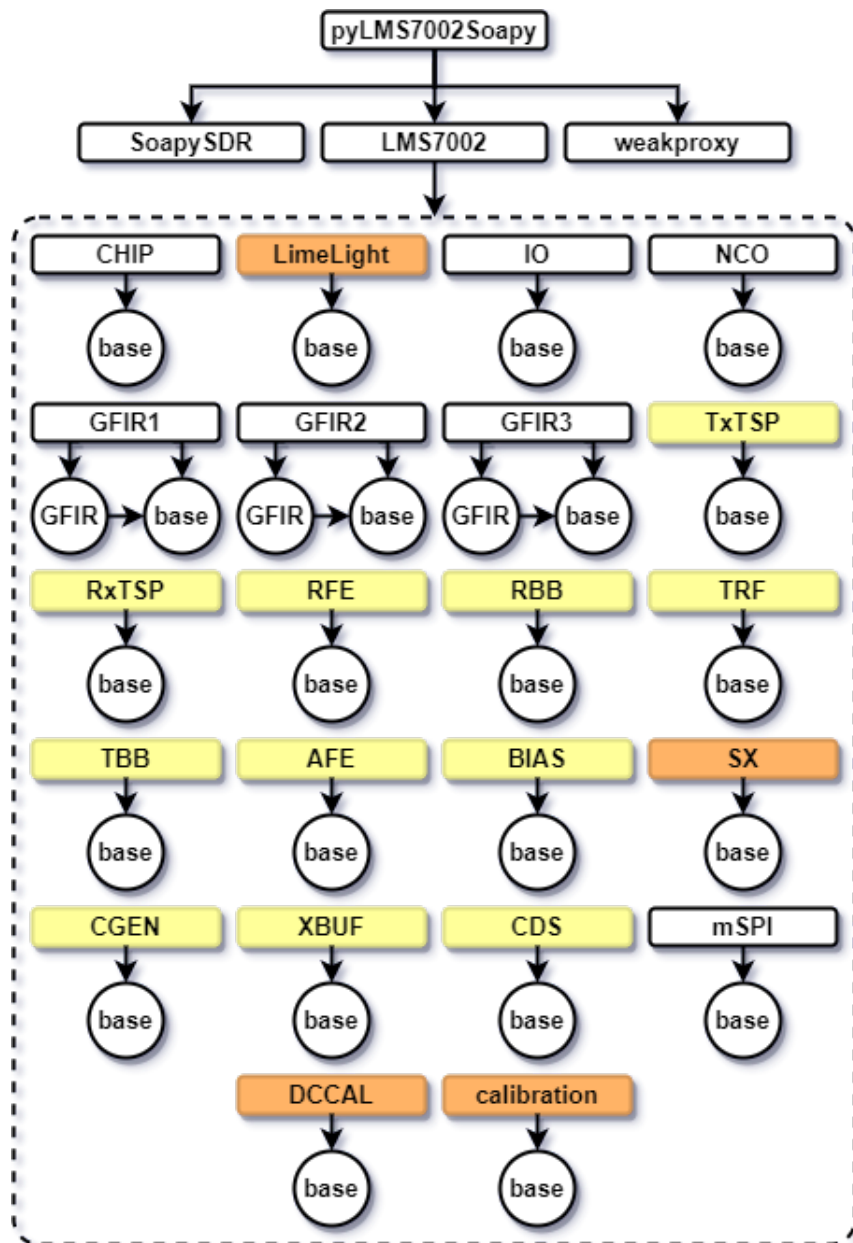


Figure 4.6: Block diagram of the API include structure. Each of the rectangular libraries within the dotted outline are individually imported by the LMS7002 library. The yellow libraries directly correspond to a LimeSuite GUI tab and the orange libraries loosely correspond to a LimeSuite GUI tab.

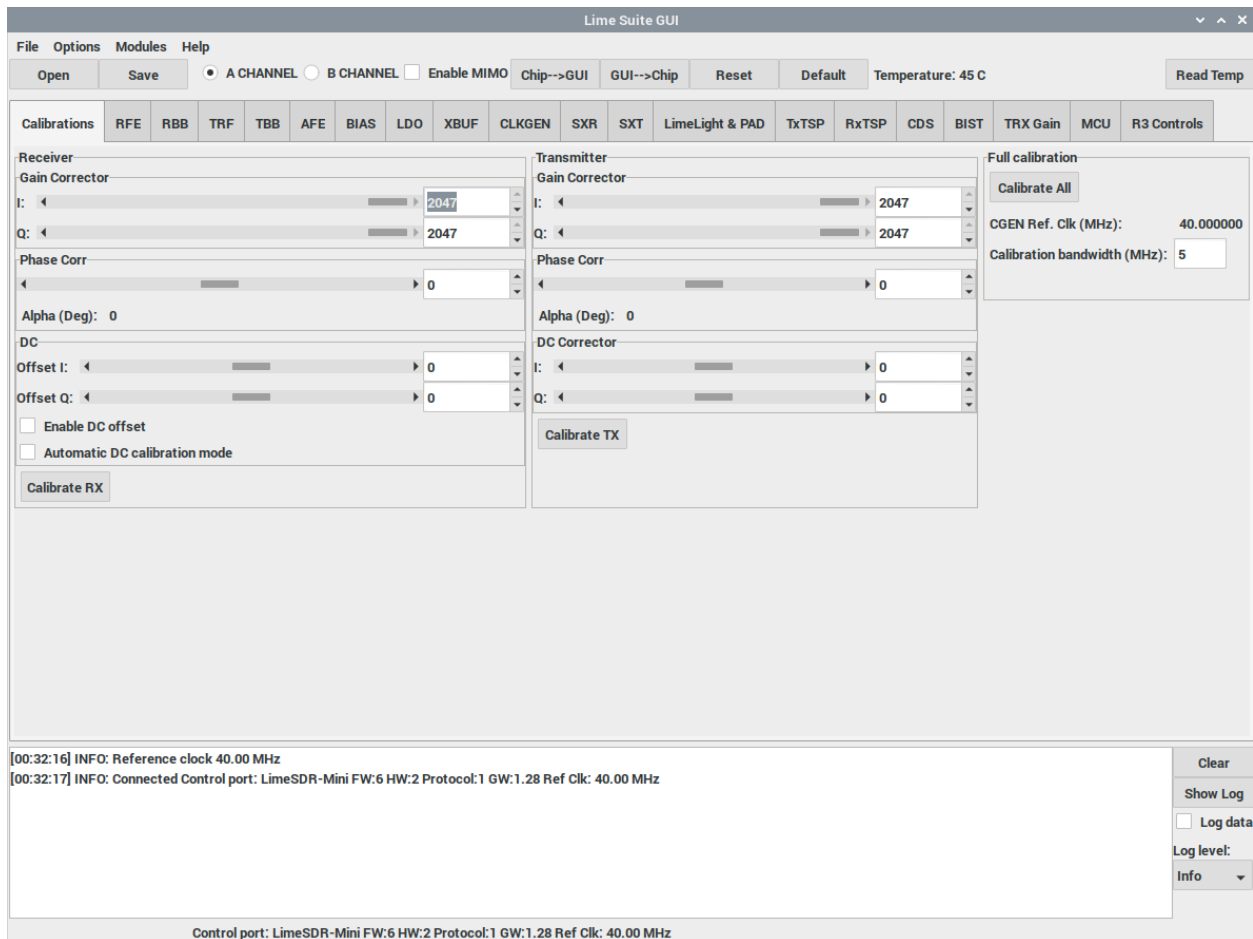


Figure 4.7: View of the LimeSuite GUI opened to the calibrations tab (default upon launch). The bottom console currently displays information about the newly connected SDR.

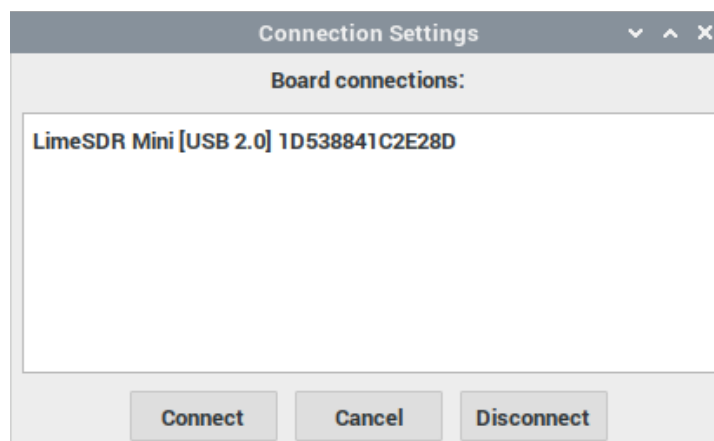


Figure 4.8: LimeSuite GUI radio selection dialog to connect/disconnect an SDR.

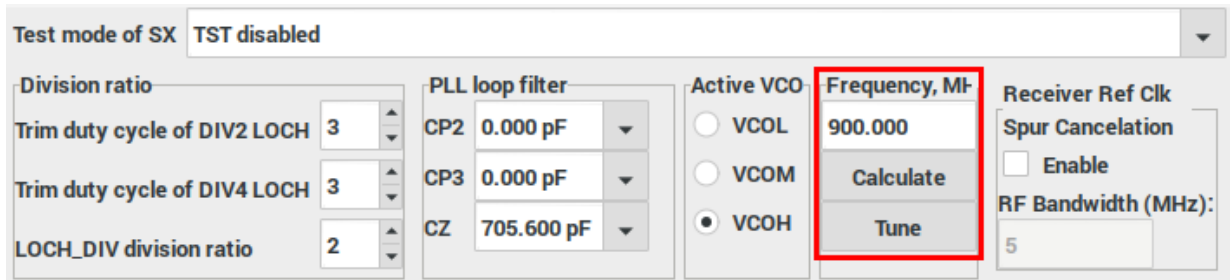


Figure 4.9: Subsection of the SXR (SXT has similar options) tab indicating the carrier frequency selection textbox and tuning button.



Figure 4.10: LimeSuite GUI FPGA controls dialog window used to feed data into the transmit portion of the SDR.

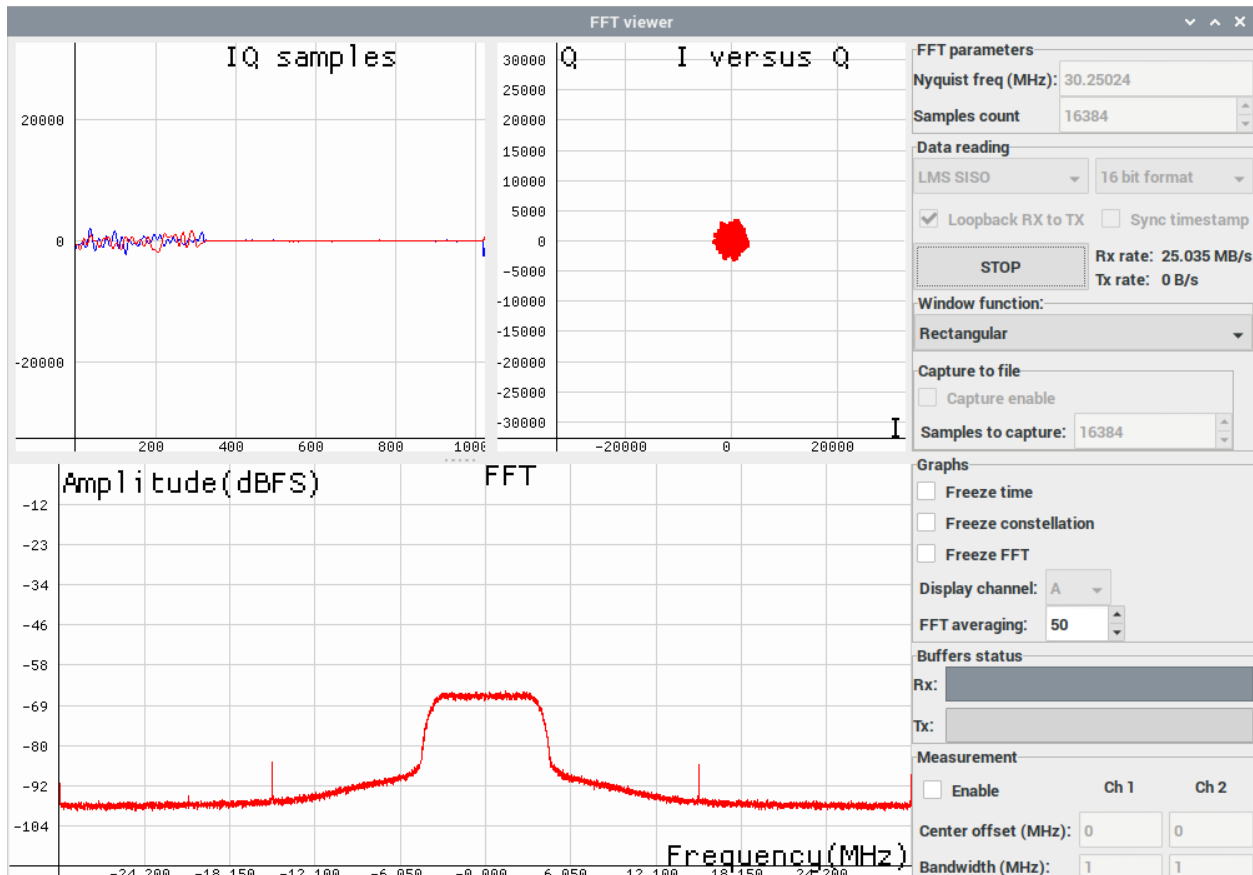


Figure 4.11: LimeSuite GUI FFT viewer currently running showing transient, constellation, and FFT representations of the received data.

```

IPython: Software/pyLMS7002Soapy
Python 3.7.3 (default, Dec 20 2019, 18:57:59)
Type "copyright", "credits" or "license()" for more information.

IPython 5.8.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

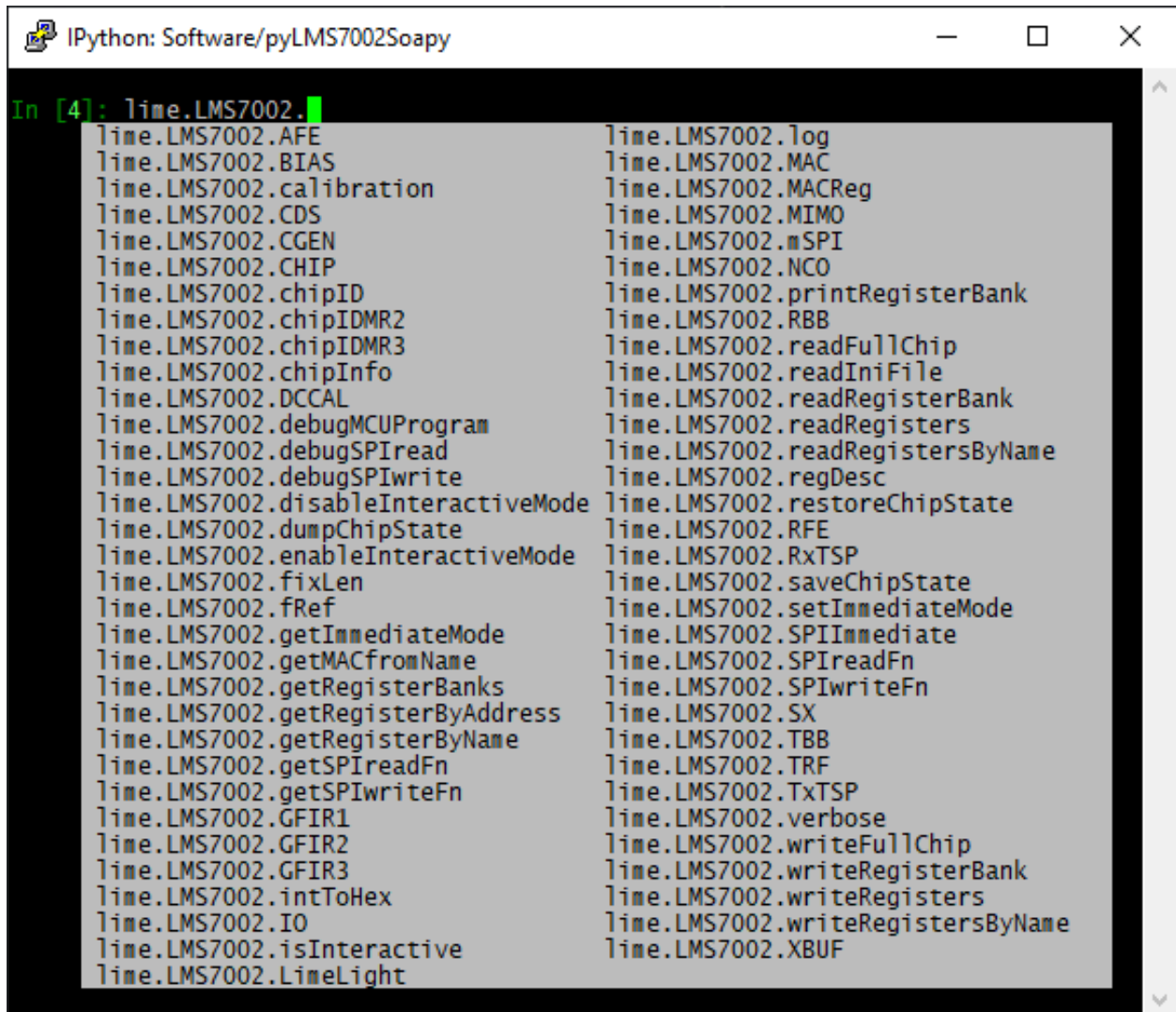
In [1]: from pyLMS7002Soapy import pyLMS7002Soapy as pylms

In [2]: lime = pylms.pyLMS7002Soapy()
[INFO] Make connection: 'LimeSDR Mini [USB 2.0] 1D538841C2E28D'
[INFO] Reference clock 40.00 MHz
[INFO] Device name: LimeSDR-Mini
[INFO] Reference: 40 MHz
[INFO] LMS7002M register cache: Disabled

In [3]: █

```

Figure 4.12: Example terminal for a successful SDR connection using the pyLMS7002Soapy API.



The image shows a screenshot of an IPython terminal window titled "IPython: Software/pyLMS7002Soapy". The prompt is "In [4]: lime.LMS7002." followed by a green cursor. The terminal displays a list of available attributes for the lime.LMS7002 module, arranged in two columns. The attributes include various methods and constants such as .AFE, .BIAS, .calibration, .CDS, .CGEN, .CHIP, .chipID, .chipIDMR2, .chipIDMR3, .chipInfo, .DCCAL, .debugMCUProgram, .debugSPIread, .debugSPIwrite, .disableInteractiveMode, .dumpChipState, .enableInteractiveMode, .fixLen, .fRef, .getImmediateMode, .getMACfromName, .getRegisterBanks, .getRegisterByAddress, .getRegisterByName, .getSPIreadFn, .getSPIwriteFn, .GFIR1, .GFIR2, .GFIR3, .intToHex, .IO, .isInteractive, .LimeLight, .log, .MAC, .MACReg, .MIMO, .mSPI, .NCO, .printRegisterBank, .RBB, .readFullChip, .readIniFile, .readRegisterBank, .readRegisters, .readRegistersByName, .regDesc, .restoreChipState, .RFE, .RxTSP, .saveChipState, .setImmediateMode, .SPIImmediate, .SPIreadFn, .SPIwriteFn, .SX, .TBB, .TRF, .TxTSP, .verbose, .writeFullChip, .writeRegisterBank, .writeRegisters, .writeRegistersByName, and .XBUF.

```
In [4]: lime.LMS7002.  
lime.LMS7002.AFE  
lime.LMS7002.BIAS  
lime.LMS7002.calibration  
lime.LMS7002.CDS  
lime.LMS7002.CGEN  
lime.LMS7002.CHIP  
lime.LMS7002.chipID  
lime.LMS7002.chipIDMR2  
lime.LMS7002.chipIDMR3  
lime.LMS7002.chipInfo  
lime.LMS7002.DCCAL  
lime.LMS7002.debugMCUProgram  
lime.LMS7002.debugSPIread  
lime.LMS7002.debugSPIwrite  
lime.LMS7002.disableInteractiveMode  
lime.LMS7002.dumpChipState  
lime.LMS7002.enableInteractiveMode  
lime.LMS7002.fixLen  
lime.LMS7002.fRef  
lime.LMS7002.getImmediateMode  
lime.LMS7002.getMACfromName  
lime.LMS7002.getRegisterBanks  
lime.LMS7002.getRegisterByAddress  
lime.LMS7002.getRegisterByName  
lime.LMS7002.getSPIreadFn  
lime.LMS7002.getSPIwriteFn  
lime.LMS7002.GFIR1  
lime.LMS7002.GFIR2  
lime.LMS7002.GFIR3  
lime.LMS7002.intToHex  
lime.LMS7002.IO  
lime.LMS7002.isInteractive  
lime.LMS7002.LimeLight  
lime.LMS7002.log  
lime.LMS7002.MAC  
lime.LMS7002.MACReg  
lime.LMS7002.MIMO  
lime.LMS7002.mSPI  
lime.LMS7002.NCO  
lime.LMS7002.printRegisterBank  
lime.LMS7002.RBB  
lime.LMS7002.readFullChip  
lime.LMS7002.readIniFile  
lime.LMS7002.readRegisterBank  
lime.LMS7002.readRegisters  
lime.LMS7002.readRegistersByName  
lime.LMS7002.regDesc  
lime.LMS7002.restoreChipState  
lime.LMS7002.RFE  
lime.LMS7002.RxTSP  
lime.LMS7002.saveChipState  
lime.LMS7002.setImmediateMode  
lime.LMS7002.SPIImmediate  
lime.LMS7002.SPIreadFn  
lime.LMS7002.SPIwriteFn  
lime.LMS7002.SX  
lime.LMS7002.TBB  
lime.LMS7002.TRF  
lime.LMS7002.TxTSP  
lime.LMS7002.verbose  
lime.LMS7002.writeFullChip  
lime.LMS7002.writeRegisterBank  
lime.LMS7002.writeRegisters  
lime.LMS7002.writeRegistersByName  
lime.LMS7002.XBUF
```

Figure 4.13: Example of traversing the API using the ipython3 tab completion feature to show all available functions and classes from the LMS7002 library.

CHAPTER 5. CONCLUSION

5.1 Work Summary

This work explores the feasibility of using a software defined radio (SDR) to create a small, low-cost, ranging radar suitable for use in a small spacecraft. In order to prove the feasibility of such a radar, a number of parameters need to be met. The selection of the LimeSDR-Mini, an open-source, single board SDR, fulfills the size and cost constraints as outlined in Appendix A. The LimeSDR-Mini also affords a number of key operating characteristics proven by this work that benefit radar operation. These include full-duplex mode, receive and transmit data streams, and calibration.

The SDR needs to have full-duplex capabilities as it allows the radar to transmit and receive simultaneously. Otherwise, the radar would have a much greater minimum detectable distance as the hardware would need to switch between transmit and receive modes. This switching time varies between different SDRs with half duplex operation. This capability is proven through GNU Radio using the flow graph shown in Fig. D.5.

The ability to stream data to and from the SDR is proven in each of the programming methods discussed in this work. GNU Radio showed this by streaming data into a LimeSuite transmit block and reading data from the corollary receive block, shown in Figs. D.4 and D.3. The LimeSuite GUI, though a bit contrived, demonstrated this through sending a waveform to the SDR using the FPGA controls and reading the data back with the FFT viewer, as shown in Chapter F. Lastly, the pyLMS7002Soapy API has this capability through the SoapySDR library, allowing the user to set up both a receive and transmit stream structure, also shown in Chapter F.

The pyLMS7002Soapy API lends itself to multiple calibration techniques to ensure signal quality and detection. The way that the API is written, using `@property` tags, enables the SDR to be automatically calibrated with ease. When a parameter is set, the corresponding setter function is called which trickles the parameter down through all the necessary classes as well as modifying

related settings. The API also provides an entire library dedicated to calibration to calculate and set necessary parameters throughout the SDR to achieve user-programmed parameters. Lastly, when needed, the user can tune the on-chip microcontroller to fine tune the SDR's operation even further.

The MatLab simulation outlined in Chapter 2 can be used to iteratively test different configuration options to see the effects on the radar's performance. These tests enable the programmer to set configurations within the API intelligently and give a expected performance baseline.

5.2 Conclusion

This work has determined that the LimeSDR-Mini can be used as a small, low-cost ranging radar front-end. Its small size (6.9 x 3.2 cm) makes it a good candidate to fit within a CubeSAT (10 x 10 x 10 cm), its cost of \$160 is cheap allowing budget to be used elsewhere, and its capabilities explored in this work prove it to be desirable SDR for radar applications.

GNU Radio Companion, given its drawbacks, is not a satisfactory programming or deploying software platform for this work. Due to its difficulty in use and implementation, the pyLMS7002Sopay API is preferred. As currently explored by this work, the API provides all of the necessary configurability and development/deployment options necessary for research and maintenance.

5.3 Future Work

Many aspects of this work were not fully explored and deserve further research, aside from working toward a first prototype.

- **LimeSuite GUI** - The LimeSuite GUI affords a plethora of options and configurations to the end user. As such, nearly all of the capabilities of the LimeSDR-Mini have not been explored. It is possible that some of these configurations will afford better radar implementation.
- **Calibration** - While calibration is explored in this work, the full extent of the calibration options and capabilities should be researched in more depth, primarily using the microcontroller. It is recommended to start with the MyriadRF documentation on LMS7002

MR3 calibration using the microcontroller (https://wiki.myriadrf.org/LMS7002Mr3_Calibration_Using_MCU).

- **API Integration** - Even though the LimeSuite GUI is organized to closely mirror the libraries found within the API, many functions in one are not readily found in the other. As such, when more detailed and developed LimeSuite GUI configurations emerge, equal effort needs to be put into discovering the corresponding capabilities within the API. Ultimately, the code that should be deployed on the TRICS is a Python script utilizing this API.

REFERENCES

- [1] P. Walton, J. Cannon, B. Damitz, T. Downs, D. Glick, J. Holtom, N. Kohls, A. Laraway, I. Matheson, and J. e. a. Redding, “Passive cubesats for remote inspection of space vehicles,” *Journal of Applied Remote Sensing*, vol. 13, no. 03, p. 1, 2019. 1
- [2] F. T. Ulaby and D. G. Long, *Microwave radar and radiometric remote sensing*. The University of Michigan press, 2014. 2
- [3] K. F. Warnick, R. Maaskant, M. V. Ivashina, D. B. Davidson, and B. D. Jeffs, *Phased arrays for radio astronomy, remote sensing, and satellite communications*. Cambridge University Press, 2018. 2, 3
- [4] F. T. Ulaby and A. E. Yagle, *Engineering signals and systems*. NTS, 2013. 2, 6
- [5] N. Levanon, *Radar principles*. John Wiley & Sons, 1988. 4
- [6] J. Rutman, “Relations between spectral purity and frequency stability,” in *28th Annual Symposium on Frequency Control*, 1974, pp. 160–165. 5
- [7] T. C. Carusone, D. Johns, and K. W. Martin, *Analog integrated circuit design*. John Wiley & Sons, 2014. 6
- [8] D. Comer, *Communication and power circuits*. Brigham Young University Press, 2017. 6
- [9] K. L. Su, *Analog filters*. Chapman & Hall, 1996. 6
- [10] N. Levanon and E. Mozeson, *Radar signals*. Wiley, 2004. 6
- [11] M. A. Richards, J. Scheer, and W. A. Holm, *Principles of modern radar*. SciTech Publishing, 2010. 8
- [12] L. Freitas, “Software defined doppler radar with limesdr mini,” 2020. [Online]. Available: <https://luigifreitas.me/2018-11-23/software-defined-radar-cw-doppler-radar-with-limesdr> 34

APPENDIX A. SDR COMPARISON AND SELECTION

This appendix outlines the hardware considerations taken toward the selection of the software defined radio to be used in this work. Table A.1 compares six different readily accepted transmit and receive capable SDRs (since many cheaper SDRs are receive capable only).

A.1 Considerations and Selection

Seven different comparison metrics are considered in the selection of an SDR for this work.

- **RF Range** - The RF range is the spread of different mixer frequencies the radio can implement. The mixer frequency is also synonymous with carrier frequency. A wider RF range means a larger variety of bands the radio can use.
- **Bandwidth** - As discussed in Chapter 2, the bandwidth of the radio is related to the range resolution. With a target range resolution finer than twenty meters, according to Equation 2.7, the SDR needs a bandwidth of at least 15 MHz.
- **Sample Depth** - The sample depth is the resolution of each sample coming into or out of the radio. With higher sample depth, the radio is capable of producing and reading signals more accurately.
- **Full Duplex** - The duplex nature of a transceiver indicates whether it can transmit and receive simultaneously (full duplex) or not (half duplex). For short range ranging radars, such as the one considered in this work, is required as the hardware switching time places additional constraints on minimum detectable range.
- **Open-Source** - With open-source hardware and software, it is possible to access and analyze schematics and code-bases. This can dramatically improve development time and future maintenance, even if the company no longer supports the project.

- **Physical Size** - The SDR must fit within the 10 x 10 x 10 cm size constraints placed by NASA for the miniature satellite to qualify as a CubeSAT.
- **Cost** - When a CubeSAT has completed its inspection of the launching satellite, rather than being retrieved, it falls and burns up in the Earth's atmosphere. As such, it is important for it to be cheap to minimize sunken cost. The SDR should be low cost to help keep costs down.

Considering the requirements listed above and observing the comparisons of the different SDRs in Table A.1, the LimeSDR-Mini is selected for use in this work as it is the only SDR considered that did not exceed constraints or fail to meet requirements.

A.2 SDR Comparisons

- **HackRF One** - <https://greatscottgadgets.com/hackrf/one/>
- **YARD Stick One** - <https://greatscottgadgets.com/yardstickone/>
- **LimeSDR-USB** - <https://myriadrft.org/projects/component/limesdr/>
- **LimeSDR-Mini** - <https://myriadrft.org/projects/component/limesdr-mini/>
- **Ettus B210** - <https://www.ettus.com/all-products/ub210-kit/>
- **BladeRF x40** - <https://www.nuand.com/product/bladerf-x40/>

Table A.1: A comparison of six different software defined radios. Green squares indicate best or good, uncolored squares are acceptable, and red squares indicate bad or outside project restrictions.

	HackRF One	YARD Stick One	LimeSDR-USB	LimeSDR-mini	Ettus B210	BladeRF x40
RF Range	1 MHz - 6 GHz	281 - 962 MHz	100 kHz - 3.8 GHz	10 MHz - 3.5 GHz	70 MHz - 6 GHz	300 MHz - 3.8 GHz
Bandwidth	20 MHz	500 kHz	61.44 MHz	30.72 MHz	61.44 MHz	40 MHz
Sample Depth	8 bits	8 bits	12 bits	12 bit	12 bits	12 bits
Full Duplex	No	No	Yes	Yes	Yes	Yes
Open-Source	Hardware	Hardware	Hardware & Software	Hardware & Software	Hardware	Hardware
Physical Size	124 x 80 x 18 mm	84 x 19 x 6 mm	100 x 60 x 6 mm	69 x 31.4 x 6 mm	Inspection: Large	131 x 87 x 18 mm
Price	\$339.95	\$99.95	\$299.00	\$159.00	\$1282.00	\$420

APPENDIX B. LFM CW SIMULATION MATLAB SCRIPT

This appendix contains the full LFM CW simulation MatLab script described in Chapter 2. In addition to the simulation script, a helper function utilized by the main script is also contained.

B.1 Main Code

Since the simulation is not operating in real-time, everything is calculated upfront. To simulate this, multiple periods of the LFM CW is calculated as the transmit signal. Then the transmit signal is copied and noise is added to simulate channel white noise. Then to simulate sampling, a random index within the first period of the transmit signal is selected and a set number of samples are taken from that index. The receive signal is sampled similarly, but the starting index is increased by a number of bins correlating to the simulation target distance. The rest of the simulation follows the LFM CW radar theory described in Chapter 2. There are a number of parameters that can be changed by the user to explore the effect on the radar operation. These include:

- `f_samp` - sample rate of the theoretical ADC and DAC
- `T` - chirp duration
- `f0` - LFM CW starting frequency
- `fbw` - LFM CW bandwidth (ending frequency minus starting frequency)
- `f1o` - local oscillator/mixing frequency/carrier frequency
- `Gs` - signal amplitude
- `Gn` - noise amplitude
- `channel_noise` - noise selection (0: none, 1: gaussian, 2: uniform)

- `sim_distance` - distance of simulated target

```

1 clear; clc; close all;
2
3 %% Paramters
4 % Constants and Simulation Params
5 c = 2.998e8; % speed of light
6 fmax = 1e9; % maximum frequency for simulation
7 fsamp = 1e6; % sample rate of receiver ADC
8 T = 100e-6; % time window
9 n_parts = 3; % number of simulated lfmw signal parts
10
11 % Signal Generation Params
12 f0 = 1e6; % base frequency for signal
13 fbw = 30.62e6; % bandwidth of linear chirp
14 flo = 245e6; % local oscillator for (de)modulation
15 Gs = 1; % signal amplitude
16
17 % Channel Parameters
18 Gn = 50; % channel noise amplitude
19 channel_noise = 0; % 0: no noise, 1: gaussian noise, 2: uniform
    noise
20 sim_distance = 184.7; % signal round trip distance in meters
21
22 %% Radar Characteristics
23 delta_r = c/fbw; % calculate range resolution
24 abs_max_range = (c*T)/2; % calculate theoretical absolute
    maximum range
25 actual_max_range = (c*fsamp*T)/(2*fbw); % calculate actual
    maximum range

```

```

26 fprintf('Simulation range is %.1f meters.\n\n', sim_distance);
    % display simulation range
27 fprintf('Range resolution is %.3f meters.\n', delta_r); %
    display info
28 fprintf('Theoretical maximum range is %.1f meters.\n',
    abs_max_range); % display info
29 fprintf('Actual maximum range is %.1f meters.\n',
    actual_max_range); % display info
30
31 if sim_distance > actual_max_range % check if input valid
    simulation range
32     % if not valid, print warning
33     fprintf('Simulated range (%.1fm) larger than maximum range
    (%.1fm).\nExiting...\n', sim_distance, actual_max_range);
34     return; % exit program
35 end
36
37 %% Signal Generation
38 t_full = 0:1/(2*fmax):T*n_parts-1/(2*fmax); % full signal time
    array
39 t_part = t_full(1:length(t_full)/n_parts); % time array for one
    signal part
40 part_len = length(t_part); % array length of each part
41
42 alpha = fbw*pi/T; % chirp rate for lfmw in rad/s
43 beta = 2*pi*f0; % base frequency for lfmw in rad/s
44 phi = 0; % initial phase for lfmw in radians
45
46 % baseband linear frequency modulated signal part
47 bb_lfmw = exp(1j*(alpha.*t_part.^2 + beta.*t_part + phi));

```

```

48
49 bb_full = zeros(1,n_parts*part_len); % empty array for full
    signal
50 % populate full signal by repeating bb_lfmcw n_parts times
51 for i = 0:1:n_parts-1
52     % populate with linear modulated chirp
53     bb_full(i*part_len+1:(i+1)*part_len) = bb_lfmcw;
54 end
55
56 %% IQ Modulate
57 tx_inph = real(bb_full).*cos(2*pi*flo.*t_full); % upmix inphase
    portion (real part)
58 tx_quad = imag(bb_full).*sin(2*pi*flo.*t_full); % upmix
    quadrature portion (imag part)
59 tx = tx_inph + tx_quad; % sum inphase and quadrature portions
    for transmit
60
61 %% Send Signal Through Channel
62 if channel_noise == 0 % check if noiseless channel
63     rx = tx; % send data through with no noise
64 elseif channel_noise == 1 % check adding gaussian noise
65     rx = tx + Gn*randn(size(tx)); % channel with gaussian noise
66 elseif channel_noise == 2 % check adding uniform noise
67     rx = tx + Gn*(2*rand(size(tx))-1); % channel with uniform
    noise
68 end
69
70 %% IQ Demodulate
71 % IQ Demodulate Received Signal
72 rx_inph = rx.*cos(2*pi*flo.*t_full); % downmix inphase portion

```

```

73 rx_quad = rx.*sin(2*pi*flo.*t_full); % downmix quadrature
    portion
74
75 % Lowpass Filter Received Data Based on BW
76 bb_inph = lowpass(rx_inph, 2*fbw, fmax/T); % lowpass inphase
    data
77 bb_quad = lowpass(rx_quad, 2*fbw, fmax/T); % lowpass quadrature
    data
78
79 bb_reconst = 2.*(bb_inph + 1j.*bb_quad); % restore original
    signal
80
81 %% Simulate Sampling Portion of Signal
82 index = ceil(rand()*(length(t_part)-1)); % transmit signal
    starts from random index
83 sample_window = length(t_part); % number of samples for "
    sampled" signal (rx and tx)
84 dist_per_sample = c/fmax; % calculate distance per sample
85
86 ch_delay = round(2*sim_distance/dist_per_sample); % calculate
    channel delay in samples
87
88 tx_sampled = bb_full(index+1:index+sample_window); % "sample"
    tx signal
89 rx_sampled = bb_reconst(index+ch_delay+1:index+sample_window+
    ch_delay); % "sample" rx signal
90 % rx_sampled = 0.2*bb_reconst(index+ch_delay+1:index+
    sample_window+ch_delay) + 0.8*tx_sampled; % "sample" rx
    signal with crosstalk
91

```

```

92 %% Range Detection
93 range_data = tx_sampled.*conj(rx_sampled); % multiply two
    signals together
94 figure();
95 plot(t_part, real(range_data), 'LineWidth', 2);
96 xlabel('Time');
97 ylabel('Signal Amplitude');
98 grid on;
99 if channel_noise == 0
100     ylim([-1.1, 1.1]);
101 end
102
103 [f, fft] = plottableFFT(range_data, T, 0); % calculate fft
104
105 fft_power = abs(fft)/max(abs(fft)); % convert fft to power
    spectral density
106 fft_power = 10*log10(fft_power.^2);
107 figure();
108 plot(f, fft_power, 'LineWidth', 2); % plot the PSD
109 xlim([-fsamp/2, fsamp/2]); % change freq limit to match ADC
    capabilities
110 grid on; % turn on grid
111 xlabel('Frequency (Hz)');
112 ylabel('Normalized Signal Power (dB)');
113 % title('Power Spectral Density of Received Signal');
114
115 max_power = max(fft_power); % find peak power
116 tgt_bin = find(fft_power == max_power); % find index of peak
117 tgt_freq = f(tgt_bin); % find frequency of peak
118

```

```

119 range = -tgt_freq*T*delta_r; % convert given frequency to
    distance
120 fprintf('\nDetected range is %.1f meters.\n', range); % print
    out detected range

```

B.2 Helper Function

This helper function calculates the frequency shifted and unshifted FFTs with corresponding frequency spectra for the given signal over a given time period. The default time period is one second if time period argument is given. Additionally, multiple figures are plotted for convenience if the third argument to the function is 1. Figures are print by default if no third argument is given. To use, copy the code into another MatLab script named “plottableFFT.m” within the same working directory as the script that is using it. Then call it similarly to other functions.

```

1 %% Custom Function to Calculate FFT and Corresponding Frequency
    Array
2 % Returns frequency shifted and unshifted fft with
    corresponding frequency spectrum
3 function [fshift, yshift, f, y] = plottableFFT(signal, T, fig)
4     %% Check Arguments and Set Default Values
5     if nargin < 3 % if number of arguments is less than 3
6         fig = 1; % set fig (print figure) to true
7     end
8     if nargin < 2 % if number of arguments is less than 2
9         T = 1; % assume given signal is over 1 second
10    end
11
12    %% Calculations
13    % calculate the max frequency (2*nyquist)
14    fmax = length(signal)/T;
15    % calculate non-shifted fft (has aliased negative signal)

```



```

16     y = fft(signal);
17     % generate corresponding frequency spectrum (0 to 2*nyquist
18     )
19     f = (0:length(y)-1)*(fmax/length(y));
20     % shift previously calculated fft
21     yshift = fftshift(y);
22     % generate shifted frequency spectrum (-nyquist to nyquist)
23     fshift = (-length(signal)/2:length(signal)/2-1)*(fmax/
24     length(signal));
25
26     %% Display Figure
27     % check if user wants figure plotted
28     if fig == 1
29         figure(); % open new figure
30         plot(fshift,real(yshift)); % plot shifted fft
31         title('FFT (real)'); % title figure
32         xlabel('Frequency (f)'); % name x axis
33         grid on; % turn on grid
34         figure();
35         plot(fshift,imag(yshift)); % plot shifted fft
36         title('FFT (imag)'); % title figure
37         xlabel('Frequency (f)'); % name x axis
38         grid on;
39         figure();
40         plot(fshift,abs(yshift)); % plot shifted fft
41         title('FFT (magnitude)'); % title figure
42         xlabel('Frequency (f)'); % name x axis
43         grid on;
44     end
45 end

```

APPENDIX C. GNU RADIO AND GR-RADAR INSTALLATION

GNU Radio is an open-source, community driven SDR programming platform. This appendix provides installation instructions for GNU Radio and the requisite toolboxes used in this work, namely gr-radar and LimeSuite, on a desktop. No other hardware is required. Due to the shortcomings of GNU Radio, with respect to this work, it is not recommended for use. Instead, follow the instructions in Appendix E to install the pyLMS7002Soapy API.

C.1 Preface

The installation directions online say that the GNU Radio Companion software is capable of running on Windows; however, this work uses Ubuntu 16.04 LTS as windows does not natively support the gr-radar toolbox. Additionally, this distro is being used (despite being four years old and not supported) because the gr-radar toolbox for GNU Radio is not updated to run on newer distros and breaks. The instructions below assume the user has just completed a fresh install of Ubuntu 16.04 LTS.

This can also be successfully installed on an Ubuntu VirtualBox (virtual machine software from Oracle). In order to configure the virtual machine to recognize a USB device, namely the LimeSDR, there are a few extra steps that need to be done. These instructions will follow the basic installation section.

C.2 Installation

C.2.1 Basic Installation

1. Open terminal in Ubuntu
2. Type the following commands:

```
$ sudo apt-get install git
$ cd Downloads
$ wget https://bootstrap.pypa.io/get-pip.py
$ sudo python3 get-pip.py
$ pip --version
```

3. The response from this should display that pip 19.1 is currently installed.
4. Type the following commands to complete the installation:

```
$ cd ~
$ sudo apt-get install python-mako
$ sudo pip install pybombs
$ sudo pip install --upgrade git+https://github.com/gnuradio/pybombs.git
$ pybombs auto-config
$ pybombs recipes add-defaults
$ pybombs recipes add gr-recipes https://github.com/gnuradio/gr-recipes.git
$ sudo pip install setuptools
$ pybombs install setuptools
$ pybombs prefix init ~/prefix -a myprefix -R gnuradio-default
$ pybombs install gr-radar
$ pybombs install gr-limesdr
```

If at any point during the installation pybombs exits with an error, it's due to missing dependencies. Manually find and install the necessary repositories and run the pybombs install command again to reattempt. Pybombs is intelligent enough to not reinstall everything that already succeeded, so subsequent attempts are faster.

If followed correctly, GNU radio, gr-radar, and the LimeSDR components have been successfully installed. To run the GNU Radio Companion software, type `pybombs run gnuradio-companion` in a terminal window.

C.2.2 Additional Steps for VirtualBox

To run GNU Radio and program a LimeSDR in a VirtualBox, a few extra things need to be completed to setup the virtual machine sufficiently. These include:

- Installing guest permissions - this installs a number of quality of life drivers, but most importantly allows for dynamic screen resolutions including full screen.
- Enabling USB support - this is necessary as it allows the virtual machine to see specific USB devices which are hidden from the VM by default. Without this, it would be impossible to see the LimeSDR from within the VM.

The following instructions assume VirtualBox basics and Ubuntu 16.04 LTS has already installed to a virtual disk.

Installing Guest Additions

1. Boot your Ubuntu VM
2. At the top of the window, select the “Devices” drop down menu and then select “Insert Guest Additions CD Image...”
3. Follow the prompts to install guest additions
4. Right click the CD on the desktop and eject it
5. Restart the VM

As stated earlier, the guest additions also allow you to full screen the VM to make it easier to use. To switch back and forth between full screen and windowed modes, press RGT_CTRL+F.

Enabling USB Support

1. On the host computer, not in the VM, go to <https://www.virtualbox.org/wiki/Downloads> and download the Oracle VM VirtualBox Extension Pack
2. Navigate to where the downloaded file was saved and double click the extensions pack

3. Follow the prompts to install the extensions
4. Shutdown the VM if it is currently running
5. Select your Ubuntu VM in the left panel of the VirtualBox dialog (not in the VM) and click “Settings”
6. Select “USB” in the left panel
7. Check the box to enable the virtual USB controller
8. Select the radio button for the USB 3.0 controller
9. Plug in the LimeSDR and wait for your main operating system to detect and register the device
10. Click on the small USB icon with a green plus on the right edge of the dialog box
11. Within the devices menu that appears, select the connected LimeSDR
12. The radio should show up as an option under the USB Device Filter section
13. Double click on the device
14. Empty all the fields except for “Name” and “Serial No.”
15. At the bottom of the same dialog, change “Remote” to “Any”

With the LimeSDR now configured to the VM, the radio will be detected by the VM when plugged in. Note that if the radio is plugged in before the VM is turned on, it will not be detected once the machine is running. To successfully have the VM detect the radio, it should be plugged in after the machine has been booted.

C.2.3 Installation Verification

At this point, the VM is configured correctly and the requisite software is installed. To ensure that the VM can see the LimeSDR:

1. Plug in your radio to a USB port
2. Open terminal
3. Type `LimeUtil --find`

The utility should find the LimeSDR and return information such as

```
* [LimeSDR Mini, media=USB 3.0, module=FT601, addr=24607:1027,
serial=1D497907CE3158].
```

Take note the serial number at the end of the return string. It can be used in GNU Radio to identify your radio from others if there are multiple radios plugged into the machine. If the “ID” field in LimeSuite transmit or receive block is left blank, the radio with the alphanumerically first serial number will be chosen by default.

To verify that the software and toolboxes have been installed correctly:

1. Open GNU Radio Companion (by typing `pybombs run gnuradio-companion`)
2. Look at the right-most column. In this column, there should be a number of different categories such as “Audio,” “GUI Widgets,” and “Peak Detectors.”
3. Either collapse the “Core” library, in which all the aforementioned items are located, or scroll to the very bottom.

If the gr-radar and LimeSuite toolboxes and components were installed correctly, two new toolboxes called “LimeSuite” and “RADAR” should be present. Expanding these displays the new blocks afforded by these toolboxes.

C.3 Resources

- Ubuntu 16.04 LTS: <http://releases.ubuntu.com/16.04/>
- VirtualBox: <https://www.virtualbox.org/>
- PyBOMBS github: <https://github.com/gnuradio/pybombs>
- GNU Radio: <https://www.gnuradio.org/>
- gr-radar: <https://grradar.wordpress.com/>

- gr-radar github: <https://github.com/kit-cel/gr-radar>
- MyriadRF (makers of LimeSDR): <https://myriadrf.org/>
- LimeSDR-USB: <https://myriadrf.org/projects/component/limesdr/>
- LimeSDR USB github: <https://github.com/myriadrf/limesdr-usb>
- LimeSDR-Mini: <https://myriadrf.org/projects/component/limesdr-mini/>
- LimeSDR-Mini github: <https://github.com/myriadrf/limesdr-mini>

APPENDIX D. GNU RADIO EXPLORATION THROUGH FLOW GRAPHS

A few of the more complete and insightful flow graphs used to explore GNU Radio Companion are given here. Fig. D.1 is used to understand the flow graph structure as well as test a few components from the gr-radar toolbox. Fig. D.2 explores the parameters and functional operation of the gr-radar FMCW signal generator block, preparatory to its use in an FMCW radar flow graph. Figs. D.3 and D.4 are used to understand the LimeSuite toolbox’s transmit and receive blocks, again in preparation for use in an FMCW radar flow graph. Fig. D.5 demonstrates the full-duplex nature of the LimeSDR-Mini by using both the transmit and receive blocks from the LimeSuite toolbox simultaneously. Fig. D.6 demonstrates a full FSK radar by utilizing blocks from the core, gr-radar, and LimeSuite toolboxes.

D.1 Flow Graphs

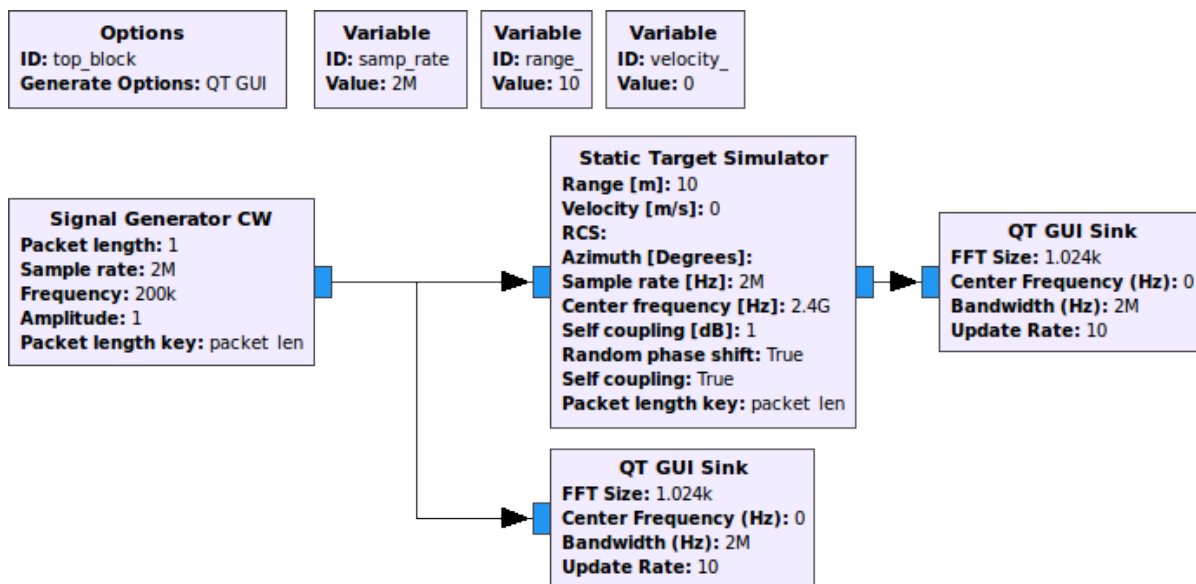


Figure D.1: Small flow graph to explore the gr-radar toolbox’s CW signal generator and target/channel simulator.

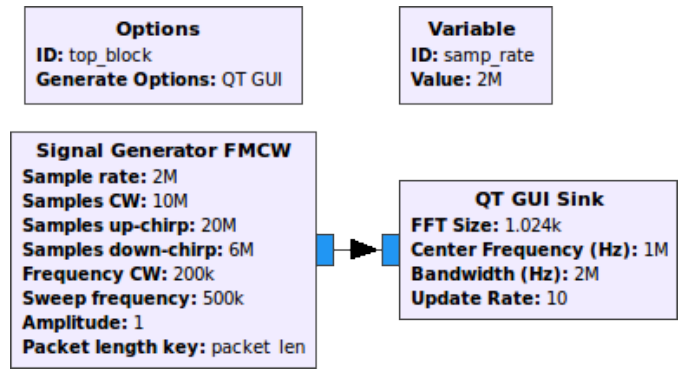


Figure D.2: Small flow graph to explore the gr-radar toolbox's FMCW signal generator.

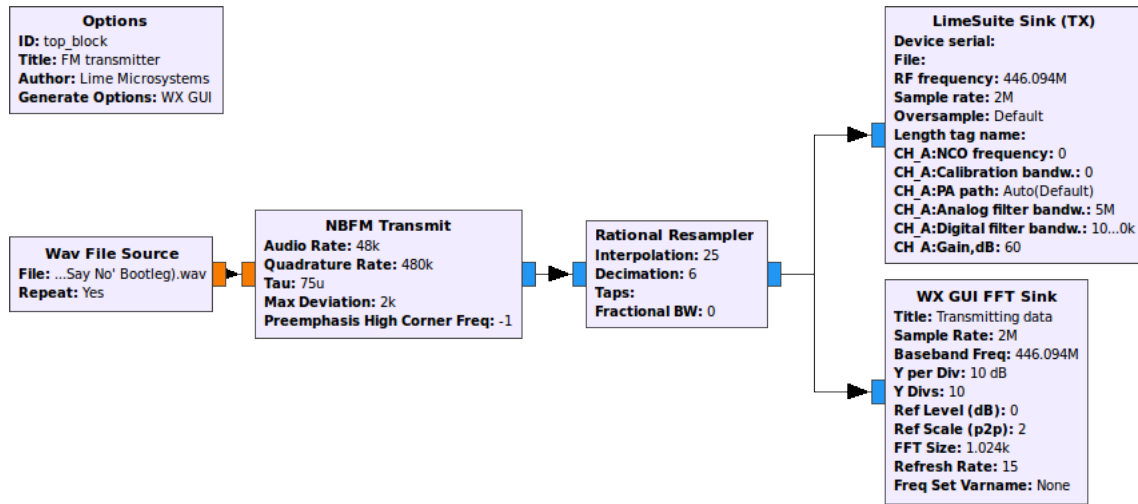


Figure D.3: Simple transmit flow graph using FM modulation to explore using the LimeSuite transmit block as well as a few other blocks.

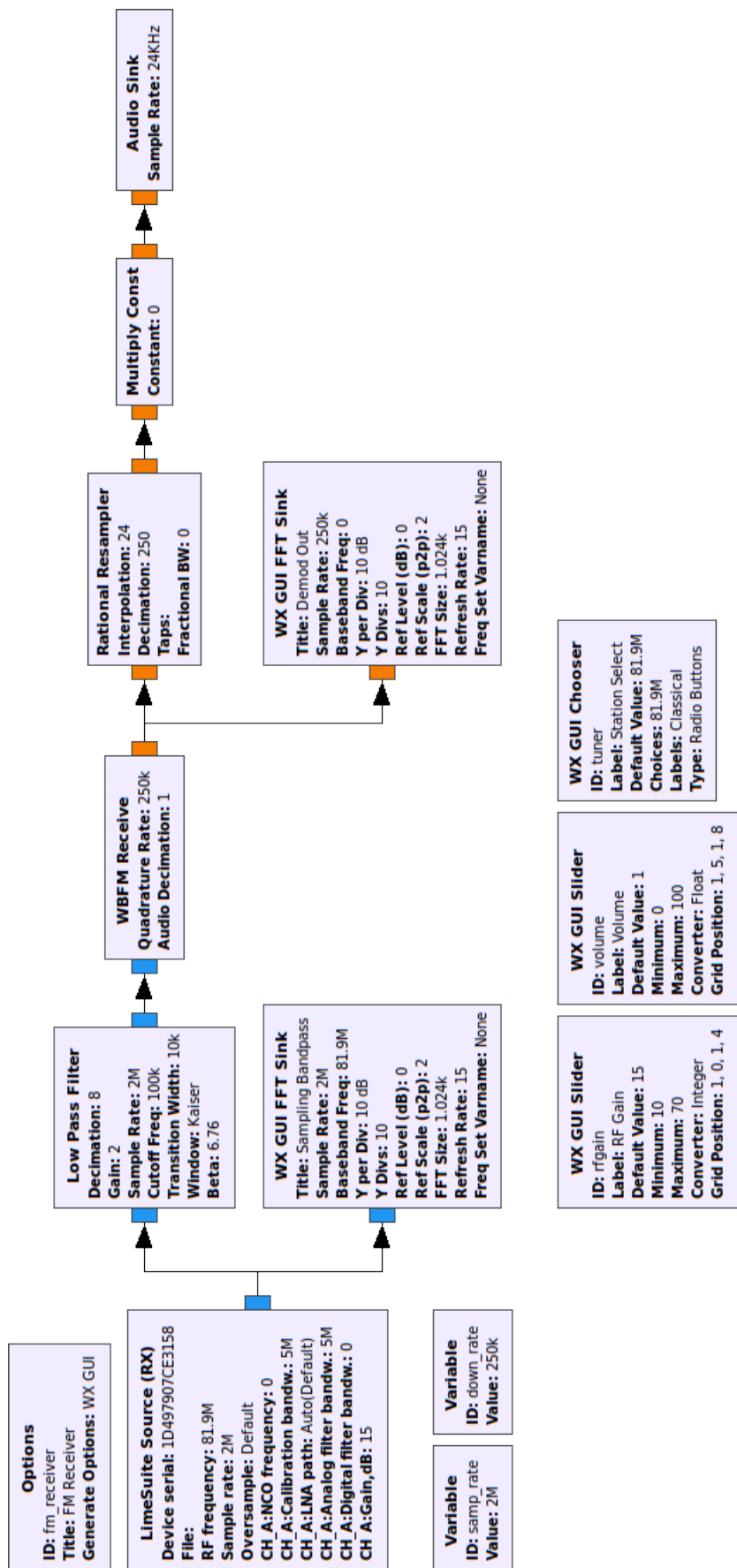


Figure D.4: Simple receive flow graph using FM modulation to explore using the LimeSuite receive block as well as a number of other blocks. Note that this flow graph has quite a few extra blocks for GUI sink control.

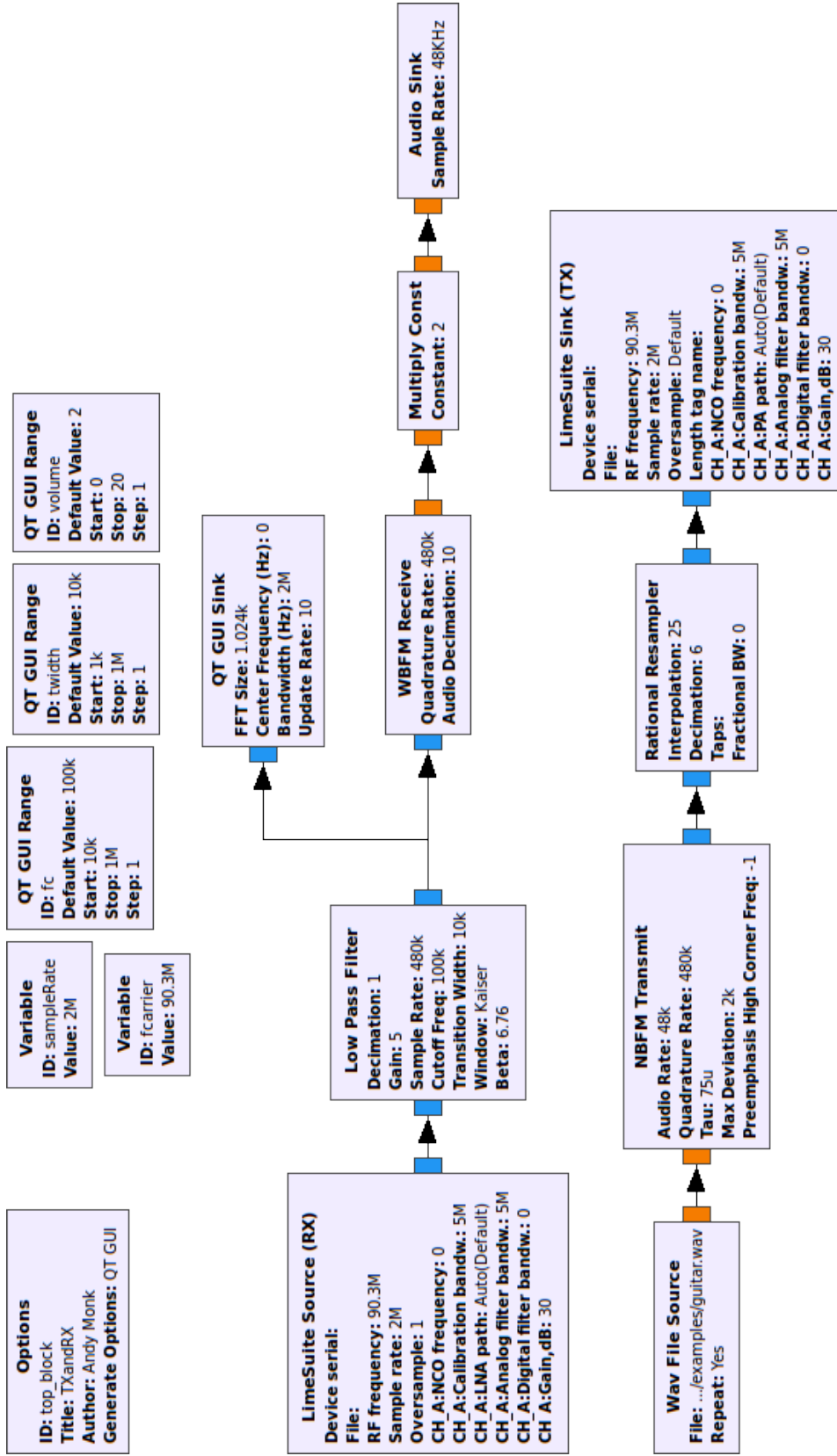


Figure D.5: This flow graph attempted to combine the transmit and receive flow graphs (Figs. D.3 and D.4). Operation did not throw any errors, however the data that did come through had issues. The most likely issue is sample rate mismatching.

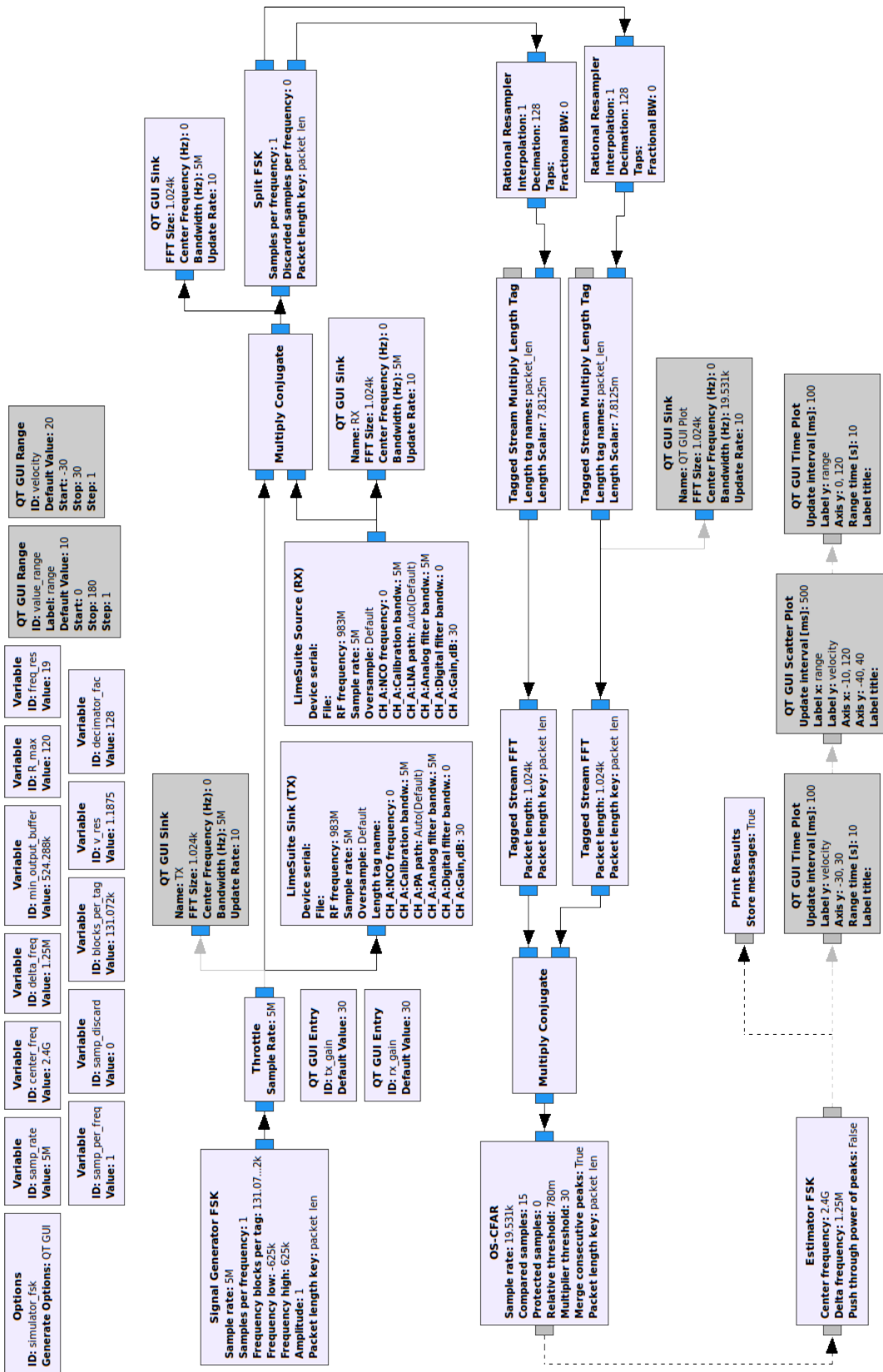


Figure D.6: Full FSK radar simulation. The original flow graph is provided in the gr-radar examples folder. The flow graph was modified to use the LimeSuite's transmit and receive blocks instead of a channel simulator block (located in the gr-radar toolbox).

APPENDIX E. PISDR INSTALLATION

PiSDR is a custom Raspbian image for the Raspberry Pi developed by Luigi Freitas. It is designed to support a range of common SDRs with a variety of useful SDR software applications out of the box. This is helpful to this work as it supports the LimeSDR-Mini and -USB as well as having all of the necessary software, with the exception of the pyLMS7002Soapy API, installed by default. Additionally, it runs on the Raspberry Pi rather than a computer which is more akin to the solution that will likely be implemented on the TRICS.

E.1 Required Hardware

- Raspberry Pi 3B+ or 4 (I used a 4, 1GB variant)
- Appropriate Raspberry Pi power supply
- Micro SD card between 8 and 32GB
- Another computer, preferably Windows, to setup microSD card
- Appropriate microSD card adapter to desktop
- Monitor with appropriate video cable
- USB keyboard and mouse

E.2 PiSDR Installation

If you know how to set up a Raspberry Pi with a custom image and enable SSH, skip ahead to the next section (pyLMS7002Soapy API Installation).

On a computer other than the Raspberry Pi:

1. Insert micro SD card

2. Download, install, and run the SD card formatter utility
3. Format the inserted SD card
4. Download, install, and run the Raspberry Pi Imager tool
5. Download the PiSDR image
6. Click “Choose OS”
7. From the Operating System dialog, select “Raspbian (other)”
8. Browse to and select the downloaded PiSDR image
9. Click “Choose SD Card” and select the newly formatted SD card from the dialog window
10. Click “Write”

At this point, the micro SD card is ready. Insert the micro SD card into the Raspberry Pi and hook up the monitor and USB peripherals. Connect the display before booting the Raspberry Pi to ensure that the display is recognized and enabled. Once the Raspberry Pi has boot:

1. Open a terminal and type `sudo raspi-config`
2. Change the user password - this is for the user “pi”
3. Under “Network Options,” change the hostname to something easily recognizable on the network
4. Under “Network Options,” set the WiFi credentials - if the WiFi credentials are set but upon opening a browser no pages load, this is likely due to incorrect local time inhibiting the browser from authenticating. To resolve this, type `sudo date --set '<year>--<month>--<day> <hour>:<min>:<sec>'` replacing each of the < >s with the appropriate number. Being perfect to the millisecond is not required as WiFi authentication allows some play in time. Also, once the Raspberry Pi has access to the internet with proper authentication, it can update the time accordingly.
5. Under “Interfacing Options,” enable SSH

Now the Raspberry Pi is set up to work in headless mode. You can now reboot and remove all of the peripherals. To access the Raspberry Pi, from another computer's terminal type `ssh pi@<ip-address>`. You will then be prompted for the user password.

E.3 pyLMS7002Soapy API Installation

1. Navigate to the desired location for the API. I chose `~/PiSDR/Software/`.
2. Type `git clone https://github.com/myriadrf/pyLMS7002Soapy`. This will clone the API repository to the current directory.
3. Navigate into the `pyLMS7002Soapy` directory created by the git.
4. Type `python3 setup.py install`.

E.4 Verification

1. Plug in the LimeSDR-Mini
2. Open a terminal and type the following commands:

```
$ LimeUtil --find
$ SoapySDRUtil --find
$ python3
>>> from pyLMS7002Soapy import *
>>> exit()
```

The outputs from these commands should look similarly to Fig. E.1. If either of the find commands do not return the LimeSDR-Mini's credentials, then something installed after the fresh PiSDR install ruined either the LimeSuite or SoapySDR architectures. If the Python import throws an error, then the API was installed incorrectly or was not completed.

Due to the ease and speed of installation of PiSDR, if you have either of these issues, a full reinstall is recommended while installing the basics described in this appendix first. Additional installations are recommended after verifying that all of the utilities and API work.

```
pi@lmcw-radar: ~/PiSDR/Software/pyLMS7002Soapy
File Edit Tabs Help
pi@lmcw-radar:~/PiSDR/Software/pyLMS7002Soapy $ LimeUtil --find
* [LimeSDR Mini, media=USB 2.0, module=FT601, addr=24607:1027, serial=1D538841C2E28D]

pi@lmcw-radar:~/PiSDR/Software/pyLMS7002Soapy $ SoapySDRUtil --find
#####
##      Soapy SDR -- the SDR abstraction library      ##
#####

Found device 0
  addr = 24607:1027
  driver = lime
  label = LimeSDR Mini [USB 2.0] 1D538841C2E28D
  media = USB 2.0
  module = FT601
  name = LimeSDR Mini
  serial = 1D538841C2E28D

pi@lmcw-radar:~/PiSDR/Software/pyLMS7002Soapy $ python3
Python 3.7.3 (default, Dec 20 2019, 18:57:59)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pyLMS7002Soapy import *
>>> exit()
pi@lmcw-radar:~/PiSDR/Software/pyLMS7002Soapy $
```

Figure E.1: Example of successful verification terminal outputs.

E.5 Recommendations

E.5.1 ipython3

ipython3 is a Python terminal similar to typing python3 into a terminal. The difference is that ipython3 supports color coding, tab completion, and tab suggestions. Navigating the API is significantly easier with these features and makes writing and debugging code that uses the API significantly faster.

To install, type `sudo apt install ipython3`. To run, type `ipython3`.

E.5.2 PuTTY

If you are working on a Windows computer, I recommend using PuTTY. To access an SSH machine through PuTTY, select the SSH radio button and input the target device's IP address into

the “Host Name (or IP address)” textbox and click “Open.” You will then be prompted to input the user (pi) and password.

E.5.3 GUI Applications through SSH

X11 Forwarding

Since this work relies on the LimeSuite GUI for API exploration, another way to access the software is needed if headless operation is desired. This can be done through SSH with the help of an x server. On windows machines, Xming is a good solution. To use it, launch the server (using the Xlaunch program) and then enabling x11 forwarding in PuTTY. If PuTTY is not being used, then an SSH connection can be made with x11 forwarding from the terminal by adding the “-X” parameter to the SSH command. For example, `ssh -X pi@<ip_address>` connects to the given IP address as the user pi with x11 forwarding.

VNC Server/Viewer

My recommendation is to use VNC as Raspberry Pis come with a free dedicated VNC server. This allows the user to pull up the Raspberry Pi desktop on any computer and work remotely. To do this, enable VNC through the interfacing options within `raspi-config`, as explained earlier.

VNC works by piping the current display(s) over a network. Due to how the Raspberry Pi handles displays, if no display is plugged in, then no display data is rendered. As such, no display data is available to send over the network. HDMI spoofers exist which are HDMI dongles that plug into the port and trick the machine into thinking that there is a display connected. These, however, are not available in micro-HDMI format, which the Raspberry Pi 4 uses. Instead, it is possible to modify the Raspberry Pi’s boot configuration to force HDMI output even when there is no detected display.

To do so, open and modify the `~/boot/config.txt` file and enable the HDMI hotplug by uncommenting the line `hdmi_force_hotplug = 1`. Now, the Raspberry Pi will run the VNC

server and correctly forward the display even when no monitor is connected. To access the Raspberry Pi from a remote computer, download and install VNC viewer and create a new connection.

E.5.4 Raspberry Pi IP Mailer

Since IP addresses are allocated by the router and are subject to change, without setting a static IP within the router's settings, it can often be frustrating to find the IP address for the Raspberry Pi running headless. To remedy this, I found and expanded a script to automatically email the Pi's IP address upon boot to a given email address.

E.6 Resources

- Raspberry Pi Imager - <https://www.raspberrypi.org/downloads/>
- SD Card Formatter - <https://www.sdcard.org/downloads/formatter/>
- PiSDR - <https://pisdr.luigifreitas.me/>
- pyLMS7002Soapy github - <https://github.com/myriadrf/pyLMS7002Soapy>
- Xming - <https://sourceforge.net/projects/xming/>
- VNC Viewer - <https://www.realvnc.com/en/connect/download/viewer/>
- Raspberry Pi IP Mailer - https://github.com/joebobs0n/rpi_ip_mailer

APPENDIX F. LIMESUITE GUI AND API

Most of the acronyms used in the LimeSuite GUI and pyLMS7002Soapy API are found in the documentation on the MyriadRF website (<https://myriadr.org/projects/component/limesdr-mini/>), but require hunting through multiple datasheets and resources as not all acronyms are defined in each document. Since MyriadRF assumes that the user of their devices automatically know their acronyms, despite often being different from the commonly accepted ones, herein are included the unabbreviated forms of many acronyms used in the LimeSuite GUI and pyLMS7002Soapy API.

F.1 LimeSuite GUI Tabs

- **Calibrations** - gain and phase calibration options for the receiver and transmitter
- **RFE** - receiver front end, particularly biasing and antenna data path selection
- **RBB** - receiver base band options
- **TRF** - transmit radio frequency options
- **TBB** - transmit base band options
- **AFE** - analog front end options (ADC and DAC controls)
- **BIAS** - bias options not covered elsewhere
- **LDO** - low drop out voltage regulator options
- **XBUF** - various buffer and reference options
- **CLKGEN** - clock generator parameters and options
- **SXR** - receiver synthesizer configuration and options

- **SXT** - transmitter synthesizer configuration and options
- **LimeLight & PAD** - input and output data stream options (LimeLight) and power amplifier driver options
- **TxTSP** - transmitter transceiver signal processor options
- **RxTSP** - receiver transceiver signal processor options
- **CDS** - clock distribution service options
- **BIST** - built in systems test options
- **TRX Gain** - alternative transmit and receive amplifier options
- **MCU** - direct control of on-chip microcontroller
- **R3 Controls** - fine tuning controls including compare voltages, RSSI options, and temperature compensation

F.2 API Libraries

- **ADF4002.py** - Analog Devices phase detector/synthesizer programming and control (chip datasheet: <https://www.analog.com/media/en/technical-documentation/data-sheets/ADF4002.pdf>)
- **LMS7002.py** - gateway to lower level API calls specific to the LMS7002M FPRF
- **LMS7002_AFE.py** - analog front end functions (ADC and DAC control/configuration)
- **LMS7002_base.py** - low level functions such as type conversion
- **LMS7002_BIAS.py** - bias functions
- **LMS7002_calibration.py** - high level calibration functions (calculates and sets parameters to comply with higher level configurations)
- **LMS7002_CDS.py** - clock distribution system functions

- **LMS7002_CGEN.py** - clock generator functions
- **LMS7002_CHIP.py** - general functions
- **LMS7002_DCCAL.py** - DC calibration functions
- **LMS7002_EVB.py** - used to control/program LMS7002 evaluation board at a high level (not the LimeSDR-Mini)
- **LMS7002_GFIR.py** - general finite impulse response filter functions (imported by each GFIR# class)
- **LMS7002_GFIR1.py** - first general finite impulse response filter class
- **LMS7002_GFIR2.py** - second general finite impulse response filter class
- **LMS7002_GFIR3.py** - third general finite impulse response filter class
- **LMS7002_IO.py** - input output functions (chip pads)
- **LMS7002_LimeLight.py** - limelight functions (IQ digital interface)
- **LMS7002_mSPI.py** - microcontroller serial protocol interface functions (for reading/writing internal memory registers)
- **LMS7002_NCO.py** - numerically controlled oscillator functions
- **LMS7002_RBB.py** - receive base band functions
- **LMS7002_regDataStructs.py** - register class
- **LMS7002_REGDESC.py** - multiline comment (very long) describing each register
- **LMS7002_REGDESC_MR3.py** - multiline comment (very long) describing each register. seems to have more than REGDESC.
- **LMS7002_RFE.py** - receiver front end functions
- **LMS7002_RxTSP.py** - receive transceiver signal processor functions

- **LMS7002_SX.py** - synthesizer (transmitter and receiver) functions
- **LMS7002_TBB.py** - transmit base band functions
- **LMS7002_TRF.py** - transmit radio frequency functions
- **LMS7002_TxTSP.py** - transmit transceiver signal processor functions
- **LMS7002_XBUF.py** - transmit and receive buffer settings (unrelated to LimeLight)
- **pyLMS7002Soapy.py** - high level chip functions/parameters
- **Si5351.py** - Silicon Labs i2c programmable any frequency clock generator programming and control (chip datasheet: <https://www.silabs.com/documents/public/data-sheets/Si5351-B.pdf>)
- **weakproxy.py** - roxy object to help garbage collection

F.3 Unused API Libraries

The following libraries, included with the pyLMS7002Soapy API, are not part of the import tree stemming from the pyLMS7002Soapy.py library. This is due to their specialized nature pertaining to different hardware (boards other than the LimeSDR-Mini or -USB), other chips, or are merely register documentation. Descriptions of these libraries are found in the previous section.

- ADF4002
- LMS7002_EVB
- LMS7002_regDataStructs
- LMS7002_REGDESC
- LMS7002_REGDESC_MR3
- Si5351

F.4 Other Useful Acronyms

The following are a few useful acronyms that are used heavily throughout the LimeSDR-Mini documentation and LMS7002M datasheet, but are not immediately defined.

- **TIA** - trans-impedance amplifier (current in, voltage out)
- **PAD** - power amplifier driver
- **ZIF** - zero intermediate frequency
- **AGC** - adaptive gain control

APPENDIX G. GENERATE LFMCW WAVEFORM MATLAB SCRIPT

This script generates a binary waveform file useable by the LimeSuite GUI. The waveform created is an LFMCW signal, however, it can be easily modified to generate any signal. The structure of the resultant file is interleaved real and imaginary 16 bit big endian samples. Since the LimeSDR-Mini has 12 bit DACs and ADCs, the maximum value stored in each 16 bit sample is $2^{12} - 1$. At the end of the script, three figures are generated alongside the output .wfm file for signal verification. Displayed are one period of the inphase transient signal, one period of the quadrature transient signal, and the power spectral density of one period of the signal. To use the file, the only parameters that should be changed are:

- `samples_per_second` - the current sample rate of the analog front end (ADCs and DACs which should be matched)
- `f0` - the starting frequency of the LFMCW signal (at baseband/ZIF)
- `fbw` - the chirp distance (i.e. the final frequency minus the starting frequency)
- `T` - the chirp duration
- `periods` - the number of periods included in the waveform (if the chirp duration is long, decrease this number to keep file size low)

These parameters are all found within the first thirteen lines of the script.

```
1 clear; clc; close all;
2
3 %% Parameters of Waveform Sampling
4 samples_per_second = 240000 * 128 / 2.44233; % sdr sample rate
   from waveform file
```



```

5 seconds_per_sample = 1 / samples_per_second; % time for each
    sample from waveform file
6 bits = 12; % sample depth
7
8 %% Signal Generation Params
9 f0 = 1e6; % base frequency for signal
10 fbw = 6e6; % bandwidth of linear chirp
11 Gs = (2^bits - 1); % signal amplitude
12 T = 100e-6; % chirp time
13 periods = 10; % number of periods to include in waveform
14
15 %% Signal Generation
16 t = 0 : seconds_per_sample : periods*T - seconds_per_sample; %
    time variable (correlates sample rate to signal time)
17 alpha = fbw*pi/T; % chirp rate for lfmw in rad/s
18 beta = 2*pi*f0; % base frequency for lfmw in rad/s
19 phi = 0; % initial phase for lfmw in radians
20 bb_lfmw = Gs*exp(1j*(alpha.*t.^2 + beta.*t + phi)); % baseband
    lfmw signal
21 I_lfmw = real(bb_lfmw); % pull out real portion as inphase
    data
22 Q_lfmw = imag(bb_lfmw); % pull out imaginary portion as
    quadrature data
23
24 %% Waveform File Generation
25 % interleave two signals (I and Q)
26 wfm = zeros(1, length(I_lfmw)*2); % initialize waveform
    variable
27 for i = 1 : 1 : length(I_lfmw) % work through each sample of I
    and Q

```

```

28     wfm(i*2 - 1) = I_lfmcw(i); % write in I
29     wfm(i*2) = Q_lfmcw(i); % write in Q
30 end
31
32 %% Write Waveform File
33 fid = fopen('lfmcw.wfm', 'w', 'b'); % open file in big-endian
    format
34 fwrite(fid, wfm, 'int16'); % write wfm variable as 2-byte
    chunks
35 fclose(fid); % close file
36
37 %% Display One Period of Data
38 figure(); % open new figure
39 plot(t(1:floor(length(t)/periods)), I_lfmcw(1:floor(length(t)/
    periods))); % plot one period of signal
40 grid on;
41 xlabel('Sample in Time (s)');
42 ylabel('Code (binary)');
43 title('Baseband LFM CW Inphase Signal');
44
45 figure(); % open new figure
46 plot(t(1:floor(length(t)/periods)), Q_lfmcw(1:floor(length(t)/
    periods))); % plot one period of signal
47 grid on;
48 xlabel('Sample in Time (s)');
49 ylabel('Code (binary)');
50 title('Baseband LFM CW Quadrature Signal');
51
52 [f, fft] = plottableFFT(bb_lfmcw(1:floor(length(t)/periods)), T
    , 0); % calculate fft for one lfmcw period

```

```

53 fft_power = abs(fft)/max(abs(fft)); % convert fft to power
    spectral density
54 fft_power = 10*log10(fft_power.^2); % convert psd to log scale
55 f_mhz = f / 1e6; % convert frequency range to mhz
56 figure(); % open new figure
57 plot(f_mhz, fft_power, 'LineWidth', 2); % plot the PSD
58 psd_max = max(fft_power); % find maximum power
59 psd_min = min(fft_power); % find minimum power
60 psd_range = psd_max - psd_min; % find power range
61 axis tight; % remove empty space in x and y axes
62 ylim([psd_min - psd_range/25, psd_max + psd_range/25]); %
    dynamically add y buffer for viewability
63 grid on;
64 xlabel('Frequency (MHz)');
65 ylabel('Normalized Signal Power');
66 title('Power Spectral Density of Generated Baseband LFM CW');

```

APPENDIX H. EXAMPLE API SCRIPT

This is a very simple example script to show a few reading and writing operations using the pyLMS7002Soapy API. Additionally, the sdrSnapshot function can be useful when debugging high level configuration.

```
1 from pyLMS7002Soapy import pyLMS7002Soapy as pylmss
2
3 # function to printout high level configuration of a given SDR
4 def sdrSnapshot(sdr):
5     print(f'\n***** ALL INFORMATION STORED IN CLASS OBJECT
6     *****\n{sdr}')
7     print(f' Board Name:\t {sdr.boardName}')
8     print(f' Clock Gen Freq: {round(sdr.cgenFrequency / 1e6,
9     2)} MHz')
10    print(f' Channel:\t {sdr.channel} (only applicable on
11    LimeSDR-USB)')
12    print(f' Freq Ref:\t {round(sdr.fRef / 1e6, 2)} MHz')
13    print(f' Previous Band: {sdr.previousBand}')
14    print(f' TDD Mode:\t {sdr.tddMode}')
15    print(f' Verbose Mode:\t {sdr.verbose}\n')
16    print(f' RX Gain:\t {sdr.rxGain}')
17    rxBW = -1
18    unset = True
19    if sdr.rxBandwidth != rxBW:
20        unset = False
21        rxBW = round(sdr.rxBandwidth / 1e6, 2)
```

```

19     print(f' {"*" if unset == True else " "}RX Bandwidth:\t    {
rxBW} MHz')
20     rxNCOfreq = 0
21     unset = True
22     if sdr.rxNCOFreq is not None:
23         unset = False
24         rxNCOfreq = round(sdr.rxNCOFreq / 1e6, 2)
25     print(f' {"*" if unset == True else " "}RX NCO Freq:\t    {
rxNCOfreq} MHz')
26     rxRffreq = 0
27     unset = True
28     if sdr.rxRfFreq is not None:
29         unset = False
30         rxRffreq = round(sdr.rxRfFreq / 1e6, 2)
31     print(f' {"*" if unset == True else " "}RX RF Freq:\t    {
rxRffreq} MHz')
32     rxSampRate = 0
33     unset = True
34     if sdr.rxSampleRate is not None:
35         unset = False
36         rxSampRate = round(sdr.rxSampleRate / 1e6, 2)
37     print(f'   RX Sample Rate:   {rxSampRate} MHz\n')
38     txGain = sdr.txGain
39     unset = True
40     if sdr.txGain != txGain:
41         unset = False
42         txGain = sdr.txGain
43     print(f' {"*" if unset == True else " "}TX Gain:\t    {
txGain}')
44     txBW = -1

```

```

45     unset = True
46     if sdr.txBandwidth != txBW:
47         unset = False
48         txBW = round(sdr.txBandwidth / 1e6, 2)
49     print(f' {"*" if unset == True else " "}TX Bandwidth:\t  {
txBW} MHz')
50     print(f'   TX NCO Freq:\t    {round(sdr.txNCOFreq / 1e6, 2)}
MHz')
51     print(f'   TX RF Freq:\t    {round(sdr.txRfFreq / 1e6, 2)}
MHz')
52     print(f'   TX Sample Rate:  {round(sdr.txSampleRate / 1e6,
2)} MHz\n')
53
54 if __name__ == "__main__":
55     # select first device by serial alphanumeric order
56     mysdr = pylms.pyLMS7002Soapy()
57     # printout SDR (high level) configuration
58     sdrSnapshot(mysdr)
59
60     # modify the receive mixing frequency
61     mysdr.rxRfFreq = 900e6
62     # modify the receive bandwidth (ADC sample rate)
63     mysdr.rxBandwidth = 30e6
64
65     # modify the transmit mixing frequency
66     mysdr.txRfFreq = 900e6
67     # modify the transmit bandwidth (DAC sample rate)
68     mysdr.txBandwidth = 30e6
69     # modify the transmit gain
70     mysdr.txGain = 2

```

```
71
72 # store the LMS7002 object
73 LMS7002 = mysdr.LMS7002
74 # run transmit calibration
75 LMS7002.calibration.txCalibration
76
77 # printout SDR configuration to see if changes were set
78 sdrSnapshot(mysdr)
79
80 # pause script
81 input()
82
83 # close out script
84 exit()
```