The Pennsylvania State University

The Graduate School

College of Engineering

ON THE DEVELOPMENT OF MODERN IONOSPHERIC SENSORS USING SOFTWARE-DEFINED RADIO TECHNIQUES

A Thesis in

Electrical Engineering

by

Alexander L. Hackett

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2013

The thesis of Alexander L. Hackett was reviewed and approved* by the following:

*Signatures are on file in the Graduate School.

Abstract

As the field of electronics continues its trend to becoming faster, smaller, and lower-powered, opportunities continue to open up for digital systems, that rival, if not exceed, traditional analog systems in performance, reliability, and affordability. Software-defined radio systems add reconfigurability and flexibility into the mix, making use of high-speed analog-to-digital converters, digital-to-analog converters, programmable logic devices, and advanced digital signal processing techniques to extend the capabilities of radio transmitters and receivers. Three remote sensor systems designed to study several upper atmospheric properties were developed at The Pennsylvania State University, taking advantage of the benefits software-defined systems have to offer through the use of the Universal Software Radio Peripheral (USRP) platform. The design, implementation, and operation of each of these sensors are presented, with preliminary results to validate their use and to encourage further development of software-defined radio techniques in the field of ionospheric science.

Table of Contents

List of Figures

List of Tables

List of Code Listings

Acronyms and Abbreviations

- 1PPS – One Pulse-per-Second
- AC – Alternating Current
- ADC – Analog-to-Digital Converter
- AM – Amplitude Modulation (broadcast radio)
- API – Application Programming Interface
- ASPIRL – Applied Signal Processing and Instrumentation Research Laboratory
- BPF – Band-pass Filter
- BPG – Bit Pattern Generator
- BPSK – Binary Phase-Shift Keying
- CADI – Canadian Advanced Digital Ionosonde
- CGI – Common Gateway Interface
- CIC – Cascaded Integrator-Comb
- CIRI – Cognitive Interferometry Radar Imager
- COCO – Coaxial-Colinear
- COTS – Commercial Off-the-Shelf
- CPLD – Complex Programmable Logic Device
- CPU – Central Processing Unit
- DAC – Digital-to-Analog Converter
- DC – Direct Current

- DDR2 – Double Data Rate, Type 2
- DDR3 – Double Data Rate, Type 3
- DDS – Direct Digital Synthesizer
- EDT – Eastern Daylight Time
- FFT – Fast Fourier Transform
- FIFO – First-In First-Out
- FM-CW – Frequency-Modulated Continuous Wave
- FPGA – Field Programmable Gate Array
- GB – Gigabyte
- GCS – GNU Chirp Sounder
- GMM – Gaussian Mixture Model
- GNU – GNU's Not Unix
- GPC – General-Purpose Computer
- GPIO – General-Purpose Input/Output
- GPS – Global Positioning System
- GPSDO – GPS-Disciplined Oscillator
- GPU – Graphics Processing Unit
- GTC – Transmitter Control Module
- GTS – Transmitter Supervisor Module
- GUI – Graphical User Interface
- HDF5 – Hierarchical Data Format 5
- HF – High-frequency
- HIF – Human Interpretable Format
- I – In-phase
- I/O – Input/Output
- I/Q – In-phase and Quadrature

- IEEE – Institute of Electrical and Electronics Engineers
- IF – Intermediate Frequency
- IIF – Instrument Interpretable Format
- IPP – Inter-pulse Period
- IRIS – Illinois Radar Interferometer System
- LAN – Local Area Network
- LPF – Low-pass Filter
- Mbps – Megabits per Second
- NCO – Numerically Controlled Oscillator
- NTP – Network Time Protocol
- O+X – Ordinary and Extraordinary
- OS – Operating System
- PARIS – PSU All-sky Radar Interferometry System
- PCB – Printed Circuit Board
- PISCO – PSU Ionospheric Sounder for Chirp Observations
- PSU – The Pennsylvania State University
- PTM – Pulse Transmitter Module
- PTS – Pulse Transmitter System
- PVC – Polyvinyl Chloride
- Q – Quadrature
- RAM – Random-Access Memory
- RF – Radio Frequency
- RPM – Revolutions per Minute
- RTI – Range-Time-Intensity
- RX – Receive
- SATA – Serial Advance Technology Attachment

- SDR – Software-defined Radio
- SKiYMET – Allsky Interferometric Meteor Radar
- SNR – Signal-to-Noise Ratio
- STL – Standard Template Library
- TB – Terabyte
- TR – Transmit/Receive
- TTL – Transistor–Transistor Logic
- TX – Transmit
- UHD – Universal Hardware Driver
- UIUC – University of Illinois Urbana–Champaign
- USB – Universal Serial Bus
- USRP – Universal Software Radio Peripheral
- VCTCXO – Voltage-Controlled Temperature-Compensated Crystal Oscillator
- VHF – Very High Frequency
- YAML – YAML Ain't Markup Language

Acknowledgments

This work would not have been possible without the support and efforts of the following people:

- *Dr. Julio Urbina* – for his support in every aspect of my academic career over the past six years, including being my thesis advisor. From my very first year at Penn State, he welcomed me into his research, provided tutoring years ahead of my coursework, traveled around the world with me, and afforded me with opportunities beyond what I ever imagined coming to college. Dr. Urbina has been and will continue to be a lifelong mentor and friend.

- *Dr. Sven Bilén* – for his founding, support of, and dedication to the Student Space Programs Laboratory, as well as support of my academic pursuits and being my thesis reader. Through the hands-on, real-world spacecraft design projects in SSPL made possible by Dr. Bilén, I have accumulated an enormous wealth of technical knowledge, skills, and friendships that have and will continue to drive my success.

- *Dr. John Mathews* – for his support of the PISCO project efforts and for being my thesis co-chair.

- *Dr. Erhan Kudeki, Dr. Steven Franke, Pablo Reyes* – for the development of the IRIS system and the technical support during the deployments of CIRI@PSU and CIRI@Andes.

- *Ryan Seal* – for design and implementation of the PARIS/GnuRadar and Radar Controller systems. He has remained an excellent technical resource and role model throughout my academic career.

- *Robert Sorbello* – for his support during the construction, deployment, and continued operations of CIRI@PSU and other lab operations.

- *Tejas Nagarmat* – for his prior efforts and guidance on the PARIS/GnuRadar system.

- *Zach Stephens* – for his development of Sauron and support during the deployment and initial operations of CIRI@PSU.

- *Burak Tüysüz* – for his technical support of CIRI@PSU and other lab operations.
- *Freddy Galindo* – for housing us during our trip to Peru and support of lab operations.
- *Karim Kuyeng Ruiz* – for her support with restoring the PARIS transmitter and continued operations of these systems.
- *Salih Bostan* – for his support of continued operations of these systems.
- *Ben Young* – for his efforts in implementation of multi-device operation with PARIS.
- *My parents, David and Lise Hackett* – for their endless encouragement and support throughout the years. I cannot express enough how much they have and continue to value my success, and for that I will always be grateful.
- *My girlfriend, Danielle Snyder* – for her encouragement and support throughout my undergraduate and graduate studies, and for keeping me grounded and sane during the difficult times.

Chapter 1

Introduction

Shortly after the first demonstrations of radio signal communication in the late nineteenth century, radio systems found application in the study of the Earth's atmosphere, namely the region named the *ionosphere* [1]. Since then, radio communications have exploded into numerous other applications, present in nearly every facet of our modern lives, and continued study of the ionosphere has never been more relevant. Deeply intertwined with space weather phenomena, the behavior of the ionosphere affects signals passing through it (e.g., Global Positioning System (GPS) signals, satellite television and radio signals, etc.), long-distance ground communications that make use of the ionosphere's reflective properties, and large ground-based networks, such as electric power grids [2]. While this field of study has matured significantly since its inception, many fundamental questions about ionospheric processes and behavior persist even today.

1.1 Motivations

The motivations behind the research presented in this thesis represent technical interests from both ionospheric sciences and the electrical/computer engineering disciplines. On the ionospheric science side, the formation and behavior of several different types of ionospheric layers, such as Sporadic-E and Spread-F, are not well understood and are an area of active research. In addition, the Earth is constantly showered by micrometeors (most too small to see with the naked eye), that, as they enter the atmosphere, excite the surrounding atoms and create localized regions of ionospheric plasma. Besides understanding the physical processes behind these meteor trails, the mass and velocities of the meteors can provide insight into the deposition of extraterrestrial matter on the surface of the Earth, as well as to indirectly study other atmospheric processes, such as neutrally charged upper atmospheric winds.

From an engineering point of view, this field of study represents an excellent opportunity to apply new digital signal processing techniques to a full suite of different ionospheric sensing instruments. The relatively low cost and flexibility afforded by software-defined systems allows for hardware reuse in different sensor applications, lowering costs and reducing hardware development time. Additionally, hybrid, multi-purpose sensors can be developed that share common hardware components. The modular design of software to control these flexible platforms and process incoming or outgoing signals enables rapid instrument prototyping, further reducing development costs. Furthermore, the software-defined approach encourages development of a flexible software framework for these types of sensor systems, promoting the development of advanced processing features previously impractical (or impossible) to realize in hardware.

1.2 Project Scope

This thesis discusses the application of software-defined radio technologies to ionospheric science in the context of three ionospheric sensor projects developed at The Pennsylvania State University (PSU), in collaboration with partners at Arecibo Observatory and University of Illinois Urbana–Champaign (UIUC), over the past few years. The PSU Ionospheric Sounder for Chirp Observations (PISCO) is a Universal Software Radio Peripheral (USRP)–based ionosonde receiver system developed using a commercial ionosonde transmitter system at Arecibo Observatory as a testbed. The project was funded for development of a low-cost, flexible high frequency (HF) radar receiver to be used with the HF ionospheric heating facility under development at Arecibo Observatory. However, because construction of the heater was delayed and operations of the facility have not yet begun, the commercial ionosonde system in place at Arecibo Observatory was used in the mean time, with basic hardware and software requirements expected to be similar for the two applications (although some details between the two systems differ significantly). Except for the GnuRadar software libraries used for USRP interface and raw data collection, all of the presented content relating to the PISCO receiver is entirely original work by the author.

The second ionospheric sensor presented is the PSU All-sky Radar Interferometry System (PARIS), a radar system designed to study the specular meteors by using a software-defined receiver. The project was begun several years prior to the author's experience with it.[1] Although relatively undocumented, the system-level architecture and hardware and software designs and implementations were to the level of basic functionality. The author's work with the system primarily involved system integration and basic testing and operations and,throughout the process, documenting the system for future efforts. Additionally, some hardware redesigns were made in an attempt to improve receiver sensitivity, and a framework for extending the data capture software to 5+ channels was implemented.

This thesis also discusses the Cognitive Interferometry Radar Imager (CIRI), another software-defined radar system designed to study a range of ionospheric phenomena, especially non-specular meteors and ionospheric layers. The CIRI project grew out of the Illinois Radar Interferometer System (IRIS), with the aim of extending the system to include "cognitive" features through a development and operations partnership between researchers at both The Pennsylvania State University and the University of Illinois. In order to develop and

[1]See references [3] and [4].

improve the various software components of the system, CIRI was deployed at the Rock Springs Radio Space Observatory, near The Pennsylvania State University (University Park) campus, involving set up of the antenna arrays, set up and integration of the transmitter and receiver hardware and software, and preliminary system operations. As was the case with PARIS, system documentation of CIRI/IRIS was limited, so this document attempts to provide a comprehensive reference of the system.

1.3 Thesis Overview

A brief overview of both the motivating ionospheric science and crucial software-defined radio technologies are discussed in Chapter 2. This chapter is intended to provide a reader with a basic understanding of the key concepts used in the work presented. The reader is encouraged to consult the References for more information as this thesis is certainly not an exhaustive resource on these topics.

Chapter 3 presents and details the system architecture, as well as the hardware and software designs and implementations for the PISCO, PARIS, and CIRI software-defined ionospheric sensor systems previously mentioned.

Some preliminary results from data taken by each of these systems are presented in Chapter 4. The results presented both validate the efforts on these projects hitherto and help to identify issues and missing features in each of these systems.

Finally, the work presented is summarized in Chapter 5. Additionally, suggestions for future work and improvements on these projects are outlined.

Appendix A presents a more extended reference of important code sections than available in the text of Chapter 3.

Preliminary procedures for various modes of operation in these different sensor systems, as well as some suggestions for system debugging, are listed in Appendix B.

Chapter 2

Background

This chapter aims to provide the reader with a basic background of some of the science and engineering concepts that lay the foundation for the three ionospheric sensors discussed in Chapter 3. Additionally, several previous and related software-defined ionospheric sensor efforts are discussed. Many of the details of these subjects have been omitted for brevity, as countless volumes have been published on each individual subject.

2.1 Ionospheric Science

Many important discoveries about the behavior of the ionosphere have been made since the field's inception, and this knowledge is relied upon by terrestrial and space-based systems around the world. Despite this, the proverbial "rabbit hole" of the ionosphere inevitably goes deeper than the past and present generations of scientists can imagine, so continued study is necessary to better understand the physical processes behind different ionospheric phenomena. The following sections briefly examine several topics of ionospheric science that the three sensor systems are designed to study, whether by answering the science questions directly or by paving the way for new generations of modern ionospheric sensors to continue the study. The operation of several typical instrument classes used for remote ionospheric

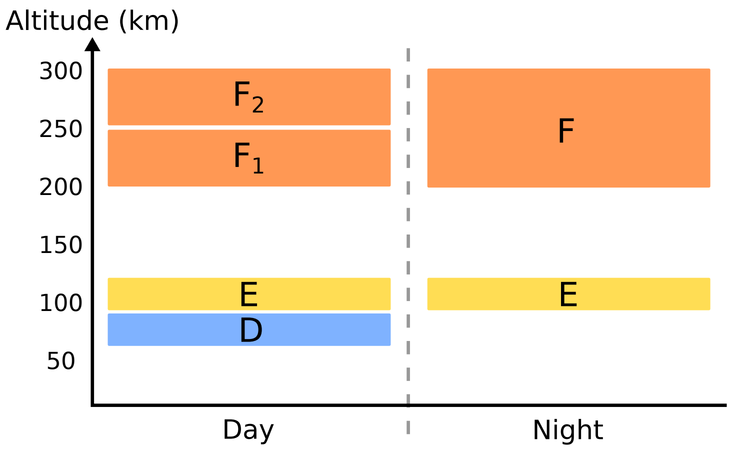


Figure 2.1: A generic representation of the different regions of the ionosphere.

sensing (under which the presented sensors fall) is also discussed.

2.1.1 Ionospheric Layers

The ionosphere is a region of Earth's upper atmosphere extending from about 75 km to beyond 500 km in altitude, markedly defined by the ionization of atmospheric constituents within this region [5]. It is subdivided into several regions based on electron density properties, namely the D, E, and F regions. Figure 2.1 shows a representation of these regions according to typical altitude ranges, both during the day and at night. Within these regions, layers of high localized ionization form, including the less common Sporadic-E and Spread-F events (discussed in the following sections), that can drastically affect radio-frequency (RF) communications.

The ionosphere's D region (i.e., "daytime" region) is typically defined to exist between 75 and 90 km, where ionization of atmospheric components occurs due to ultraviolet rays from the Sun, explaining its disappearance at night [1, 5, 7]. Although the region is ionized during the day, the relatively high neutral molecular density results in a fairly low net ionization and high absorption of RF energy.[1]

As illustrated by Figure 2.1, the E region of the ionosphere exists both during the day and at night, extending from around 100 km to 120 km in altitude [1, 5]. This region has a significantly higher electron density (several orders of magnitude) than the D region, but lower neutral density. RF signal absorption in this region is low, and, except during the presence of layers, RF signals, especially HF and very high frequency (VHF), can penetrate this region with little absorption.

The top-most region of the ionosphere, the F region, on average has a comparable electron density to the E region, although this can vary significantly with altitude, geographic location, and season [1, 5]. Typically the F region extends from around 200 to 300 km in altitude, although occasionally effects can be seen up to and beyond 500 km. Neutral density in this region is lower than both the D and E regions. During the day, the region consists of the F_1 and F_2 regions, supporting the formation of multiple F-region layers. At night, the two regions merge into a single F region, and usually only one layer is present.

2.1.1.1 Sporadic-E

As the name suggests, Sporadic-E layers (also called E_s layers) occur rather infrequently (although more during the summer months), and the causes behind and the formation mechanisms of these layers are not well understood. Forming within the E region, these layers are typically small and highly ionized, persisting on the timescale of minutes to hours [1, 6]. The very high electron density within Sporadic-E layers often permits HF and VHF radio signals to be reflected off these layers for very long distance, over-the-horizon communications, often utilized by amateur radio operators.

[1]The high absorption of the D region during the day is reason why distant amplitude modulation (AM) radio signals (and other HF communications) are typically stronger at night.

2.1.1.2 Spread-F

Besides Sporadic-E, another fairly infrequent ionospheric layer, Spread-F, is not well understood and is a current topic of several research efforts. Spread-F events appear as a thick (in altitude) F-region layer, "spread" across a large portion of the F region [6, 7, 8]. This type of event mainly at night, when the F_1 and F_2 regions have merged into a single F region. More studies have been conducted on Spread-F at (magnetic) equatorial regions and low latitudes, where its physical processes appear to be simpler, than at higher latitudes.

2.1.2 Meteors

Another topic of interest in ionospheric science is meteors. As Earth travels through space, it is constantly bombarded by dust particles and debris from a variety of extraterrestrial sources, including comets and remnants of meteor impacts with the Moon. As these particles (termed *micrometeoroids*) enter and accelerate through the Earth's atmosphere, atmospheric friction causes them to heat up and ablate [9]. During this process, a small disturbed region of ionospheric plasma forms around the meteor event.

When using radar systems to probe the ionosphere for these events (see Section 2.1.4.1), there are three main classes of reflections from these plasma regions that can be observed: specular trails, non-specular trails, and head echoes. Specular reflections occur when a meteor's trajectory passes through the radar beam perpendicular to the direction of the radar beam itself [10]. The resulting echo is a "mirror-like" reflection from the disturbed plasma, with an exponential power decay as the excited electrons recombine with ions.

Non-specular meteor trails and head echoes, resulting from magnetic field interactions with the plasma generated by meteor ablation, have only been more recently observed [10]. These reflections may require higher power radar systems for observation, as most of the reflected power is not scattered back to the observing radar system.

The properties of reflections from meteor events can be studied individually in order to gain a better understanding of the physics behind the meteor ablation process. They can also be studied statistically, in order to gain a better understanding of the larger scale properties, including diurnal, seasonal, and geospatial variables and total mass deposit due to meteors.

2.1.3 Upper Atmospheric Neutral Winds

In order to study wind patterns and neutral molecular composition in the upper atmosphere with ground-based remote sensing instruments, indirect methods must be used, as RF energy does not reflect and scatter off the neutral content of the ionosphere the same way it does off ionized regions [11]. One of these methods involves observing the statistical properties of meteors. By monitoring the position and velocity vectors of meteors over time, neutral wind properties can be inferred.

2.1.4 Instruments

In order to study the terrestrial ionosphere, a vast suite of different types of instruments can be used, including *in situ* measurements made by rocket, satellite, and balloon payloads, as well as ground-based remote sensing instruments. The scope of this discussion is limited to the latter, and specifically, the two classes under which the three sensor instruments presented fall.

2.1.4.1 Coherent Scatter Radar

The basic principle that governs the operation a pulsed radar system is that distance is directly related to time delay, through the propagation speed of a wave in a medium [12]. In the case of ionospheric radar systems, a transmitter sends pulsed RF energy through the atmosphere, where the signal reflects and scatters off any target in its path (e.g., meteor

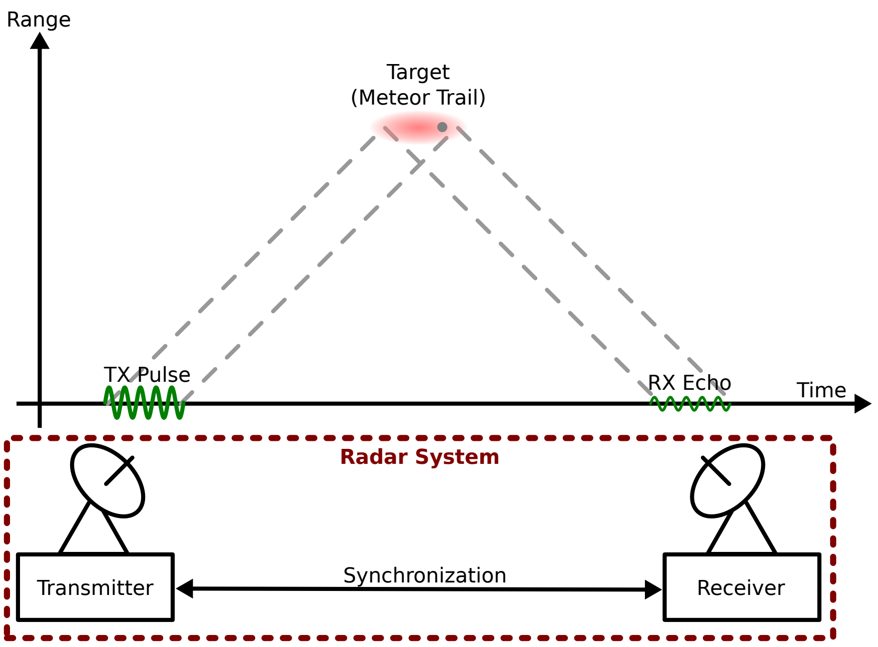


Figure 2.2: Basic operation of a pulsed radar system.

trails, ionospheric layers, airplanes, etc.). Some of this energy returns to the ground and is captured by the receiver, where a time delay between the transmitted pulse and the received echo is measured, which is proportional to the distance to the target. Figure 2.2 graphically illustrates this basic operation. The "coherent" descriptor refers to the receiver detecting the strong reflection that occurs at the target, representative of the transmitted pulse, without detecting the small "incoherent" effects on the signal caused motion of the ions and electrons within the plasma, a phenomenon that requires a high-power transmitter and very sensitive receiver.

The behavior of a particular radar can be summarized by a few key parameters, namely:

- *Inter-pulse Period (IPP)* – the regular time interval spacing between transmitted pulses. This parameter defines the maximum range that the radar system can detect without range aliasing effects. This parameter is sometimes called pulse-repetition interval (PRI).

- *TX Pulse Length* – the time length of the transmitted pulse. This parameter defines the range resolution, or the smallest range unit for target reflections.

- *Duty Cycle* – the ratio of TX pulse length to IPP. Some transmitters are limited to a maximum duty cycle for hardware protection.

- *TX Peak Power* – the peak transmitted power. This the signal power during the transmit pulse, ignoring the rest of the IPP.

- *TX Average Power* – the average transmitted power. This is the signal power averaged over a single IPP.

One of the basic trade-offs when selecting radar parameters involves the transmitted pulse length. A shorter pulse length gives the radar the ability to resolve smaller details, but it also reduces average power transmitted, reducing the power of the reflections (potentially below the system noise floor). Additionally, a shorter pulse length increases the spectral bandwidth (by properties of the Fourier Transform), which could be an issue, depending on operating conditions.

One technique commonly used to overcome the pulse length/average power trade-off is called phase-coded pulse compression [13, 14]. By phase-modulating the pulsed RF at subdivisions of the TX pulse length, known as bauds, with special binary codes, it is possible to obtain the range resolution of the shorter baud length, but retain the full power of the longer TX pulse. This binary phase modulation of the TX pulse is illustrated in Figure 2.3. The use of phase coding adds a few more radar parameters to the list:

- *Baud Length* – time length of one subdivision in the phase-coded TX pulse signal.

- *Phase Code* – the binary phase modulation signal. There are many different types of coding signals that can be used, but a key feature is a strong peak in the autocorrelation function of the code, and very small sidelobes.

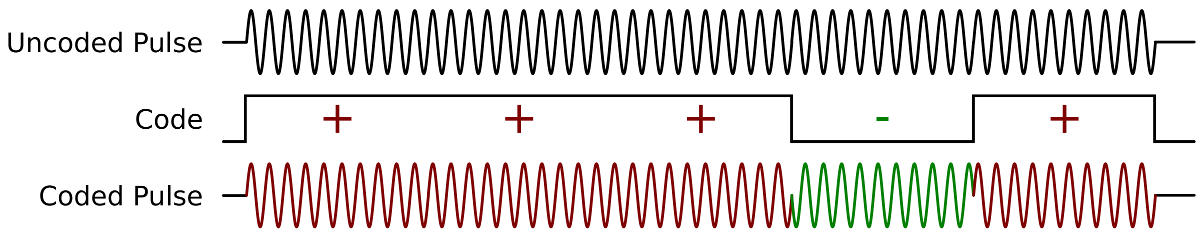


Figure 2.3: Phase-coded pulse compression using the Barker-5 code.

When the target reflection echo is received, the signal is still phase-coded, and must be decoded with a matched filter to take advantage of the applied coding [14]. In some cases, such as with meteor-head echoes, the use of phase coding can distort the reflection, so special care and additional processing must be used to correctly obtain the true reflection.

2.1.4.2 Ionosonde

An ionosonde is essentially a coherent scatter radar system that sweeps through a range of carrier frequencies during its operation, rather than just using a single fixed-frequency carrier[2] [5]. The different carrier frequencies penetrate or reflect off ionospheric layers, and the reflection frequency, amplitude, and altitude information is used to determine electron density information in the E and F regions of the ionosphere. The data taken by an ionosonde is used to generate an *ionogram*, a special radar graph that shows ionospheric layer reflections on a range-vs.-frequency plot. These ionograms can be studied independently or be used to provide auxiliary measurements about ionospheric conditions for other instruments.

2.2 Software-Defined Radio

The emergence of high-precision and (more importantly) low-cost analog-to-digital converters (ADCs) has fueled the explosive growth of a relatively new technology called software-defined

[2]There is also a type of ionosonde that uses a frequency-modulated continuous wave (FM-CW) instead of pulsed operation. The details are not discussed here, but the reader is referred to [15].

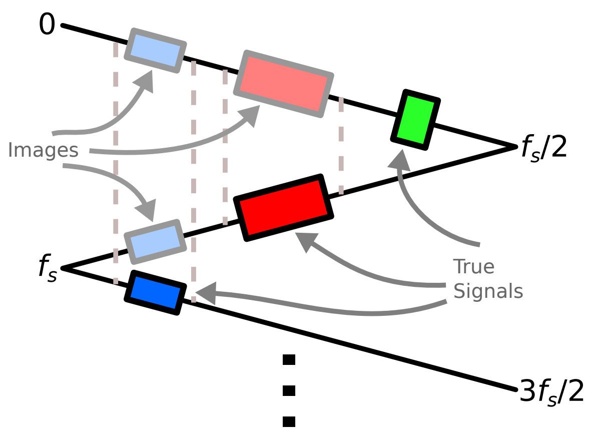


Figure 2.4: "Carpenter's ruler" diagram illustrating signal aliasing used for digital downconversion.

radio (SDR) [16]. The fundamental principle behind software-defined radio systems is the shift of signal processing elements that are traditionally realized in the analog domain (e.g., mixers, demodulators, filters, etc.) into the digital domain, with high-speed ADCs and digital-to-analog converters (DACs) providing the interfaces between the two domains. Once digitized, these signals can be processed on high-speed programmable logic devices, including field-programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs); in software on a host computer's central processing unit (CPU) or graphics processing unit (GPU); or, more commonly, a combination of both high-speed logic and host computer software.

There are many important concepts and tools that can be used in software-defined radio systems, and [17] provides an excellent reference on the subject. Only one of these instrumental tools is discussed here, and that is the concept of digital downconversion, which takes advantage of signal aliasing caused by sampling at a rate below the twice the Nyquist frequency (see [17]). Figure 2.4 shows a "carpenter's ruler" diagram that illustrates aliasing effects. Each leg of the ruler represents a frequency range of half the value of the sampling frequency (f_s). The green signal (right side) on the first leg of the ruler represents a signal that can be perfectly reconstructed as predicted by Nyquist. The red signal (middle), however, has spectral content between the sampling and Nyquist frequencies, resulting in a frequency-inverted image of the true signal to be aliased into the 0-to-Nyquist-frequency band (as well as all other legs of the ruler, though not shown on Figure 2.4 for clarity). Finally, the blue signal (left) lies above the sampling frequency, resulting in two lower-frequency images, with the baseband image having undergone frequency inversion twice, returning it to normal. In order to avoid image content overlapping with true signals in the baseband, external analog band-pass filters, called anti-aliasing filters, around the desired signal must be used.

Software-defined radio systems are advantageous over traditional hardware-defined systems in several ways, mentioned in Section 1.1, each of which is intertwined with the others. The system flexibility added by software-defined techniques is unparalleled in the hardware domain [16]. Instead of replacing physical hardware components to use a different frequency or modulation scheme, the system can simply be reprogrammed to incorporate this new operational mode.[3] That flexibility also directly translates into cost savings, allowing, if not encouraging, the same physical hardware to be used for multiple applications. In a similar vein, rapid instrument development and prototyping can occur on these software-defined platforms, drastically reducing the number of hardware revisions necessary before a final product is ready to be released. Software patches can be applied to systems operating in the field, introducing another level of cost savings. This list of advantages continues to grow as the software-defined technology field is developed further.

2.2.1 Universal Software Radio Peripheral

The Universal Software Radio Peripheral (USRP) is a family of extremely flexible, low-cost commercial software-defined radio hardware platforms developed and sold by Ettus Research (now owned by National Instruments) [18]. The systems use a modular design,

[3]This is within certain device limits, of course.

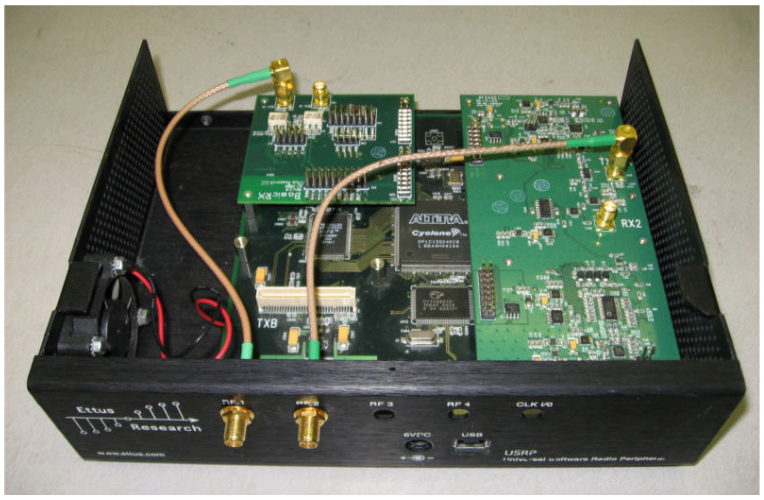


Figure 2.5: The Ettus Research USRP1 device, fitted with a BasicRX daughterboard (left side) and an RFX-400 daughterboard (right side).

with a single motherboard that contains the common processing elements, and a number of pluggable daughterboards, designed to cover a wide spectrum of frequency ranges and target applications. The motherboard uses high-speed ADCs and DACs to interface “real-world” analog signals with the high-speed, programmable digital signal processor, the consumer-grade FPGA. The digital signals are transmitted to and/or received from a host computer, where device control and lower bandwidth signal processing can occur.

USRP devices have been used for a very wide spectrum of applications, including, but certainly not limited to, hobbyist projects, academic and scientific research, and military communications systems. The systems presented in this thesis utilize the USRP as a receiver device for ionospheric science applications; however,the flexible, modular design of the USRP family allows these device to be re-purposed for vastly different applications within minutes, simply by swapping daughterboards and running different software.

There are a several different USRP platforms offered by Ettus Research (in addition to several sold by National Instruments). The original USRP device produced, the USRP1, pictured in Figure 2.5, has four 12-bit ADCs, two 14-bit DACs,[4] an Altera Cyclone-series

[4]There is a four channel maximum, e.g., two RX channels and two TX channels, or four RX channels.

FPGA, and a USB 2.0 interface to the host computer [18]. The device is clocked by an onboard 64-MHz crystal oscillator, although a few simple hardware modifications enable the use of an external clock.

The other devices offered by Ettus Research are a part of the USRP2 class of devices (as successors to the USRP1), including the N-series (networked – uses a Gigabit Ethernet interface for host communication), the B-series (bus – uses a USB 2.0 interface for host communication), and the E-series (embedded – features embedded Linux on an onboard microcontroller) [18]. The sampling rates and hardware specifications vary by device; however, all USRP2 devices uses a Xilinx Spartan 3A-series FPGA and support two RX channels, two TX channels, or one TX and one RX channel. Additionally, these devices support external clock and pulse-per-second (PPS) synchronization natively (without hardware modifications).

2.2.1.1 Tuning Frequency Errors

The USRP devices are tuned to a specific operating frequency by using a numerically controlled oscillator (NCO) within the FPGA design. The following analysis discusses timing and frequency errors due to the limited resolution (32 bits) of the NCO [19].

The frequency resolution, δf, of the USRP's NCO is:

$$\delta f = \frac{f_s}{2^{l_{tw}}}, \tag{2.1}$$

where f_s is the sampling rate of the USRP device and l_{tw} is the length of the tuning word (32 bits). In the case of the USRP1, where $f_s = 64$ MHz, the frequency resolution is $\approx$ 0.0149012 Hz. For the USRP2 devices, where $f_s = 100$ MHz, the frequency resolution is $\approx$ 0.023283 Hz. For most frequencies, the USRP devices cannot be tuned *exactly* and a residual frequency error results,

$$\epsilon = f_d - \left\lfloor \frac{f_d}{\delta f} \right\rfloor \delta f = f_d - \left\lfloor \frac{f_d}{f_s} 2^{l_{tw}} \right\rfloor \frac{f_s}{2^{l_{tw}}}, \tag{2.2}$$

where f_d is the desired tuning frequency and $\lfloor \cdot \rfloor$ takes the integer part of the argument (floor rounding). For a desired frequency of 21.4 MHz on the USRP1, the frequency error is $\approx$ 0.00894 Hz. Similarly, on the USRP2, the frequency error is $\approx$ 0.00800 Hz. These frequency errors can be eliminated by accounting for the error in software or by choosing an *exact* tuning frequency.

Frequencies that can be tuned *exactly* on the USRP, with no frequency error, must be integer multiples of δf that result in integer (in Hz) frequencies. The smallest integer multiple that fits this criterion can be found by taking the prime factorizations of the divisor and dividend in Equation 2.1 and simplifying the resulting fraction. The numerator of the result is the smallest tuning interval, in Hz, and the denominator is the digital count that corresponds this frequency. For example, in the case of the USRP1,

$$\frac{64\text{ MHz}}{2^{32}} = \frac{2^{12} \cdot 5^6}{2^{32}} = \frac{5^6}{2^{20}} = \frac{15625}{1048576}, \tag{2.3}$$

resulting in a minimum *exact* tuning interval of 15.625 kHz, corresponding to a digital count (tuning word) of 1048576. In the case of the USRP2,

$$\frac{100\text{ MHz}}{2^{32}} = \frac{2^{8} \cdot 5^8}{2^{32}} = \frac{5^8}{2^{24}} = \frac{390625}{16777216}, \tag{2.4}$$

resulting in a minimum *exact* tuning interval of 390.625 kHz, corresponding to a digital count of 16777216.

2.2.1.2 Dynamic Range and Noise Figure

Using the analysis in [19] as an example, the theoretical dynamic range and noise figure of the USRP devices can be calculated, which is necessary for determining the required receive RF front-end gains and noise figures for both PARIS and CIRI.

The maximum input voltage (without ADC saturation) to a USRP device is 2 V_{pp} (0.707 V_{RMS}), which, in a 50-Ω environment, corresponds to a maximum input power +10 dBm. The peak-to-peak value can be divided by the number of quantization values (2^{12} for the USRP1, 2^{14} for the USRP2) to obtain quantization step sizes (δ) of 488.3 µV and 122.1 µV, for the USRP1 and USRP2, respectively. Assuming that quantization errors can be modeled as uniform random distributions over $\pm\frac{\delta}{2}$, the root mean square (RMS) quantization noise is:

$$\sigma_q = \left(\frac{1}{\delta}\int_{-\delta/2}^{\delta/2} x^2 dx\right)^{\frac{1}{2}} = \left(\frac{\delta^2}{12}\right)^{\frac{1}{2}}. \tag{2.5}$$

This evaluates to 141.0 µV_{RMS} and 35.24 µV_{RMS} for the USRP1 and USRP2, respectively. Maximum theoretical dynamic range can then be calculated as the ratio of maximum to minimum input signal levels, i.e.,

$$\text{DR} = 20\log_{10}\left(\frac{V_{\max}}{\sigma_q}\right) = 20\log_{10}\left(\frac{0.707\ \text{V}}{\sigma_q}\right). \tag{2.6}$$

So, for the USRP1, the dynamic range is 74 dB, and for the USRP2, the dynamic range is 86 dB. The minimum noise floor can also be calculated from the quantization noise level:

$$P_{\text{floor}} = 10\log_{10}\left(\frac{P_q}{1\ \text{mW}}\right) = 10\log_{10}\left(\frac{\sigma_q^2}{(50\ \Omega)(1\ \text{mW})}\right), \tag{2.7}$$

resulting in −64.0 dBm for the USRP1 and −76.0 dBm for the USRP2. Finally, under the assumption that the quantization noise is uncorrelated and uniformly spread over the Nyquist bandwidth, the noise figure of the ADCs can be calculated using the power spectral density and standard noise temperature of (290 K $\rightarrow$ -174 dBm/Hz) as

$$\mathrm{NF}_{\mathrm{ADC}} = P_{\mathrm{floor}} - 10\log_{10}\left(\frac{f_s}{2}\right) - 174 \text{ dBm/Hz}. \tag{2.8}$$

This expression evaluates to 34.9 dB and 21.0 dB for the USRP1 and USRP2 devices, respectively.

2.2.2 GNU Radio

USRP devices are commonly used with a software package called GNU Radio. It is a powerful open-source signal processing framework for developing digital (and software-defined) radio systems [20]. Analogous to analog RF hardware design, high-speed signal processing blocks, such as mixers, demodulators, digital filters, amplifiers, signal visualization tools, etc., are connected together to form a complete signal flowgraph. Each flowgraph has one or more signal "sources" and one or more signal "sinks," both of which could be data file interfaces, sound card interfaces, or USRP device interfaces through the Universal Hardware Driver (UHD) software provided by Ettus Research. The high-speed signal processing blocks are written in C++, and they can be connected up in C++, Python (through a software wrapper interface), or using a drag-and-drop graphical user interface (GUI), called GNU Radio Companion (front end for generating Python flowgraphs). As an open-source software package, there is a very active and helpful online community of GNU Radio users with a large knowledge base on both GNU Radio and the USRP devices, providing support for beginners and advanced users alike.

2.3 Previous and Related Work

This chapter concludes by briefly reviewing some of the previous and concurrent work done on applications of software-defined radio to ionospheric science.

2.3.1 SKiYMET

The All-sky Interferometric Meteor Radar (SKiYMET) is an advanced meteor radar developed by Genesis Software[5] and MARDOC Inc. [21]. The system is a fully-integrated radar system, providing a solid-state transmitter, five-channel receiver, frequency synthesizer, synchronous radar timing generator, and control and data analysis software necessary for a operating a meteor radar system. SKiYMET is a commercially available system intended for rapid deployment and both short- and long-term operations.

2.3.2 Cobra

Cobra is all-sky interferometric meteor radar system developed by researchers at the University of Colorado at Boulder [12]. The system uses five Yagi antennas for interferometric reception, in the same configuration as PARIS (see Section 3.2.2.1.1), and four Yagi antennas for transmission. The system has been successfully deployed and operated at various VHF frequencies in Colorado, Alaska, and at the South Pole.

2.3.3 GCS

The GNU Chirp Sounder (GCS) is a USRP-based ionosonde receiver, similar to the PISCO project (see Section 3.1), developed at the Sodankylä Geophysical Observatory in Finland [22]. The system listens for ionospheric reflections from nearby frequency-modulated

[5]This is the same company that designed and built the transmitter for CIRI (see Section 3.3.2.1.2)

continuous-wave (FM-CW) ionosonde systems, and generates amplitude and phase ionogram plots, making use of the dual polarization antenna. The system can track and process multiple ionosonde transmitters simultaneously.

2.3.4 Open Radar Initiative

In the spirit of open-source software, the Open Radar Initiative is a community of scientists and engineers developing radar systems for ionospheric studies, who release the hardware and software designs of parts or all of their systems online freely [23]. The goal of the group is to reuse and build upon each other's work in order to reduce the amount of duplicated efforts and freely collaborate with others in the community.

Chapter 3

System Designs

Several different ionospheric sensor systems have been developed at The Pennsylvania State University (PSU) by the Applied Signal Processing and Instrumentation Research Laboratory (ASPIRL) using varying levels of software-defined radio techniques. Three of these projects are presented in the subsequent sections of this chapter. Each of these systems utilizes a USRP as the central digital receiver device, with additional supporting hardware and software specific to the application.

3.1 PSU Ionospheric Sounder for Chirp Observations

The PSU Ionospheric Sounder for Chirp Observations (PISCO) receiver is a USRP-based ionosonde receiver system developed at The Pennsylvania State University and tested and currently operating at Arecibo Observatory. The system uses commercial off-the-shelf (COTS) hardware and open-source software to provide receiver control, digital signal processing, and data collection, display, and storage for listening to ionospheric reflections from ionosonde transmitters (also called sounders and chirpers). Table 3.1 lists the design requirements for the PISCO receiver system. The design presented herein was carefully developed to meet these design requirements.

Table 3.1: Design requirements for the PISCO receiver system.

ID	*Requirement*
1	Develop the system from low-cost, COTS components
2	Minimize necessary hardware by taking advantage of SDR techniques
3	Design the system to be flexible for different deployment configurations

3.1.1 System Overview

Figure 3.1 shows a high-level systems diagram of both the ionosonde receiver and how it interacts with the Canadian Advanced Digital Ionosonde (CADI) system at Arecibo Observatory. The CADI system is a standalone ionosonde system, complete with a synchronized transmitter, receiver, and host computer software for data storage, processing, and display. It uses two co-located antennas, one for transmission and one for reception.

The PISCO receiver system is composed of four main hardware elements: 1) the active antennas responsible for transducing the RF energy to electrical signals; 2) the USRP1 responsible for digitization and preliminary signal processing of the received signals; 3) the direct digital synthesizer that provides a stable reference clock for the USRP1; and 4) the host computer responsible for controlling the USRP1 device, as well as data collection, processing,

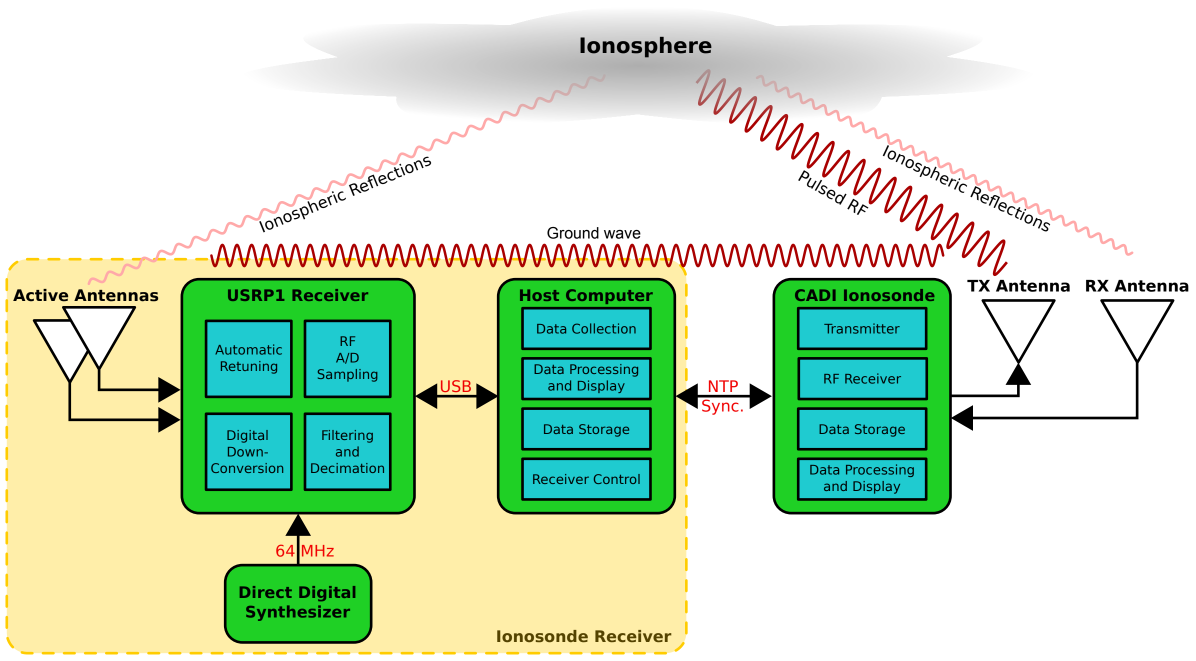


Figure 3.1: Overview of the PSU Ionospheric Sounder for Chirp Observations system used with the Canadian Advanced Digital Ionosonde.

display and storage. The hardware and software for these components are discussed further in Section 3.1.2.

When the CADI system pulses RF energy straight up into the ionosphere, some energy is reflected back to CADI, as well as in other directions due to scattering of the electromagnetic waves by ionospheric plasma. Some of this energy is captured by the antennas of the PISCO system, allowing it to "listen in" during CADI operation. Additionally, some of the pulsed RF energy from CADI propagates parallel to the ground (i.e., direct- or groundwave), and some of this energy, too, is captured by the PISCO antennas. Detection of this groundwave is critical for resolving reflection distance and is discussed further in Section 3.1.2.2.4.

It is important to note that the only electrical connection between the two systems is an Ethernet (IEEE 802.3) connection to the Arecibo Observatory local area network (LAN). Other than remote access, the sole purpose of this connection is for host computer clock synchronization via the Network Time Protocol (NTP), and even this could be replaced by some form of wireless time synchronization, such as GPS. In order to address Requirement 3, one of the major design goals of the PISCO receiver was to isolate the system from CADI as much as possible, thereby facilitating flexible operation of the system 1) at sufficiently distant locations from CADI, such that electrical delays in hard-wired cables prevent proper synchronization; 2) at distant locations from CADI where hard-wired cables between the systems would not be feasible; and 3) with transmitters other than CADI, which may not have a compatible electrical interface. While hardware synchronization (i.e., the use of discrete timing signals) between the two systems may be the simplest and most straightforward solution conceptually, it could be very difficult to implement in practice, depending on the setup of the two systems. Fortunately, modern computing power and signal processing techniques enable an offline "reconstructive" synchronization in software that rivals hardware synchronization in accuracy, without the need for a physical electrical connection, thereby addressing one aspect of Requirement 2.

3.1.2 Implementation

The following sections detail the hardware and software components from which the PISCO receiver is constructed.

3.1.2.1 Hardware

As per Requirement 2, by taking a software-defined approach to the PISCO receiver system design, very little hardware was necessary to accomplish an ionosonde receiver with comparable performance to the CADI system in place at Arecibo Observatory. The following sections detail each of the different hardware components shown in Figure 3.2, all of which are COTS parts (see Requirement 1).

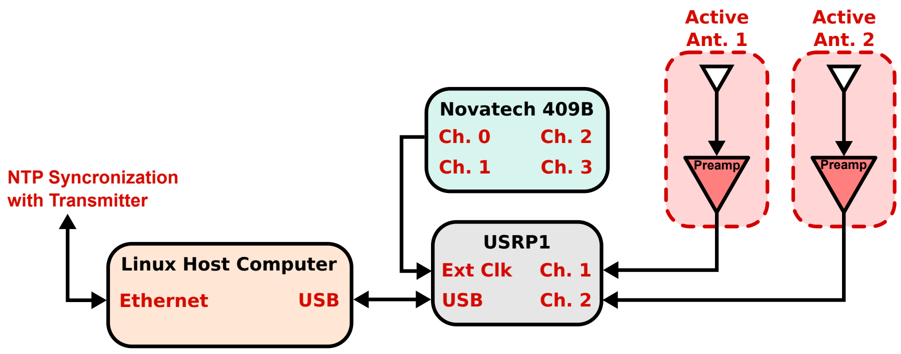


Figure 3.2: PISCO hardware diagram (power connections omitted).

3.1.2.1.1 Active Antennas The PISCO receiver uses two wideband active antennas for capturing the groundwave and ionospheric reflections from pulsed RF signal transmitted by CADI. The DX Engineering ARAH2-1P Active Horizontal Antenna (now replaced by the ARAH3-1P) is a dual-whip antenna with a built-in preamplifier, suitable for receiving between 100 kHz and 30 MHz [24]. Although advertised as ideal for amateur radio and shortwave listening, due to its sensitivity over the wide bandwidth that many ionosonde systems typically sweep, it turns out to be a useful antenna for an ionosonde receiver as well.

The active antennas are set up in cross pattern, with one aligned to the magnetic North–South direction and the other aligned to the East–West direction, as illustrated in Figure 3.3. Figure 3.4 shows the antennas as they are set up near the ionospheric heating facility at Arecibo Observatory.

3.1.2.1.2 USRP The core of the receiver hardware is the USRP1 device, which digitizes the incoming RF signals from the active antennas. A BasicRX daughterboard fitted into the USRP's RXA slot provides the two signal inputs to which the active antennas are connected. While these inputs are 50-Ω terminated and transformer coupled, they provide no signal conditioning and are virtually direct inputs to the USRP1's ADCs.

The sampling rate of the on-board ADCs is 64 MHz, which, by the Nyquist-Shannon Sampling Theorem, yields a sampling bandwidth of 32 MHz [17]. The usable frequency

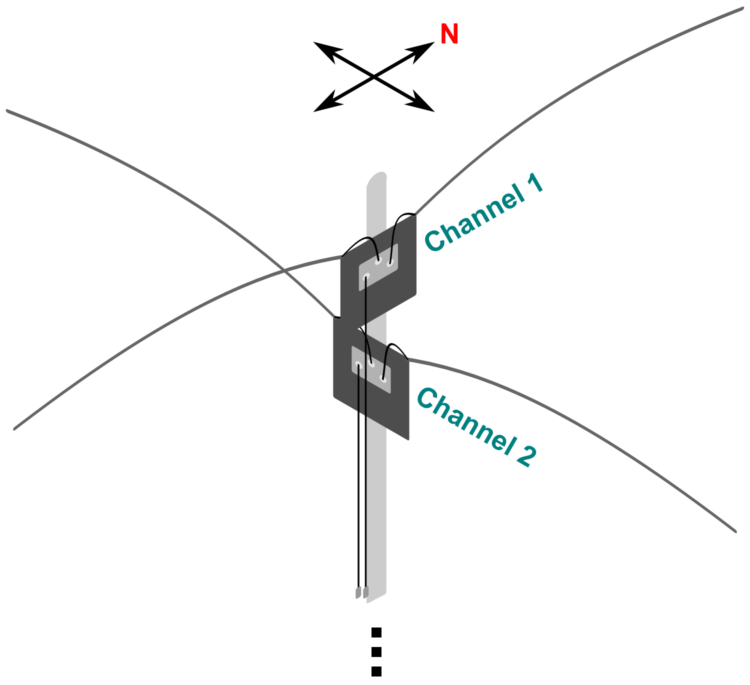
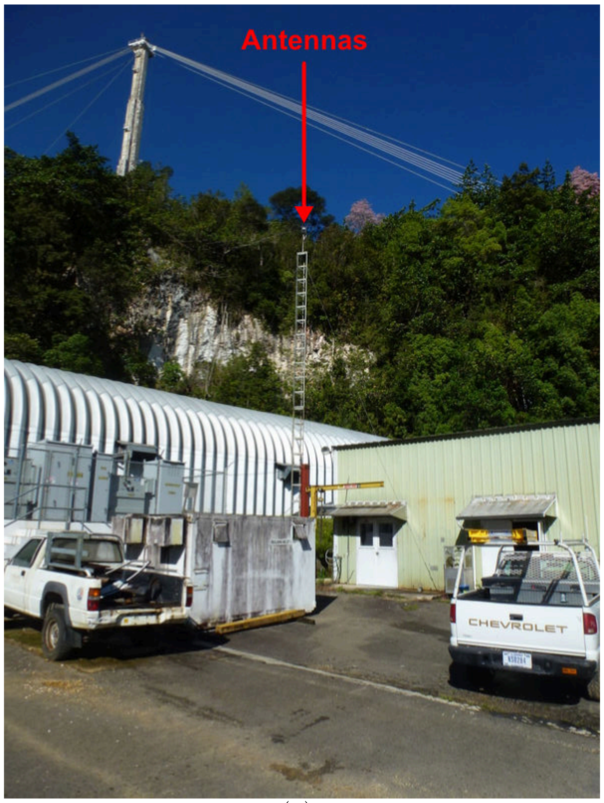


Figure 3.3: Setup of the ARAH2-1P antennas for the PISCO system (not to scale).

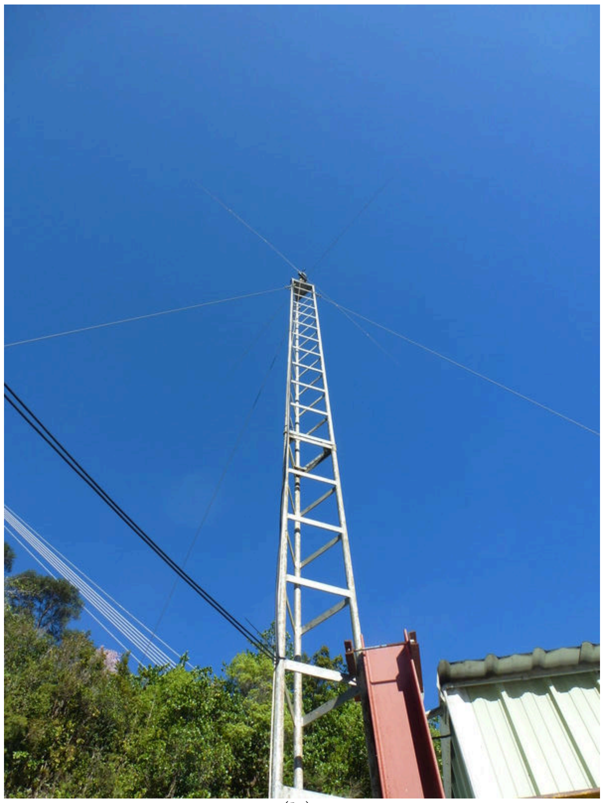
range for the BasicRX daughterboard is 1–250 MHz [18], limited on the low end by the transformer coupling of the input signal. Thus, the USRP system is capable of directly digitizing (with no downconversion) signals in the range of 1–32 MHz, which adequately covers the range of most ionosonde systems (typically 1–20 MHz) [5]. However, because the USB 2.0 bus, over which the USRP is connected, has a bandwidth limitation of 480 Mbps [25], the maximum sampling bandwidth that could theoretically be streamed continuously from the USRP is

$$\mathrm{BW}_{\mathrm{max}} = \frac{\mathrm{BW}_{\mathrm{USB}}}{\text{Sample Size}} = \frac{(480\text{ Mbps})}{(32\text{ bits})} \longrightarrow \mathrm{BW}_{\mathrm{max}} = 15\text{ MHz}\ . \tag{3.1}$$

The USRP sampling rate is 64 MHz, so data decimation must be performed on the FPGA in order to meet the USB 2.0 bandwidth requirement. However, only powers of two in the range [4,256] may be used for decimation in the `usrp1_fpga.rbf` FPGA image, resulting in



(a)



(b)

Figure 3.4: The ARAH2-1P antennas set up at Arecibo Observatory for PISCO, shown from (a) far away and (b) below.

a maximum bandwidth of 8 MHz. As long as the signal bandwidth does not exceed 8 MHz, the USRP center frequency can simply be tuned to the center frequency of interest, allowing the full 1–20 MHz band to be covered, albeit not all at once.

Although the USRP has an on-board 64-MHz crystal oscillator, it is prone to temperature and age drifting, and is bypassed in this application. Instead, an external clock source, the Novatech 409B Benchtop Signal Generator supplies the 64-MHz clock to the USRP (discussed further in Section 3.1.2.1.3). In order to configure the USRP to accept an external clock, several minor hardware modifications need to be made, as outlined in [26].

The PISCO receiver uses the `usrp1_fpga.rbf` FPGA image, provided with the UHD software package (and formerly with GNU Radio), modified to retune the device according to the predefined ionosonde transmitter behavior (frequency list and dwell time). The implementation of this is discussed further in Section 3.1.2.2.1. The standard FPGA image provide a programmable NCO and complex multiplier to produce a complex (in-phase and quadrature) data stream from the output of the each of the four ADCs [27]. Additionally, it provides cascaded integrator-comb (CIC) decimation and a half-band filter to reduce the effective sampling bandwidth to a rate that can be accommodated by the USB 2.0 interface to the host computer, over which data transfer and device configuration occur.

3.1.2.1.3 Direct Digital Synthesizer Because the USRP's 64-MHz crystal oscillator is prone to temperature drift and aging effects, the system uses an external direct digital synthesizer (DDS) to provide a stable clock source for the device. The Novatech 409B is a programmable oscillator with four independent outputs featuring programmable frequency, phase, and amplitude, an external reference input, and is accurate to ±1.5 ppm over 10 to 40 °C [28]. The device is programmed over a serial interface, either by writing directly to the raw serial device (e.g., `/dev/ttyS0`, or `/dev/ttyUSB0` for a USB/serial interface under Linux), or by using terminal emulator such as `minicom`.

Table 3.2: Basic Hardware Specifications of the PISCO Receiver GPC.

Processor Family	Intel Core-i7 960, 3.2 GHz
RAM	12 GB DDR3 (3×4 GB)
Hard Drive	1 TB, 7200 RPM SATA
Network Interface	1000 Mbps Ethernet

3.1.2.1.4 General-Purpose Computer The general-purpose computer (GPC) is responsible for data storage and processing, as well as control of the USRP device, discussed further in Section 3.1.2.2. In terms of hardware, any modern desktop or laptop computer with a USB 2.0 and Ethernet interface should be suitable for the GPC. The basic hardware specifications for the data storage and processing desktop computer on which the system was tested are listed in Table 3.2.

3.1.2.2 Software

The following sections discuss the operation of the various software components that work together in the PISCO system. This includes the automatic receiver retuning, data collection, processing, and storage, much of which has been moved from the traditional hardware domain to the software domain, in accordance with Requirement 2. Additionally, the flexible scheduling and scripting (see Requirement 3) that tie these components together are also discussed.

3.1.2.2.1 Frequency Sweeping The automatic frequency retuning of the USRP's FPGA is a critical process that enables practical sampling bandwidths while covering the entire band swept by the transmitter. Initially, the retuning process was implemented in software on the host computer, but the imprecise timing of the user space software (compared to an FPGA hardware implementation, for example) resulted in discontinuous data during the period when the retuning command was sent to the USRP over USB. A more accurate and robust FPGA-based solution was implemented to ensure continuous data (no gaps in the data), at

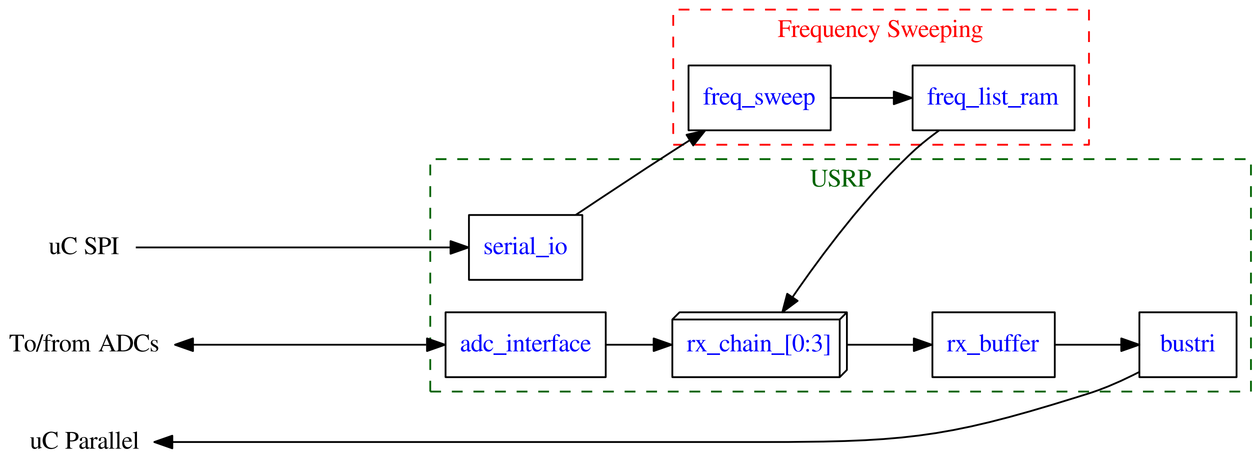


Figure 3.5: Simplified architecture of PISCO FPGA design.

the cost of a slight reduction in flexibility (hard-coded frequency tuning list).

The FPGA image used in the PISCO system is the `usrp1_fpga.rbf`, provided with the UHD software package (and formerly with GnuRadio), modified to retune the device according to the predefined ionosonde transmitter behavior (frequency list and dwell time). Figure 3.5 shows a simplified top-level architecture of the FPGA design.[1] The blocks boxed in green represent the modules provided in the standard FPGA design, and the blocks boxed in red represent the modules added for the frequency retuning functionality.

The FPGA receives data samples from the USRP's ADCs through the `adc_interface` module. This module is a driver that handles configuration, handshaking, and data transfer from the ADCs. The data output of the `adc_interface` module is wired to one or more `rx_chain` modules, depending on the configured number of receiver channels. Each instance of the `rx_chain` module contains a complex multiplier to generate in-phase (I) and quadrature (Q), a decimation stage, and a filter [27], conditioning the received signal to a usable bandwidth. The output of each of these `rx_chain` modules is wired into a single `rx_buffer` module, which prepares the data for streaming to the host computer by interleaving the

[1]There are several additional modules for control and input/output (I/O) not shown in Figure 3.5 and not discussed here. Refer to the UHD documentation for more information.

one or more channels into a single data stream. Finally, this data stream is wired to the `bustri` module, a tri-state logic module that acts as a driver to communicate with and transfer data to the USRP's USB microcontroller. Additionally, the standard design contains a `serial_io` module that enables live reconfiguration of many settings in the FPGA from the host computer.

The `rx_chain` module contains an NCO used for tuning the USRP. In the standard FPGA design, the frequency of the NCO outputs is determined by a frequency tuning word, set by the host computer through the `serial_io` module. In order to accomplish automatic retuning of the device, this tuning word connection was replaced with the output of a random-access memory (RAM) block (the `freq_list_ram` module) containing tuning words corresponding to the carrier frequencies of the transmitter. This RAM is addressed using some simple clock divider logic (discussed below) in the `freq_sweep` module. The frequency sweeping logic is controlled by several settings registers (one for resetting the logic and one for enabling the frequency sweeping operation), set by the host computer through the `serial_io` module.

Figure 3.6 shows a simplified flow diagram of the logic governing the access of the frequency tuning RAM, included in Listing 3.1. During each clock cycle, the flowchart is followed through the decision logic from the red "always" start block to one of the blue "action" blocks. After checking the reset (`sweep_reset`) and enable (`freq_sweep`) registers, a simple clock divider is used to control the frequency list address (`freq_list_addr`) that points to a tuning word in the `freq_list_ram`. The master clock is divided down using a counter (`switch_counter`) that counts up to the number of clock cycles spent at each frequency (i.e., the frequency dwell time), incrementing each clock cycle. This is calculated as

$$\text{SWEEP\_COUNTER\_MAX} = \frac{\text{Sampling Rate} \times \text{Total Sweep Time}}{\text{Number of Frequencies}} - 1. \tag{3.2}$$

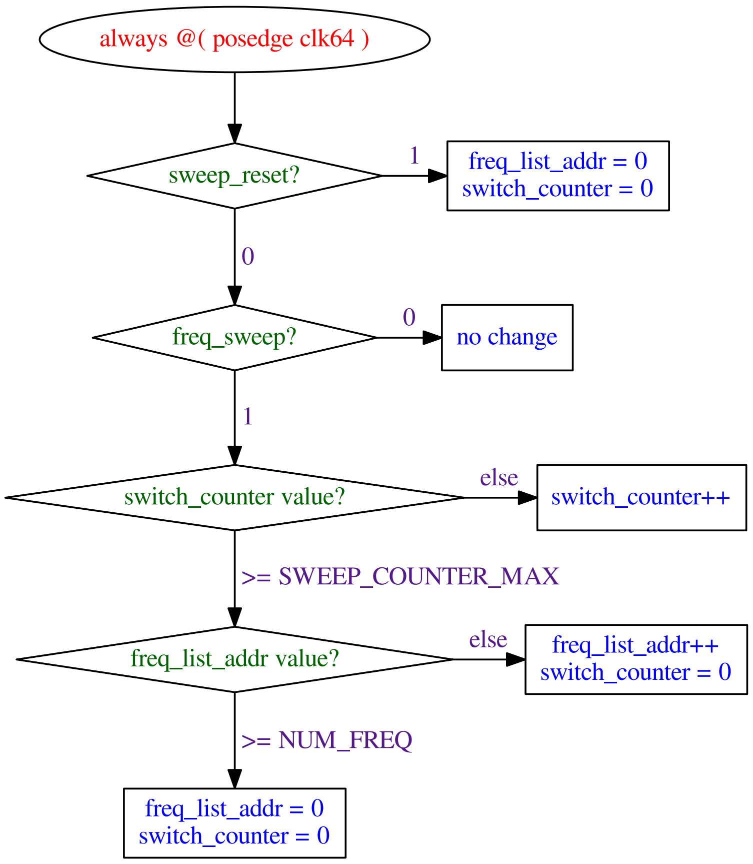


Figure 3.6: Simplified flow diagram for PISCO FPGA frequency retuning.

When the value of `switch_counter` reaches `SWEEP_COUNTER_MAX`, `freq_list_addr` is incremented, pointing to the next tuning frequency in the list, and the counter is reset. This behavior continues until the end of the frequency list is reached, at which point, both `switch_counter` and `freq_list_addr` are reset.

In the current configuration of the CADI system at Arecibo Observatory, 300 frequencies logarithmically spaced between 1 MHz and 20 MHz are transmitted, with a total sweep time lasting one minute. Thus, by Equation 3.2, `SWEEP_COUNTER_MAX` $= 12799999$, or equivalently, the frequency dwell time is 0.2 s.

It is important to note that the predefined tuning word values in the `freq_list_ram` module are not equivalent to the desired tuning frequencies. Rather, they can be calculated

Listing 3.1: Excerpt of frequency sweeping code added to `usrp_std.v`.

```
freq_list_ram arecibo (
    .address ( freq_list_addr ),
    .clock ( master_clk ),
    .data ( 32'd0 ),
    .wren ( 1'b0 ),
    .q ( ddc_tuning_freq )
)

always @( posedge clk64 )
begin
    if( sweep_reset )
    begin
        switch_counter <= #1 32'd0;
        freq_list_addr <= #1 9'd0;
        sweep_strobe <= #1 1'b0;
    end
    else if( freq_sweep )
    begin
        if( switch_counter == 32'd0 )
        begin
            sweep_strobe <= #1 1'b1;
            update <= #1 ~update;
            switch_counter <= #1 switch_counter + 1'b'1;
        end
        else if( switch_counter >= SWEEP_COUNT_MAX )
        begin
            if( freq_list_addr >= NUM_FREQ )
                freq_list_addr <= #1 9'd0;
            else
                freq_list_addr <= #1 freq_list_addr + 1'b1;
            switch_counter <= #1 32'd0;
            sweep_strobe <= #1 1'b0;
        end
        else
        begin
            switch_counter <= #1 switch_counter + 1'b1;
            sweep_strobe <= #1 1'b0;
        end
    end
end
```

by the following relationship [29]:

$$\text{Tuning Word} = \frac{\text{Tuning Frequency}}{\text{Sampling Rate}} \times 2^{\text{Tuning Word Width}}, \tag{3.3}$$

where the tuning word width is 32 bits. For example, for a tuning frequency of 1 MHz, the tuning word would be 67108864.

3.1.2.2.2 Data Capture The data capture software (`IonosondeRxRun`) is based heavily on the GnuRadar software libraries developed for the PARIS project (see Section 3.2.2.2), which provides the all the necessary framework for implementing a software radar receiver. `IonosondeRxRun` mimics the basic data collection and storage functionality of the GnuRadar program `gradar-run-server` without a network interface or data synchronization.

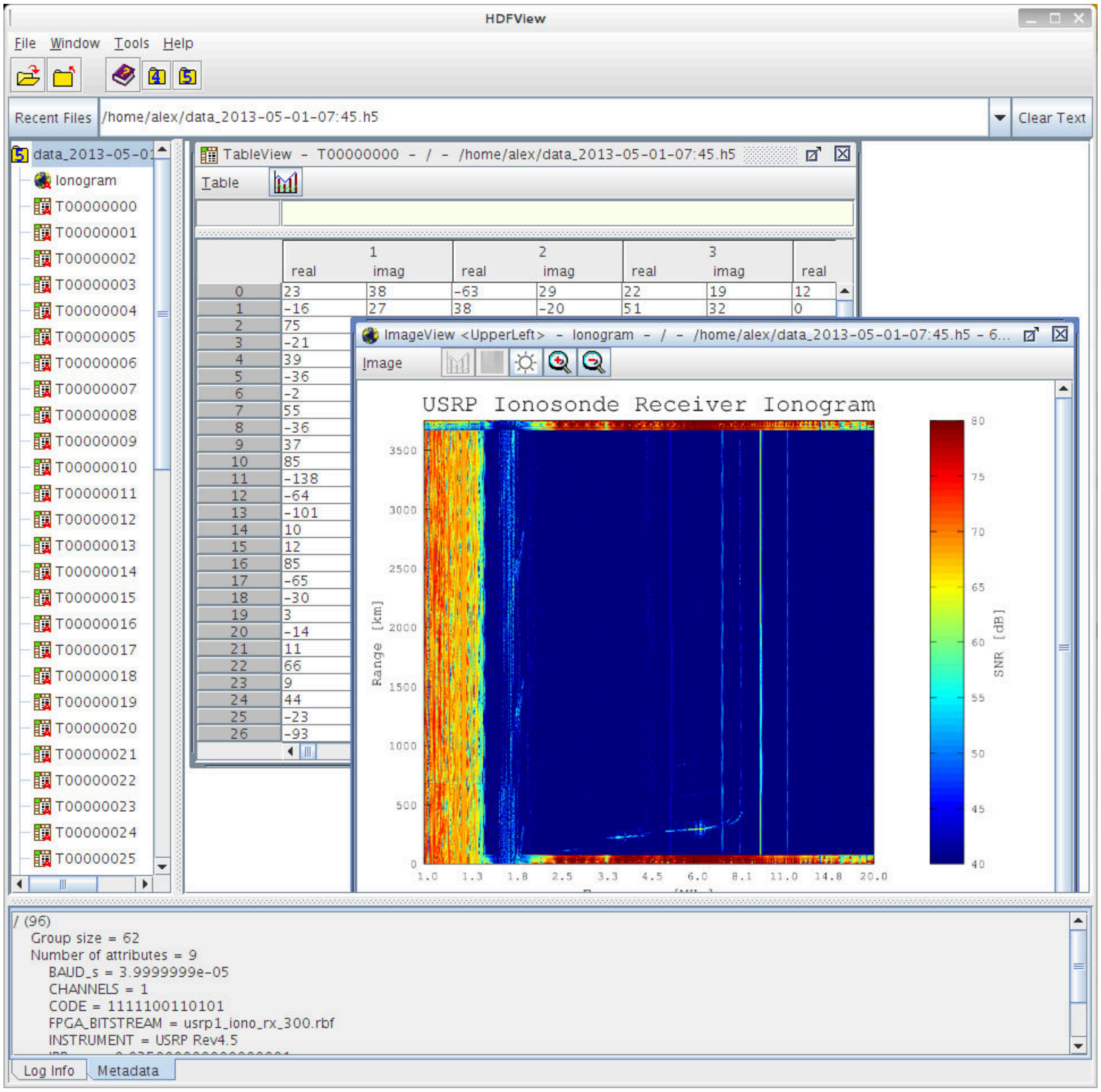
Following the object-oriented nature of the GnuRadar project, the `IonosondeRxDevice` class representing the receiver device is a child class of `GnuRadarDevice` and `SThread`, inheriting the important USRP interface functionality and multi-threading capabilities, while also adding member functions specific to the ionosonde application. The `GnuRadarDevice::RequestData()` inherited member function handles the USRP device interface, both sending configuration commands to the device and receiving data packets from the device. Although `GnuRadarDevice` expects to be receiving synchronized data, the modified `usrp1_fpga.rbf` image does not include a synchronization module (see Section 3.2.2.2.2), and the USRP freely transfers the continuous, unsynchronized data to the GPC.

When called, the virtual function `IonosondeRxDevice::Run()` writes a value to a register on the FPGA that sets the `freq_sweep` Boolean to logic high, starting the frequency sweeping mechanism. The function then calls the `SThread::Sleep()` function from its base class for a duration of the total sweep time (in the case of the installation at Arecibo Observatory, this is 60 s). Because of the multi-threaded implementation of `SThread`, the function does not put any load on the CPU, and other processes, such as the data collection, can be run simultaneously. After the timer expires, the thread is woken up and `IonosondeRxDevice::Run()` sets the `freq_sweep` Boolean to logic low, concluding the frequency sweeping.

3.1.2.2.3 Data Storage The raw in-phase and quadrature (I/Q) data from each ionosonde sweep is stored in a single Hierarchical Data Format 5 (HDF5) file (see [30]). This is a commonly used standard file format within the atmospheric sciences community. The metadata structure allows any number of arbitrary data tags to be stored alongside the data, which itself can be of mixed data structures (e.g., multi-dimensional tabular, image, and plain text). Numerous software application programming interfaces (APIs) exist for accessing HDF5 files, including C++, FORTRAN 90, Java, Python, and MATLAB [30, 31, 32]. Additionally, the program `HDFView` allows easy inspection of data structures, data tags, and data information itself for any HDF5 file. A screenshot of the program is shown in Figure 3.7.

The PISCO software stores the data as 32-bit complex integer (16 bits in-phase, 16 bits quadrature) in tabular form within the HDF5 file structure, with data from multiple channels interleaved at the sample granularity. The data tags stored include the following:

- Baud length, in seconds
- Number of channels
- Phase code
- Name of the USRP FPGA bitstream used for data collection
- The specific receiver device
- Inter-pulse period length, in seconds
- USRP sampling bandwidth, in hertz
- Data format of each sample
- Frequency dwell time, in seconds



	1		2		3		
	real	imag	real	imag	real	imag	real
0	23	38	-63	29	22	19	12
1	-16	27	38	-20	51	32	0
2	75						
3	-21						
4	39						
5	-36						
6	-2						
7	55						
8	-36						
9	37						
10	85						
11	-138						
12	-64						
13	-101						
14	10						
15	12						
16	85						
17	-65						
18	-30						
19	3						
20	-14						
21	11						
22	66						
23	9						
24	44						
25	-23						
26	-93						

```
/ (96)
  Group size = 62
  Number of attributes = 9
    BAUD_s = 3.9999999e-05
    CHANNELS = 1
    CODE = 1111100110101
    FPGA_BITSTREAM = usrp1_iono_rx_300.rbf
    INSTRUMENT = USRP Rev4.5
```

Figure 3.7: Screenshot of HDFView showing tabular and image data from the PISCO receiver.

These data tags provide important information for proper analysis and understanding of the data, for both the data plotting software and users of the data.

3.1.2.2.4 Data Processing The asynchronous nature of this receiver design necessitates added complexity in processing the data and generating meaningful plots of the I/Q data. Fortunately, modern computing systems have adequate processing power for these extra steps and to reconstruct the data in software. The `ionoplotter` program, consisting of Python and Octave code, takes these general steps to produce ionogram plots:

1. *Load the radar parameters and I/Q data from the HDF5 data file* – The `h5py` Python module is used to read the tabular I/Q data from the HDF5 data file into a single complex `numpy` array to pass to the plotter function. The array has dimensions of sample number × IPP, where sample number maps to range and IPP maps to carrier frequency. Additionally, the data tags containing the radar parameters (see Section 3.1.2.2.3) are stored into a Python dictionary structure, which is also passed to the plotter.

2. *Generate power map from I/Q data* – Because the receiver is not synchronized to the transmitter, the groundwave pulse needs to be detected and used as a ground reference. The I/Q data is read into the Octave plotter function (with the help of the `oct2py` Python module) and a power map is computed from the complex data samples as

$$P^2 = I^2 + Q^2. \tag{3.4}$$

   At this point, the groundwave pulse is apparent and can be used to align the I/Q data.

3. *Use the Hough Transform to find the groundwave pulse* – The Hough Transform is an image processing technique that facilitates feature extraction in images, e.g., straight

lines or circles, by mapping points in the original image into an accumulated transform space of parameters r and θ [33]. In this case, the feature of interest is a straight line with a relatively shallow angle from horizontal[2] (i.e., $< 10°$). It is important to search only for lines close to horizontal to avoid the vertical lines that result from background interference. Before performing the Hough Transform from the Octave `image` package, an edge filter is to reduce the power map to a binary image highlighting only sharp edges. The edged image is mapped into the accumulated transform space, and peaks appear corresponding to the most likely near-horizontal lines in the image.

4. *Shift I/Q data to align groundwave pulse to 0 km* – The strongest "line" candidate from the maximum peak in the accumulated transform space is assumed to be the groundwave pulse. Because the transform space is specified in polar coordinates, the slope and y-intercept of the groundwave pulse can easily be found by rearranging the following relationship

$$r = x\cos(\theta) + y\sin(\theta) \tag{3.5}$$

into slope–intercept form:

$$y = -\cot(\theta)x + r\csc(\theta) \Longrightarrow m = -\cot(\theta), b = r\csc(\theta), \tag{3.6}$$

where r and θ are the peak coordinates in the accumulated transform space. With the slope and intercept information, the I/Q data is circularly shifted to align the groundwave pulse to 0 km in range.

5. *Find frequency offset in each IPP, combine multi-channel data, and bring down to baseband* – Due to the lack of precise synchronization to the first pulse of the transmitter, some residual frequency offset between the transmitted carrier signal and receiver

[2]This slight slope arises from a constant frequency offset in either the Novatech signal generator or the equivalent sampling clock in the CADI system.

tuning frequency remains, manifesting as a slight amplitude modulation of the I/Q data that must be eliminated for proper signal decoding. For each IPP, the spectrum is calculated using the Fast Fourier Transform (FFT). Only a small range within the groundwave pulse is used for this in order to mitigate the background interference. The spectral peak for each IPP is used to generate a complex exponential for demodulation of that IPP, for each channel of data. Additionally, the complex exponential for one of the two channels contains a $90°$ phase shift term that is applied before the data from the two channels is summed. This results in the desired baseband information, with a 3-dB enhancement in signal-to-noise ratio (SNR) due to the combined channel data [34].

6. *Use a finite impulse response (FIR) filter to decode the Barker phase-coded signal* – The transmitted pulse is modulated using binary phase-shift keying (BPSK). This is a common technique used in radar systems to attain high range resolution without reducing signal power due to shorter pulses [13]. The CADI system uses a Barker-13 code, which gives excellent range resolution with minimal sidelobes. In order to decode the I/Q data, an FIR filter is used. The taps for this filter correspond to the elements in the Barker code, each with a multiplicity equal to the transmitted baud length multiplied by the receiver sampling rate.

7. *Generate final RTI power map and plot* – At this point, the I/Q data has been fully corrected and decoded, and a final power map is generated according to Equation 3.4. The IPP number axis is replaced by a logarithmically spaced frequency axis, as the two are synonymous in this system, and the range axis is calculated using the speed of light in free space ($c = 3 \times 10^5$ km/s) and the sampling rate. The resulting plot is saved as an image file to the hard disk.

8. *Save the plot back in the HDF5 file* – Again using `h5py`, the previously plotted ionogram image is stored back into the original HDF5 data file, as shown in Figure 3.7. This allows future users of the data to quickly preview the I/Q data content without the need to reprocess using the steps described above.

3.1.2.2.5 Scheduling Software In order for the PISCO system to properly listen for ionospheric reflections from an ionosonde transmitter, such as the CADI system, it must have *a priori* knowledge of precisely when the transmitter will begin pulsing. In the case of the CADI installation at Arecibo Observatory, ionosonde operation is scheduled to occur every 15 minutes, on the 15, 30, 45, and top of every hour.

Two tiers of schedulers are used for proper temporal alignment: the standard UNIX `cron` and a custom high-precision scheduler. `cron` provides an easy to use, very flexible, and robust scheduler system through the use of a plain-text configuration file, called a crontab (i.e., cron table) [35]. The crontab allows the user to schedule tasks arbitrarily, by month, day of month, day of week, hour, and minute, which is perfect for scheduling the 15-minute operation intervals, but it does not provide or guarantee a time resolution finer than one-minute intervals. This necessitates the use of a scheduler with better than one-second time resolution.

The high-precision scheduler (`Scheduler`) is a simple class containing only one member function, `Scheduler::Run()`, which is called immediately upon construction. Figure 3.8 illustrates the operation of `Scheduler::Run()`. This member function makes use of a timer function with microsecond resolution to schedule tasks to the top of the second. It does so by calculating the time difference (with microsecond resolution) between the scheduled execution time and the current time, halting execution of the program[3] with the `nanosleep()` function for exactly half of the calculated time difference. This process is repeated in a loop

[3]Note that `Scheduler` is not multi-threaded, contrasting the operation of `IonosondeRxRun`, discussed in Section 3.1.2.2.2. It is this single-threaded halting of execution that enables `Scheduler` to work properly.

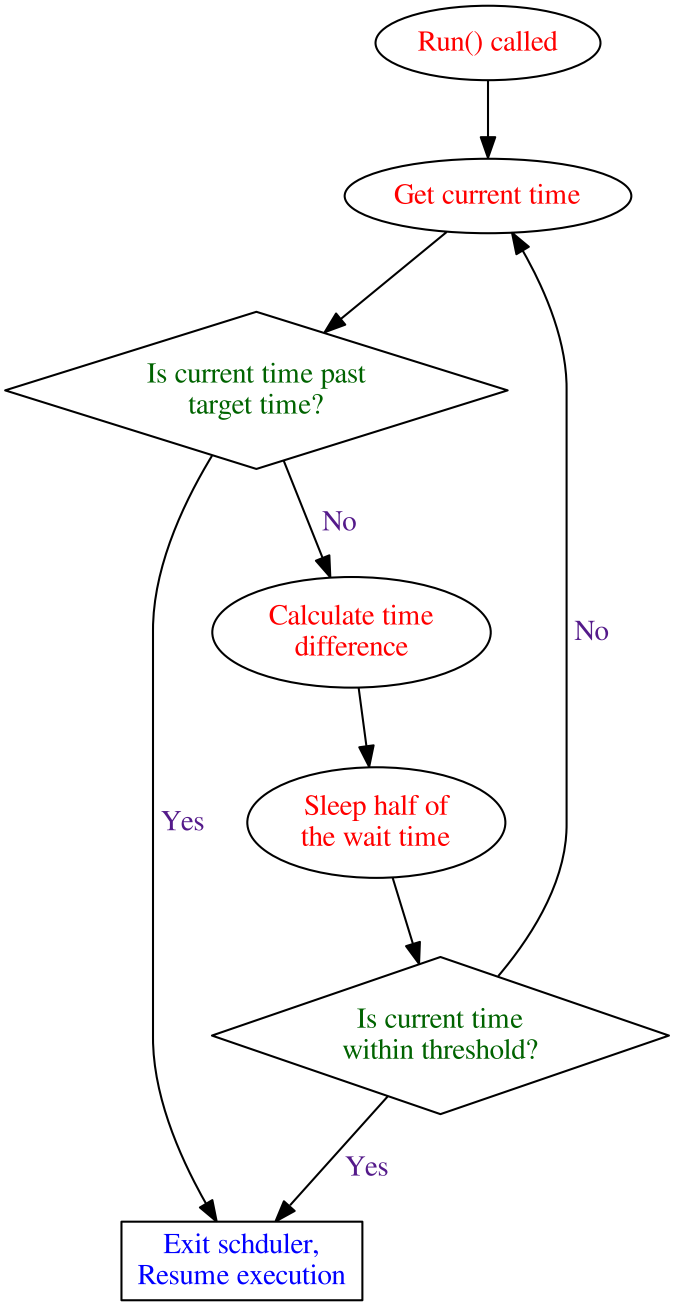

Run() called
Get current time
Is current time past target time?
No
Calculate time difference
Sleep half of the wait time
Is current time within threshold?
No
Yes
Yes
Exit schduler, Resume execution

Figure 3.8: Flow diagram of the high-precision scheduler developed for PISCO.

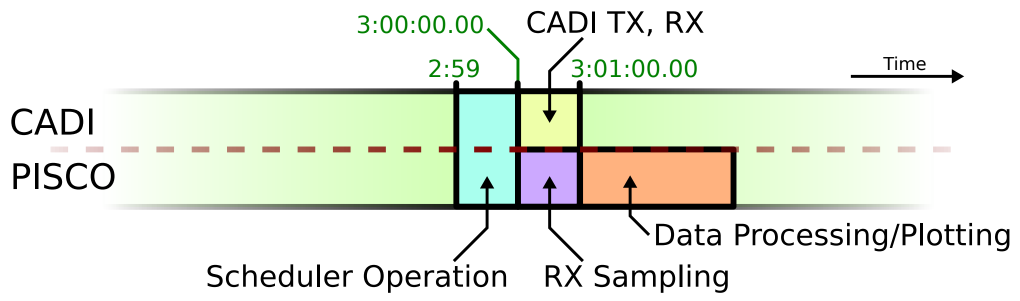


Figure 3.9: Scheduling of CADI and PISCO software.

until the time difference is within an acceptable threshold of the scheduled execution time. As a failsafe, the `Scheduler` will resume execution of the scheduled task immediately if the time difference is negative (i.e., if the scheduled execution time has already passed).

Both the PISCO and CADI host computers use the high-precision scheduler; however, in the case of PISCO, `Scheduler` is incorporated directly into the main program, `IonosondeRxRun`, whereas, because CADI is closed-source, a simple wrapper program was written to execute the CADI software upon completion of `Scheduler`.

Figure 3.9 pictorially represents the scheduling of CADI and PISCO operations. With time on the horizontal axis, the top half of the diagram shows CADI operations and the bottom half shows PISCO operations for a given ionosonde run, e.g., at 3:00 am. The crontab on both CADI and PISCO host computers schedules the high-precision scheduler for operation one minute prior to the ionosonde run. Due to the one-minute time resolution of `cron` this could potentially occur anywhere between 2:59:00 and 2:59:59. Once the high-precision scheduler is called on the CADI system, it schedules the CADI software to begin operation with approximately 10-ms precision. Because the CADI software includes some sort of short initialization before pulsing,[4] a "fudge factor" time constant (experimentally determined to be about 1.75 s) was introduced to account for this delay, thus beginning the

[4]This is an assumption based on the observable delay between calling the CADI software and when the groundwave pulse was visible on a spectrum analyzer.

ionosonde run at 3:00:00.00. Simultaneously, the receiver software on the PISCO system starts the receiver data collection from the USRP. Because the receiver software framework (GnuRadar) is open-source, the high-precision scheduler is able to be built into the receiver software (occurring after initialization), eliminating the need for a "fudge factor" time constant. After one minute, the CADI system finishes pulsing and the PISCO system stops data collection from the USRP. Finally, the PISCO data processing and plotting software is called, producing the range-time-intensity (RTI) plot discussed in Section 3.1.2.2.4.

3.1.2.2.6 Scripting Because there are many different pieces of software, each with their own specific function, that comprise the PISCO receiver, there is a need to tie everything together so that each program does not need to be run separately and manually. Fortunately, scripting languages are designed exactly for this purpose. Listing 3.2 shows the `ionorun` script used by the PISCO system to coordinate all the different software components on the GPC. The script begins by initializing and setting some time-based variables for filenames. After collecting data into `/data/IonosondeRx.h5` with the `IonosondeRxRun` data collection program, the script renames the data file to include a datestamp for easy cataloging. Then the data plotter (`ionoplotter`) software is run, generating the `/data/Ionogram.png` image file, which is renamed with a datestamp matching its corresponding data file. The data file is then compressed using the `bzip2` compression algorithm to save disk space. Finally, several symbolic links are created to the latest data file and plot, allowing easy access to the most recent data run.

3.1.2.2.7 Remote Access The general-purpose computer for this receiver system is set up to be accessible through `ssh` on TCP port 22 as the user `radar`. At the time of writing, the machine has been given a fixed IP address of 192.231.95.182. In order to access the machine, the connecting host should either be inside the Arecibo Observatory LAN or remotely connected via `ssh` to `remoto.naic.edu`. Note that an Arecibo Observatory user

Listing 3.2: `ionorun` script to coordinate PISCO receiver software.

```bash
#!/bin/bash

# Load the bash environment settings
source /home/radar/.cronenv

# Set a couple variables that are used for filenames
TIME=`date -d "next minute" +%F-%R`
DATA="/data/data_"$TIME".h5"
IMAGE="/data/rti_"$TIME".png"

# Run the IonosondeRxRun (the data collection program)
/usr/local/bin/IonosondeRxRun

# Rename the output file to a datestamped filename
mv /data/IonosondeRx.h5 $DATA

# Run the ionogram/RTI plotter
python /usr/local/bin/ionoplotter $DATA -m

# Rename image to a datestamped filename
mv /data/Ionogram.png $IMAGE

# Compress the data file
pbzip2 $DATA

# Create a symbolic link to the most recent ionogram/RTI image
unlink /data/rti_latest.png
ln -s $IMAGE /data/rti_latest.png

# Create a symbolic link to the most recent data file
unlink /data/data_latest.h5.bz2
ln -s $DATA".bz2" /data/data_latest.h5.bz2
```

Listing 3.3: Helpful ssh configuration entry for remote connection to the PISCO installation at Arecibo Observatory.

```
Host           coruscant
 User          radar
 Hostname      192.231.95.182
 ProxyCommand  ssh <USER>@remoto.naic.edu nc %h %p 2> /dev/null
```

account is required in order to make the connection.

A simple way to automate the connection through `remoto.naic.edu` is to use it as an `ssh` proxy. By adding the configuration in Listing 3.3 to the connecting host's `~/.ssh/config` file, connection to the remote machine can be made simply by running the command

```
$ ssh -Y coruscant
```

Data files and ionogram images are stored in `/data`. The ionogram plot from the most recent data run can be viewed using the command

```
$ eog /data/rti_latest.png
```

3.1.3 Summary

This section has presented the overall hardware and software architecture of the PISCO receiver system, in addition to many of the critical details that enable proper functionality of the system. Additional, more extensive code listings for the software powering PISCO can be found in Appendix A.

3.2 PSU All-sky Radar Interferometry System

The PSU All-sky Radar Interferometry System (PARIS) is a 50-MHz digital radar designed to study specular meteor trails. The first deployment of the system is located near The Pennsylvania State University (University Park) campus, at the Rock Springs Radio Space Observatory (magnetic mid-latitude). The system is a traditional radar in the sense that the transmitter and receivers are colocated and the system is fully synchronized by logic-level signals in hardware. COTS hardware components and open-source software comprise the radar system, providing transmit pulse control and conditioning, as well as receiver control; digital signal processing; and data collection, display, and storage. Ryan Seal is largely responsible for the system-level, hardware, and software designs [3], while the receive RF front-end design, system integration and documentation, preliminary data analysis software, and extension of the receiver software to support the five channels necessary for interferometry described by [36] were the primary duties of the author.

3.2.1 System Overview

Figure 3.10 shows a high-level systems diagram of PARIS, comprised of the transmit and receive segments. The transmit segment (outlined in gold) of this radar system is composed of five main high-level components: 1) the direct digital synthesizer responsible for generating the RF carrier signal, as well as several clock signals for other components; 2) the radar controller responsible for generating a variety of logic-level pulses necessary for radar operation, including the transmit pulse gate signal; 3) the transmit RF front end responsible for gating the RF carrier, as well as signal conditioning prior to transmission; 4) the transmitter responsible for power amplifying the pulsed RF carrier in preparation for transmission; and 5) the antenna responsible for transducing the electrical signal to RF energy. The pulsed RF signal propagates into the ionosphere from the transmit antenna, and reflects and scatters

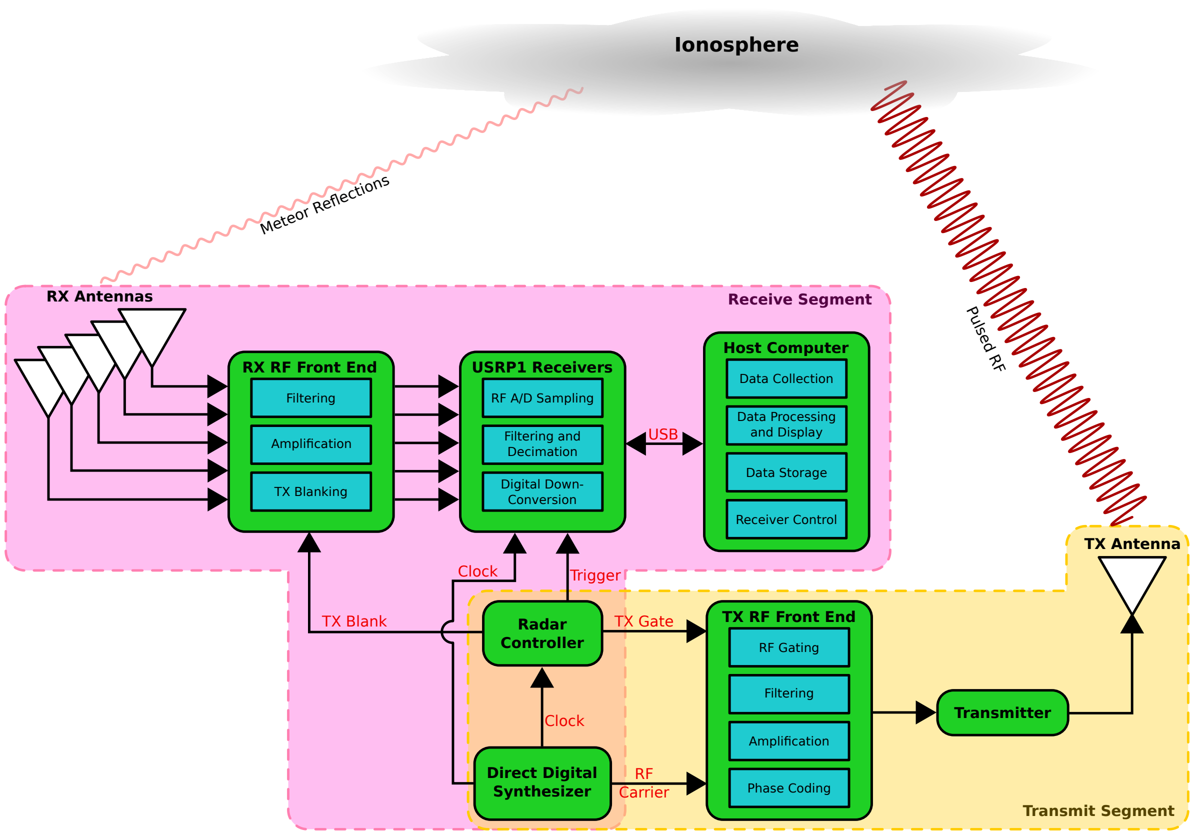


Figure 3.10: Overview of the PSU All-sky Radar Interferometry System.

off any available targets (e.g., meteor events, ionospheric layers, airplanes, etc.). Some of this reflected energy is captured by the receive antennas, the beginning of the signal path in the receive segment.

The six main components of the receive segment (outlined in pink in Figure 3.10) are: 1) the array of receive antennas responsible for transducing the captured RF energy from target reflections into electrical signals; 2) the receive RF front end responsible for some basic hardware signal conditioning; 3) the USRP1 receiver devices responsible for digitization and preliminary signal processing of the received signals; 4) the host computer responsible for controlling the USRP1 devices, as well as data collection, processing, display, and storage; 5) the radar controller responsible for generating logic-level signals for receiver triggering and protection; and 6) the direct digital synthesizer responsible for providing a stable clock source for the USRP1 devices. All of the components in both the transmit and receive segments are further detailed in Section 3.2.2.

3.2.2 Implementation

The following sections detail the hardware and software components from which PARIS is comprised.

3.2.2.1 Hardware

A systems diagram of the different hardware components and their interconnects is presented in Figure 3.11. These components are detailed in the following sections.

3.2.2.1.1 Antennas PARIS uses six five-element Yagi antennas: one for signal transmission, and five for interferometric signal reception. Each antenna is mounted on the ground vertically, illuminating the region of the sky directly above the antenna, with a beamwidth

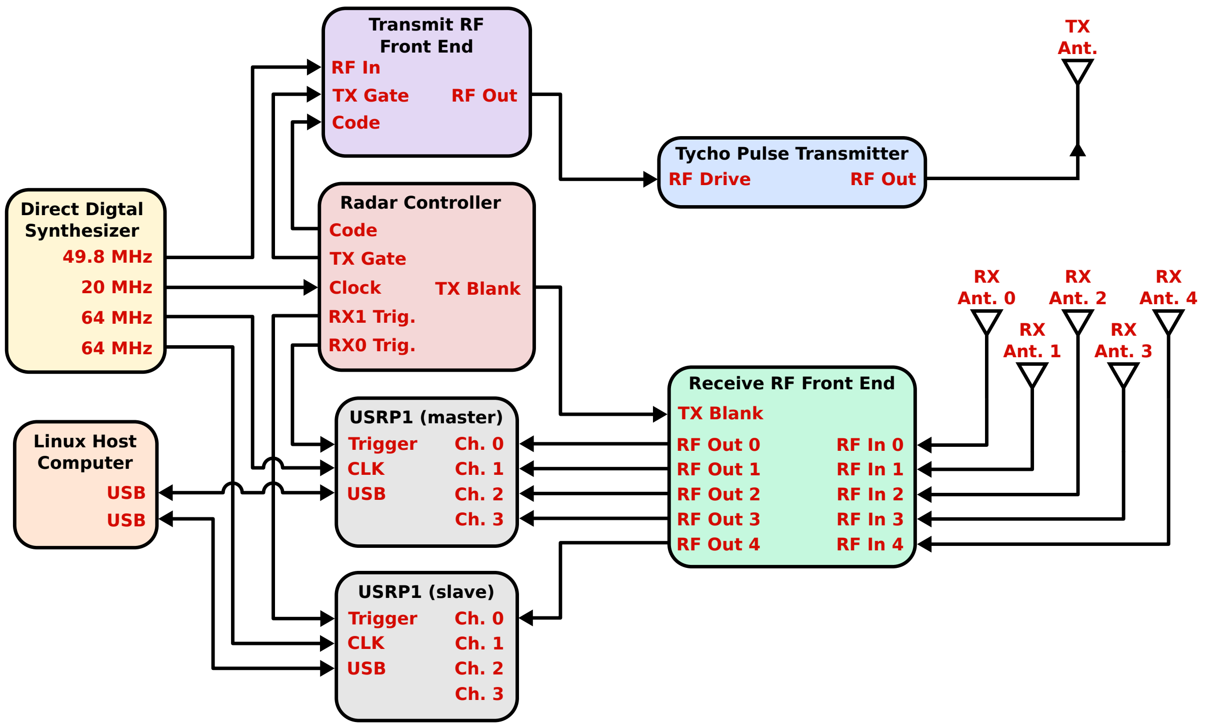


Figure 3.11: PARIS systems diagram (low-voltage DC and AC power connections omitted).

of approximately 60° [3].[5]

The physical layout of the five receive antennas is shown in Figure 3.12, based on the classical cross pattern for interferometry, as described in [36], with three antennas aligned along the magnetic North–South line and three aligned along the magnetic East–West, sharing the common central antenna. The antennas are oriented such that the Yagi elements are aligned to the magnetic East–West line (for consistency), and inter-antenna spacing is either 2λ or 2.5λ, where λ is the wavelength of the system RF carrier. The physical location of the transmit antenna with respect to the receive antenna array is unimportant as long as it is in the radiative far-field of the receive antennas [37].

Each of the five Yagi antennas used by the PARIS receive segment includes a preamplifier mounted directly on the antenna itself. Figure 3.13 shows the electrical configuration

[5]The descriptor "all-sky" is derived from the fact that the antennas used illuminate such a large region of the sky.

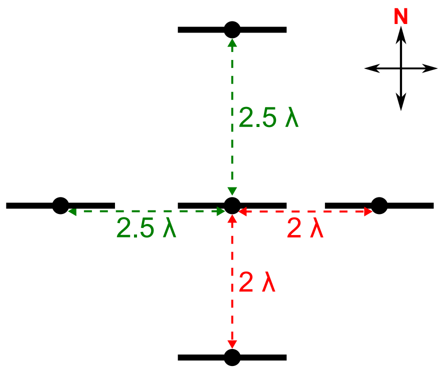


Figure 3.12: Top view of receive antenna array layout for PARIS, adapted from [36] (not to scale).

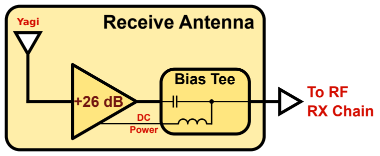


Figure 3.13: Hardware configuration of one receive antenna for PARIS.

of this preamplifier on the antenna, and Table 3.3 lists the individual components. The preamplifier's purpose is to amplify the received signal prior to sending it through the antenna transmission line, over which several decibels of attenuation will occur. Without this preamplifier, very low-amplitude signals could potentially drop below the system noise floor, at which point the signals are unrecoverable, regardless of receiver front-end hardware (discussed in Section 3.2.2.1.3). The preamplifier not only alleviates this issue, but also provides some preliminary filtering, as it is a tuned preamplifier, reducing out-of-band noise.

In order to power the preamplifiers on each of the antennas, a DC voltage is injected onto the antenna transmission line at the receive RF front end (see Section 3.2.2.1.3). This

Table 3.3: Receive antenna components.

Manufacturer	Part Number	Description
—	—	Yagi Antenna
Advanced Receiver Research	P50VDG	+26-dB RF Amplifier
MiniCircuits	ZFBT-282-1.5A+	Bias Tee

Table 3.4: Radar controller output signal descriptions, as used in PARIS.

Name	Description	Destination
TX Gate	Transmit pulse window	RF TX Chain
Code	BPSK phase code for pulse compression	RF TX Chain
TX Blanking	Receiver disable for protection from groundwave	RF RX Chain
RX0 Trigger	Receiver window	USRP1 (master)
RX1 Trigger	Receiver window	USRP1 (slave)

DC voltage is separated from the RF output of the preamplifier through the use of a bias tee. This device is functionally equivalent to a capacitor and inductor sharing a node, with high-frequency (i.e., RF) signals passing through the capacitor and low-frequency (i.e., DC) signals passing through the inductor.

3.2.2.1.2 Radar Controller Because the fundamental operation of a radar system is dependent on translating time intervals into physical distances, it is crucial for the different components of both the transmit and receive segments to operate synchronously and consistently. The radar controller (also called radar pulse generator) is the master device that generates the radar timing signals necessary for operating the rest of the system synchronously. The waveforms of these signals are defined by a number of radar parameters including, but not limited to, IPP, baud length, phase code, transmit pulse width, receiver window pulse width, and several delays. These digital signals are distributed throughout the radar system as illustrated in Figure 3.11.

The radar timing signals that are used in PARIS are listed in Table 3.4, and graphically illustrated in Figure 3.14. The TX Gate signal is used to limit the RF carrier (generated by the direct digital synthesizer) to just a small time window to transmit, defining the basic range-resolution of the radar (although this is improved through the use of pulse compression). The Code signal is the same length as the TX Gate, and controls the phase of the RF carrier (either $0°$ or $180°$ for BPSK). The TX Blanking pulse disables the receive RF front end for protection by using an active-low signal slightly wider than the TX Gate signal

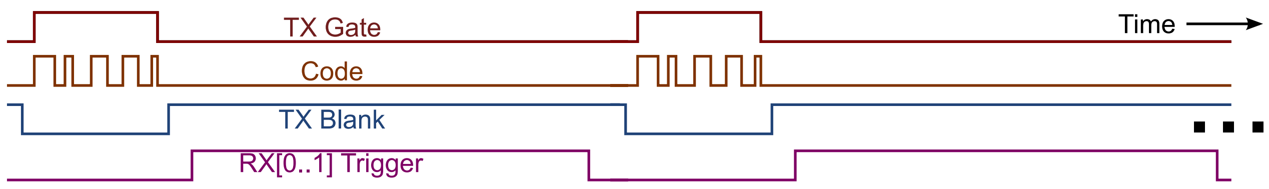


Figure 3.14: Radar controller output signals timing used by PARIS.

to account for transient effects from the transmitter outside the TX Gate window. Finally, two trigger signals (RX0 and RX1 Triggers) are used to enable sampling on the two USRP1 devices.

The radar controller used by PARIS uses an FPGA for accurate, stable, and reconfigurable radar pulse timing. Sixteen front-panel transistor–transistor logic (TTL) compatible, 50-Ω outputs are fully software programmable from the PicoITX form-factor computer inside the radar controller 2U rack-mount chassis. Additionally, an LCD screen on the front panel displays real-time information about the radar controller. The system can be synchronized to one of three different clock signals (one internal and two external), as well as several external timing triggers. Two open-source programs on the radar controller, `bpg-generate` and `bpg-shell`, are used to configure and operate the device[6] (discussed in Section 3.2.2.2.1).

3.2.2.1.3 Receive RF Front End After the antennas convert the captured RF energy into electrical signals, the receive RF front end provides important analog signal conditioning prior to digitization by the USRP1 receiver device. Figure 3.15 shows the configuration of one of the five signal chains in the receive RF front end, with each of the components listed in Table 3.5.

The receive RF front end begins with a bias tee at the input from the antenna cable in order to perform the DC power injection for biasing the antenna preamplifiers, as discussed in Section 3.2.2.1.1. The output of the bias tee is followed immediately by a diode-based

[6]See [39] for more information about both the hardware and software of the radar controller.

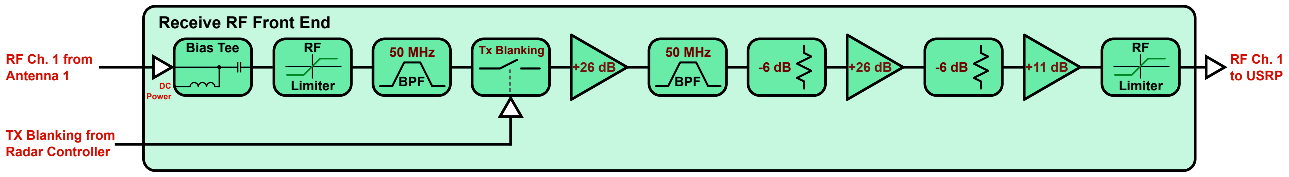


Figure 3.15: One channel of the PARIS receive RF front end.

Table 3.5: Receive RF front-end components.

Manufacturer	Part Number	Description
MiniCircuits	ZFBT-282-1.5A+	Bias Tee
Ryan Seal	—	RF Limiter
KR Electronics	KR-2867	Band-pass Filter
MiniCircuits	ZYSW-2-50DR	RF Gating Switch
Advanced Receiver Research	P50VDG	+26-dB RF Amplifier
KR Electronics	KR-2867	Band-pass Filter
MiniCircuits	VAT-6+	6-dB Attenuator
Advanced Receiver Research	P50VDG	+26-dB RF Amplifier
MiniCircuits	VAT-6+	6-dB Attenuator
Advanced Receiver Research	P30-1000/11VD	+11-dB RF Amplifier
MiniCircuits	VLM-33+	RF Limiter

RF limiter circuit, designed to clip the input signal if it exceeds one diode threshold voltage, protecting components further down the chain from input overvoltage conditions. Following the RF limiter is a 50-MHz band-pass filter, with a bandwidth of approximately 9 MHz and very steep passband rolloff edges, effectively removing any out-of-band signals that may have been picked up between the antenna and this filter. This device also serves as the important anti-aliasing filter, enabling the USRP1 receiver to use digital down-conversion to demodulate the carrier, without introducing out-of-band information. Next is the RF gating switch used for blanking the receiver during transmitter operation, protecting the components further down the chain from input overvoltage. The logic-level signal that controls operation of this device is the TX Blanking signal from the radar controller (see Section 3.2.2.1.2). Next is the same low-noise, tuned preamplifier that is used on the antenna, providing a gain of approximately +26 dB. Another 50-MHz band-pass filter follows the amplifier, further reducing out-of-band noise. A 6-dB attenuator is then used for two purposes: first, to reduce the signal to a level within the linear range of the amplifiers further down the chain (i.e., avoiding output clipping); and second, to reduce all frequency content equally, potentially pushing some out-of-band noise below the system noise floor. Another tuned preamplifier and attenuator are then used for the same reasons previously discussed, increasing the desired signal level and reducing noise level. Next, a wideband +11-dB amplifier is used, further increasing signal level. Finally, just before the receive RF front-end output, an RF limiter is used to clip any large input signals to the USRP1's maximum input range (2 V_{pp} or +10 dBm), avoiding saturation and damage to the USRP1 device.

Utilizing the Friis formula for cascaded noise figure analysis [38], a basic MATLAB script was written to calculate noise figure and overall system gain for RF front-end signal chains (code listed in Appendix A). For PARIS's receive RF front end, the calculated system noise figure is 0.6 dB with a gain of 59.3 dB, including the USRP1's ADC (see Section 2.2.1.2), but excluding the antenna (as the antenna's noise temperature/noise figure are unknown).

3.2.2.1.4 USRP Like the PISCO system (discussed in Section 3.1), PARIS also uses the USRP1 device as the central unit in the receiver. However, because each USRP1 device is limited to a maximum of four receive channels, two devices are needed for the five-channel setup of PARIS. The "master" device is fixed with two BasicRX daughter boards (in the USRP's RXA and RXB slots), each providing two channels to connect the output of a receive RF front end. The "slave" device is fitted with just a single BasicRX daughterboard in the RXA slot, providing the fifth channel needed for interferometry. The daughterboards provide 50-Ω termination and transformer coupling of the input signal to the USRP's ADCs. Also like the PISCO system, the USRP1 devices of PARIS use a sampling clock of 64 MHz from the external direct digital synthesizer (Section 3.2.2.1.7). A discussion of the sampling frequency, system bandwidth, and the USRP1 internal oscillator issues is presented in Section

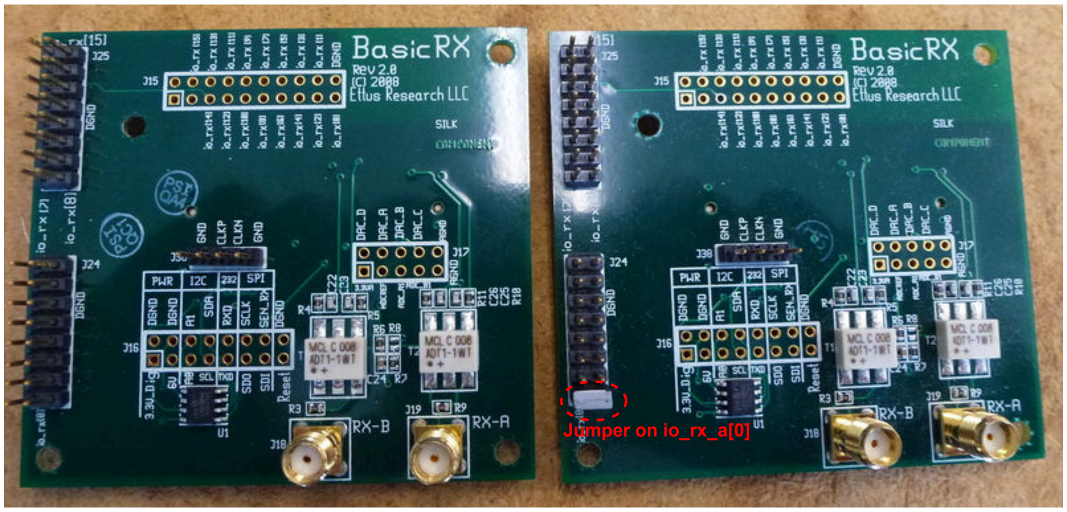


Figure 3.16: Grounding jumper positions on the BasicRX daughterboard for the master (left) and slave (right) USRP configurations used by PARIS.

3.1.2.1.2.

The master and slave USRP1 devices are configured as such through the use of the external general-purpose input/output (GPIO) lines from the FPGA. In addition to providing two signal inputs, the BasicRX daughterboard also features several break-out headers that give access to sixteen digital GPIO connections to the FPGA. The `usrp_trigger.rbf` FPGA image used by PARIS is configured to pull up the first fifteen GPIO lines on the BasicRX daughter board in the RXA slot (`io_rx_a[0:14]`). The use of grounding jumpers allows these first fifteen GPIO lines to form a binary address[7] used by the host software to differentiate between USRP devices and allow them to be correctly ordered in software. Because the GPIO lines are pulled high and grounding jumpers are used, the address is actually counted backwards from 0x7FFF. This means the master device uses no grounding jumpers on `io_rx_a[0:14]` with the address 0x7FFF, and the slave device uses a grounding jumper on `io_rx_a[0]` with the address 0x7FFE. Figure 3.16 shows the BasicRX daughterboards in both the master and slave configurations.

[7]This addressing technique is actually expandable to $2^{15} = 32768$ USRP1 devices, even though only two are used in PARIS.

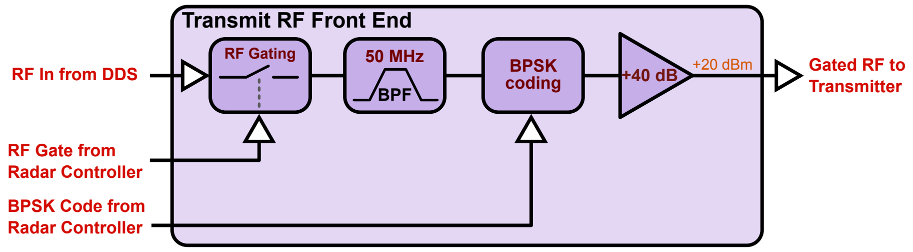


Figure 3.17: PARIS transmit RF front end.

Table 3.6: Transmit RF front-end components.

Manufacturer	Part Number	Description
MiniCircuits	ZYSW-2-50DR	RF Gating Switch
MiniCircuits	SIF-50+	Band-pass Filter
Julio Urbina	—	Phase-Coding Circuit
MiniCircuits	ZHL-5W-1	+40-dB RF Amplifier

The sixteenth GPIO signal on the RXA BasicRX daughterboard (`io_rx_a[15]`) is used as the trigger signal input from the radar controller that enables sampling on the USRP during the receive trigger window (see Section 3.2.2.1.2). A custom-built 50-Ω terminated cable is used to connect between the USRP's chassis and the GPIO and ground pins.

The triggering implementation in `usrp_trigger.rbf` on the FPGA is discussed further in Section 3.2.2.2.2, but generally the structure is similar to the standard `usrp1_fpga.rbf` design provided by the UHD software package. A programmable NCO and complex multiplier produce a complex data stream from the USRP's ADCs, and also provides CIC decimation and filtering of the signal. The USRP1 device transfers a channel-interleaved data stream to the host computer over a USB 2.0 interface.

3.2.2.1.5 Transmit RF Front End In order to generate the pulsed RF signal to be amplified and sent to the transmit antenna by the transmitter, several stages of analog signal conditioning need to be applied to the RF carrier signal from the DDS. The transmit RF front end consists of four stages, as illustrated in Figure 3.17 and listed in Table 3.6. The first component in the chain is an RF gating switch, limiting the RF carrier to the width of

Table 3.7: Basic hardware specifications of the PARIS receiver GPC.

Processor Family	Intel Core-i7 960, 3.2 GHz
RAM	12 GB DDR3 (3×4 GB)
Hard Drive	1 TB, 7200 RPM SATA
USB Interfaces	8×USB 2.0

the TX Gate signal from the radar controller, as described in Section 3.2.2.1.2. The output of the RF gating switch is connected to a 50-MHz band-pass filter in order to reduce the high-frequency harmonic content that appears on the gated RF carrier due to the RF gating switch itself. The next component in the transmit RF front end is the phase-coding circuit. Given the logic-level Code signal generated by the radar controller, it applies a phase shift of $0°$ or $180°$ to the input signal, BPSK-encoding the pulsed RF used for pulse compression radar operation. Finally, the signal is amplified with a high-gain (+40-dB) amplifier in order to bring the signal level up to the 6-V_{pp} (+20-dBm) level required by the RF input of the transmitter.

3.2.2.1.6 General-Purpose Computer The host general-purpose computer used by PARIS is responsible for data capture and processing from the two USRP1 devices. Most modern desktop or laptop computers with at least two USB 2.0 interfaces should be capable of acting as the GPC for PARIS.[8] Table 3.7 lists the basic hardware specifications of the GPC tested and used with PARIS.

3.2.2.1.7 Direct Digital Synthesizer In order to properly synchronize the different components of the radar system, a single master clock reference from which all other clock signals are derived is necessary. The Novatech DDS9m is a 170-MHz, four-channel programmable oscillator that synthesizes the output waveforms from either the internal Voltage-Controlled Temperature-Compensated Crystal Oscillator (VCTCXO) or an external clock

[8]The sum total USB bandwidth on the GPC is a possible limiting factor of channel bandwidth on the USRP1 devices, although this has not been fully investigated.

Table 3.8: DDS output signals.

Channel	Frequency (MHz)	Amplitude (Counts)	Destination
0	20.000000	1023	Radar Controller Std. Clock
1	49.796875	60	RF Input to Transmit RF Front End
2	64.000000	1023	USRP1 (slave) clock
3	64.000000	1023	USRP1 (master) clock

source [40]. Each of the four independent outputs can be programmed with frequency, amplitude, and phase over a serial interface.

Table 3.8 lists the output configuration of the Novatech DDS9m as used in PARIS. Note that the phase of all four channels is set to 0°. The radar controller uses a 20-MHz clock for pulse generation, while the USRP1 devices each use a clock of 64 MHz, all with maximum amplitude count, as the exact amplitude of these signals is not critical for proper operation. The RF carrier's amplitude, however, is calibrated such that the output of the transmit RF front end meets the 6-V_{pp} (+20-dBm) input level required by the transmitter.

This device is nearly identical in performance to the Novatech 409B used by PISCO (see Section 3.1.2.1.3); however, it is sold at the printed circuit board (PCB) level, without its own chassis, thus requiring a custom housing.

3.2.2.1.8 Transmitter In order for the receiver to detect the weak signal returns from meteor events and other ionospheric phenomena, the pulsed RF signal to be transmitted must be amplified significantly before it is applied to the transmit antenna. The transmitter is designed specifically for this purpose, while also providing control and safety mechanisms necessary for high-power electronics.

PARIS uses the Tycho Technologies WPT-50 Pulse Transmitter (shown in Figure 3.18), a 50-MHz (tunable) pulse transmitter developed for wind profiling radar applications. The transmitter uses three amplifier stages in the RF signal path: a solid state amplifier followed by two vacuum tube amplifier stages (driver and final) [41]. The transmitter is controllable

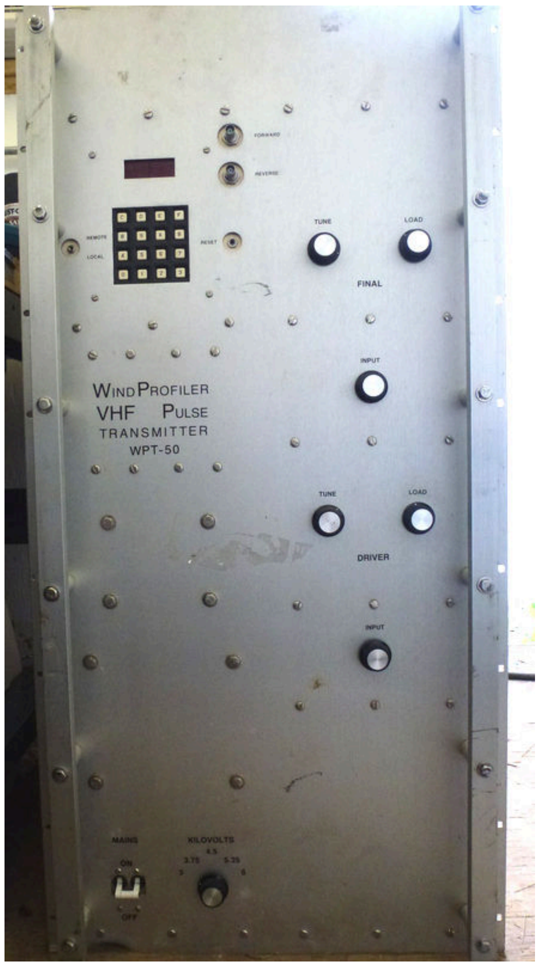


Figure 3.18: Front view of the Tycho Technologies WPT-50 Pulse Transmitter.

via a number-pad and seven-segment display on the front panel, as well as through a remote serial interface. A number of analog sensors monitor the status of various parameters of the transmitter (voltages, currents, temperatures, etc.), and an on-board microcontroller uses these sensor measurements to determine if the transmitter is operating within expected bounds. If so, the user is permitted to advance the transmitter through states 1 (warm-up period), 3 (high-voltage biasing of the tubes), and 4 (input signal applied and transmission). If any sensor value is out of range, the transmitter disables the amplifiers for safety and displays an error code. Additionally, six standard and two high-voltage fuses are used to protect both the internal electronics and operator in the case of a failed component or unsafe external operating condition.

The transmitter requires a pulsed RF input signal, with amplitude of approximately 6 V_{pp} (+20 dBm) and a duty-cycle of no greater than 2%, to be applied to the "RF Input" port (BNC-type) on the rear of the transmitter [41]. The transmit antenna feedline is connected to the "RF Output" port (HN-type) also on the rear of the transmitter. Several signal "sense" ports are available to monitor the pulsed RF signal at different stages.

Experience operating the transmitter has shown high sensitivity to the final stage filament voltage sensor measurement, which is an AC voltage derived from the 220–250 V_{AC} mains power input. Because the mains voltage varies daily and over the course of the year, it is sometimes necessary to adjust the filament voltage sensor potentiometer, such that the filament voltage measurement stays within the 9.8- to 10.7-V operational range.

3.2.2.2 Software

Being a software-defined radar system, a large portion of the PARIS's flexibility is in the software. The following sections provide a high-level overview of the software run by the radar controller and host GPC; however, the reader is encouraged to see [3, 4, 39, 42, 43, 44] for further details.

3.2.2.2.1 Radar Controller As previously mentioned, the radar controller contains a PicoITX computer (running Gentoo Linux) for both radar controller configuration and operation. The software that governs the radar controller operation is part of a project called Bit Pattern Generator (BPG), designed for a general class of digital waveform generator systems [43, 44]. Operating the radar controller is typically a three-step process, illustrated in Figure 3.19. The user begins by writing a plain-text configuration file (or modifying an existing configuration), called a human-interpretable file (HIF), that defines the behavior of the Bit Pattern Generator device. The user then calls the `bpg-generate` program to translate the HIF into an instrument-interpretable file (IIF) for the radar controller hardware. Then, the user runs `bpg-shell`, an interactive console-type interface that allows the user to control the radar controller's hardware (see Section 3.2.2.1.2 for more details), during the course of which the radar experiment takes place.

3.2.2.2.1.1 bpg-generate After the user or experiment designer has defined a mode in the HIF, the defined parameters and signal outputs must be translated into a format understood by the radar controller hardware, namely the FPGA. The `bpg-generate` program is responsible for this task, provided the syntax of the HIF is correct. Additionally, the configuration parameters are validated against a predefined list of rules specific to the hardware of radar setup. For instance, when operating with the Tycho WPT-50 transmitter, the program will throw an error if a transmit pulse duty cycle of >2% is detected. Finally, if the configuration passes the ruleset, an IIF is generated.

The command to generate an IIF from an HIF is as follows:

```
$ bpg-generate -o mymode.iif mymode.hif
```

where `mymode.hif` is the plain-text HIF defined by the user and `mymode.iif` is the output IIF generated by `bpg-generate`. Any number of modes can be generated and saved for immediate or future use.

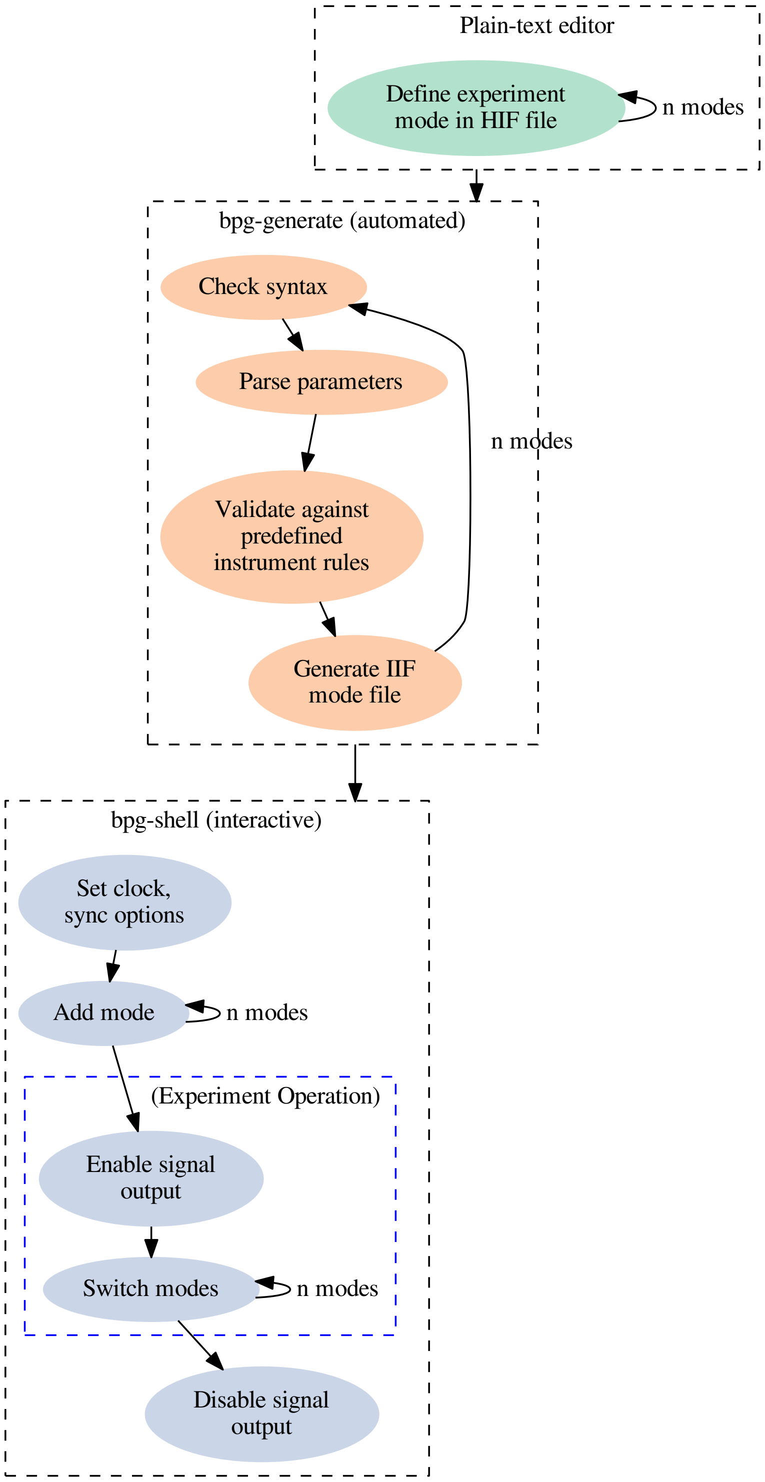


Figure 3.19: Typical operational workflow with the Bit Pattern Generator software.

3.2.2.2.1.2 bpg-shell After generating one or more IIF configuration modes, the `bpg-shell` program may be used to control the radar controller hardware outputs. `bpg-shell` accepts around fifteen different commands; however, during normal operation of systems like PARIS, only a few are used. Because the FPGA can connected to multiple clock sources, the user must specify which clock source to use (internal, standard external, or drifted external), with the `clock` command. Additionally, a synchronization source (e.g., 1PPS) can be selected if desired. Next, the user can add any number of previously defined IIF modes to the shell's mode list, via the `add` command. After at least one mode is added, the radar controller's signal outputs can be enabled with the `start` command (additional modes can be added or removed (`remove`) from the mode list after the output has been started if necessary). At this point, radar experiment operations may begin, assuming all other hardware is properly configured and operating, of course. While the output is enabled, the user can synchronously switch (`switch`) between operating modes, traversing the mode list linearly. At the conclusion of the radar experiment, the signal outputs should be disabled with the `stop` command.

The command to start the interactive shell programs is as follows:

```
$ bpg-shell
```

`bpg-shell` is typically left running for the duration of the experiment. It is important to note that although the FPGA is configured by and receives commands from the radar controller computer, it operates independently. This means it is possible for the radar controller to be outputting signals even after the user has exited `bpg-shell`. Upon running `bpg-shell` again, the FPGA will be reset and all signal outputs will cease.

3.2.2.2.2 USRP FPGA The FPGA design used by PARIS (and in general, the Gnu-Radar software package) is `usrp_trigger.rbf`, modeled after the standard FPGA design provided by UHD, but with special modifications for triggering capabilities. Figure 3.20

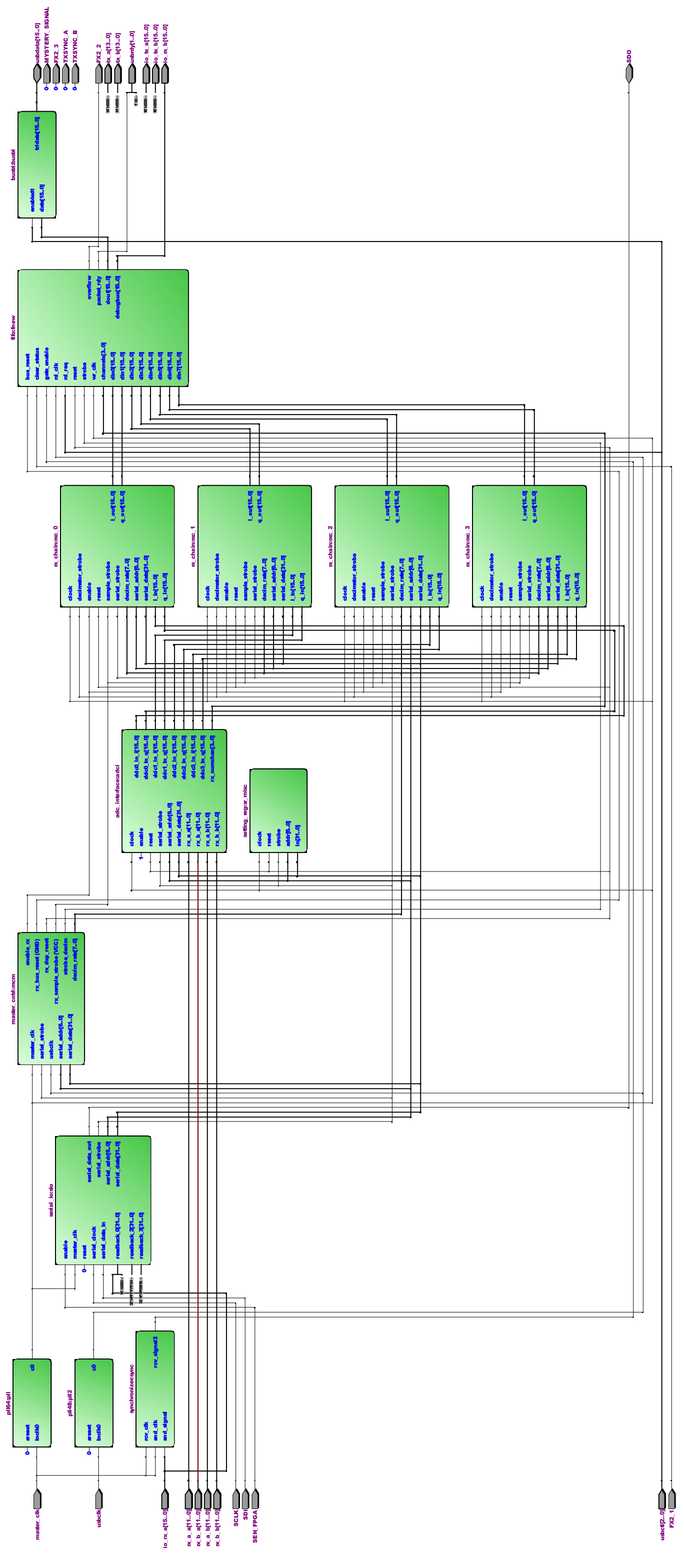


Figure 3.20: Toplevel schematic view of the `usrp_trigger` FPGA design used by PARIS/GnuRadar.

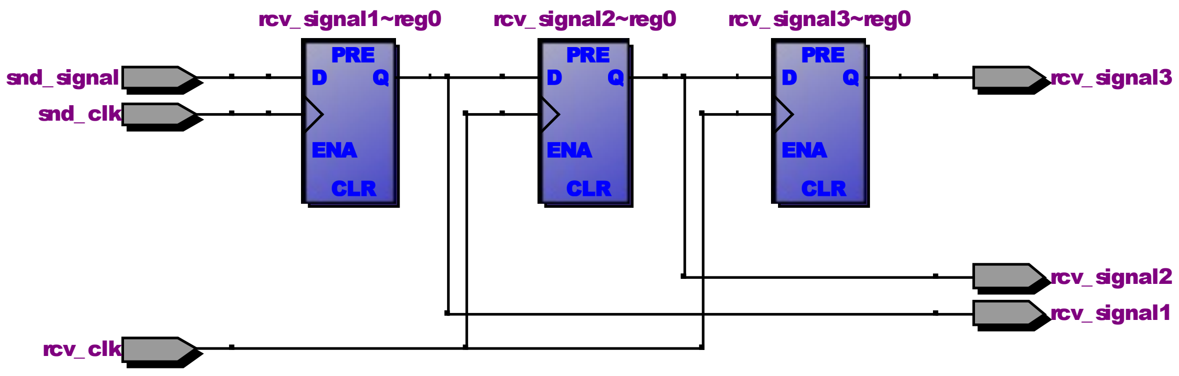


Figure 3.21: Schematic view of the `synchronize` module used for triggering the `usrp_trigger` FPGA design in GnuRadar.

shows a schematic representation of the top-level module of the `usrp_trigger` design. The main signal path begins with the `rx_a_a[15:0]`, `rx_a_b[15:0]`, `rx_b_a[15:0]`, and `rx_b_b[15:0]` signals that are received by the `adc_interface` module. The signals are then distributed out to the four `rx_chain` modules, which provide logic for I/Q generation, frequency tuning via the internal NCO, signal decimation, and filtering. The output signals from the `rx_chain` modules are sent to a first-in first-out (FIFO) memory with some additional logic to interleave the I and Q signals from each of the four channels. Finally, the output signal is sent to a tri-state buffer module, responsible for interfacing with the USB microcontroller on the USRP. Additionally, there are several control modules that handle generating control signals and managing settings registers, as well as clock modules for managing the FPGA input clock signals.

The `synchronize` module was added to the top-level module for the triggering functionality, enabling sampling only during the logic-high receive window (signal `io_rx_a[15]`) from the radar controller. A schematic representation of the `synchronize` module logic is shown in Figure 3.21. Several D-type flip-flops are used to buffer the trigger signal and then apply it to the `fifo` module, where the processed signals are synchronized.

In addition to interleaving the I and Q signals from the output of the four `rx_chain`

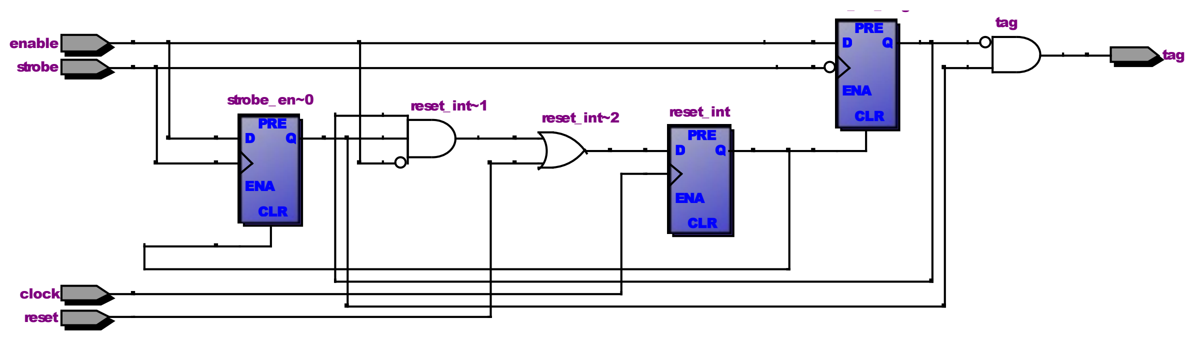


Figure 3.22: Schematic view of the `data_tag` module in GnuRadar.

modules, the `fifo` module also includes the logic for injecting the data tag into the each of the processed data streams, added specifically for GnuRadar. Figure 3.22 shows a schematic representation of the logic for generating the tagging control signal, implemented in hardware as several D-type flip-flops and a few logic gates.

3.2.2.2.3 Host Computer As illustrated in Figure 3.10, the host GPC is responsible for several tasks including data collection, storage, processing, and display, as well as receiver control. The following sections describe the different software pieces of GnuRadar involved in data collection and display, as well as receiver control. The received data is stored in tabular format within HDF5 files, in the same way the PISCO data files are stored (as PISCO also uses the GnuRadar software package), so the reader is referred to Section 3.1.2.2.3 for further details. Figure 3.23 shows the typical operational workflow of the different GnuRadar software components, discussed in the following sections.

3.2.2.2.3.1 Configuration Being a software-defined radar software package, many different radar configuration options are available to allow the user to tailor the system to their needs. The GnuRadar configuration file is used to collect many of these parameters into a single location for two purposes. First, some of the parameters are important to configure the USRP1 receiver device during the radar experiment (e.g., IPP length, sampling rate,

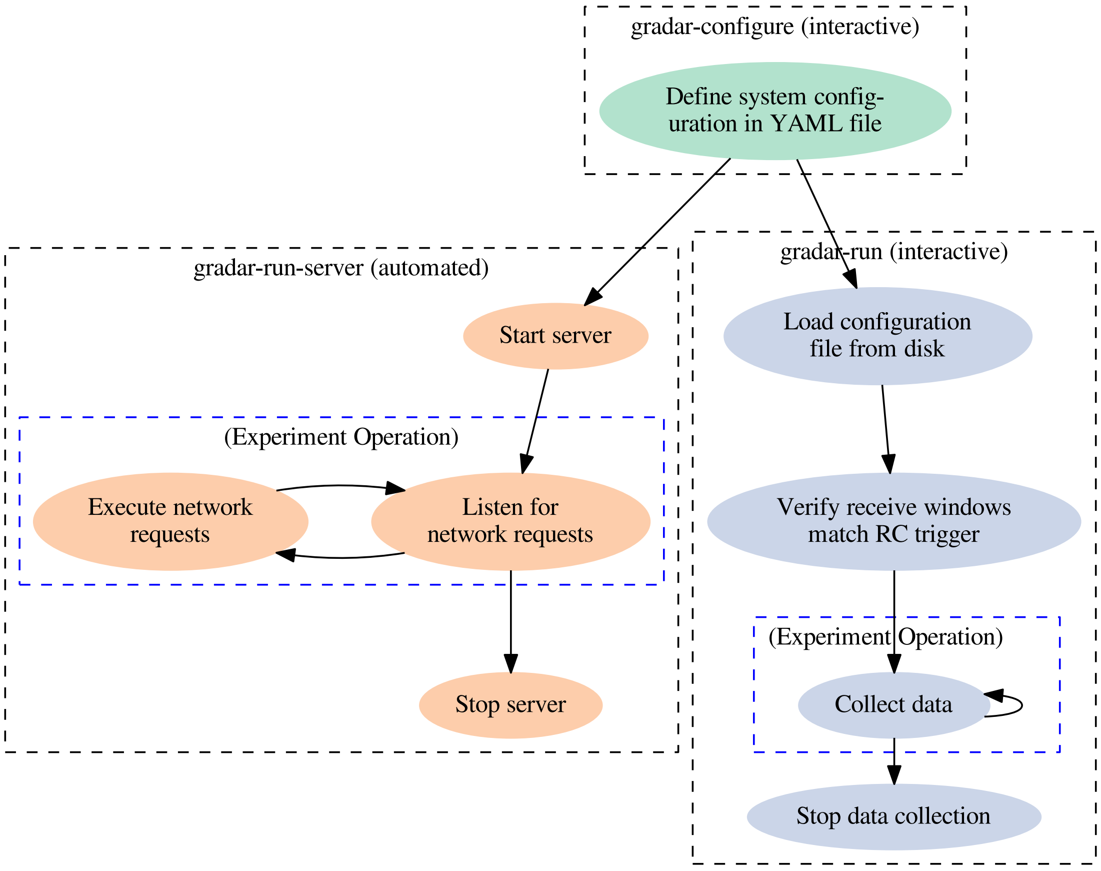
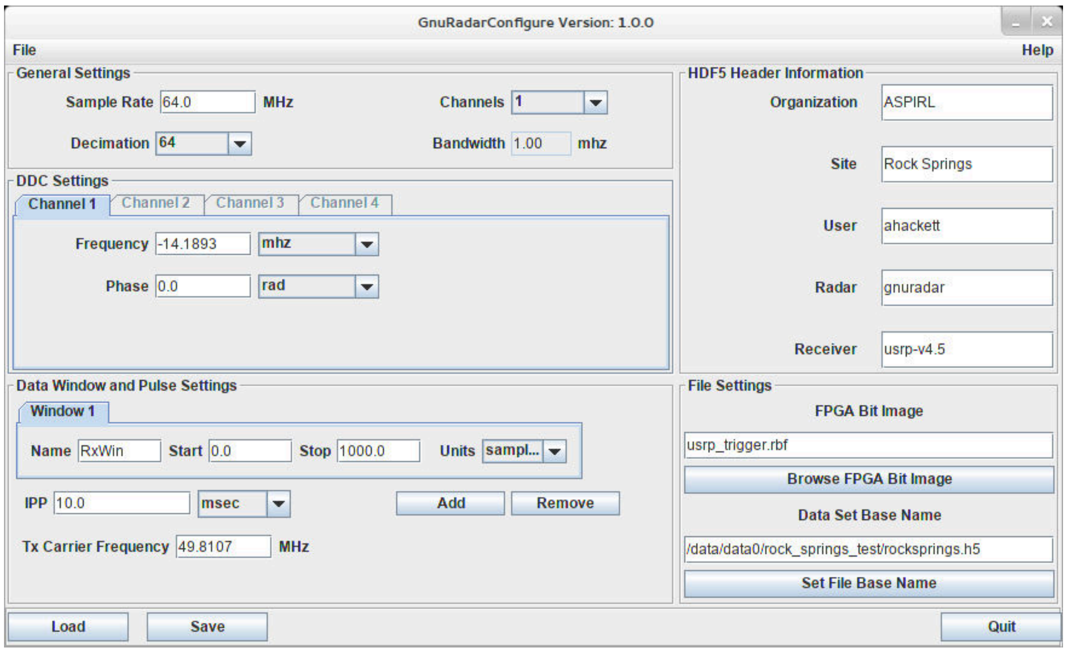

gradar-configure (interactive)
Define system config-uration in YAML file
gradar-run-server (automated)
Start server
(Experiment Operation)
Execute network requests
Listen for network requests
Stop server
gradar-run (interactive)
Load configuration file from disk
Verify receive windows match RC trigger
(Experiment Operation)
Collect data
Stop data collection

Figure 3.23: Typical operational workflow with GnuRadar.



Sample Rate 64.0 MHz
Channels 1
Decimation 64
Bandwidth 1.00 mhz
Frequency -14.1893 mhz
Phase 0.0 rad
Name RxWin Start 0.0 Stop 1000.0 Units sampl...
IPP 10.0 msec
Tx Carrier Frequency 49.8107 MHz
Organization ASPIRL
Site Rock Springs
User ahackett
Radar gnuradar
Receiver usrp-v4.5
FPGA Bit Image usrp_trigger.rbf
Data Set Base Name /data/data0/rock_springs_test/rocksprings.h5

Figure 3.24: Screenshot of the `gradar-configure` utility.

number of channels, etc.). Second, some of the parameters are important to describe the experiment to data users (e.g., radar name, receiver device, RF carrier frequency, etc.) for both processing and results analysis. All of these parameters (as well as several others) are stored in a YAML file, which is parsed by GnuRadar during runtime.

The GnuRadar software package provides a GUI for simplifying the creation and modification of configuration files. A screenshot of the interface is shown Figure 3.24. The use of dropdown menus and dynamically added channel window options not only speeds up the process of configuration file creation, but it also helps prevent invalid configuration options from ending up in the configuration file.

3.2.2.2.3.2 Data Capture The `gradar-run-server` program is a network-based listener that waits to receive control command packets from `gradar-run` (or any other program capable of sending the control packets). These commands and brief descriptions are listed below; however, the reader is referred to [4, 42] for a more detailed description of their implementation and operation.

- *Verify* – Verifies that the receive window defined in the GnuRadar configuration file matches the receive window detected by the USRP
- *Start* – Initiates data collection from the USRP device
- *Stop* – Terminates data collection and closes out the data file

Upon receipt of any of the commands, the `gradar-run-server` back end executes the command while the network interface continues to listen for the next command packet. Figure 3.25 illustrates the fundamental software architecture of the `gradar-run-server` back end, the backbone of which is the `ProducerConsumerModel`. This is a multi-threaded object class that manages the `ProducerThread` and `ConsumerThread` object classes, which generate and store data, respectively. `ProducerThread` is a parent class of `Device`, which provides a generalized interface for data source hardware. `Device` itself is a parent of the object class `GnuRadarDevice` (not shown in Figure 3.25), which provides a specialized interface for data collection from the USRP1, using the low-level application programming interface (API) formerly provided by GnuRadio. It is within `GnuRadarDevice` that most of the modifications for multi-device operation were made.

Before discussing the implementation details, it is important to note that the design philosophy taken when modifying GnuRadar for multi-device operation, the main goal was to minimize overall code impact, taking advantage of the abstraction layers provided by the object-oriented design. By taking this approach, coding efforts were minimized, reducing time to market (so to speak).

Two major components of `GnuRadarDevice` were modified for multi-device operation, the first of which being the constructor (`GnuRadarDevice::GnuRadarDevice()`). Figure 3.26 illustrates the constructor operation process, both before and after mutli-device modifications were made. Instead of creating only one USRP device object, the constructor now creates

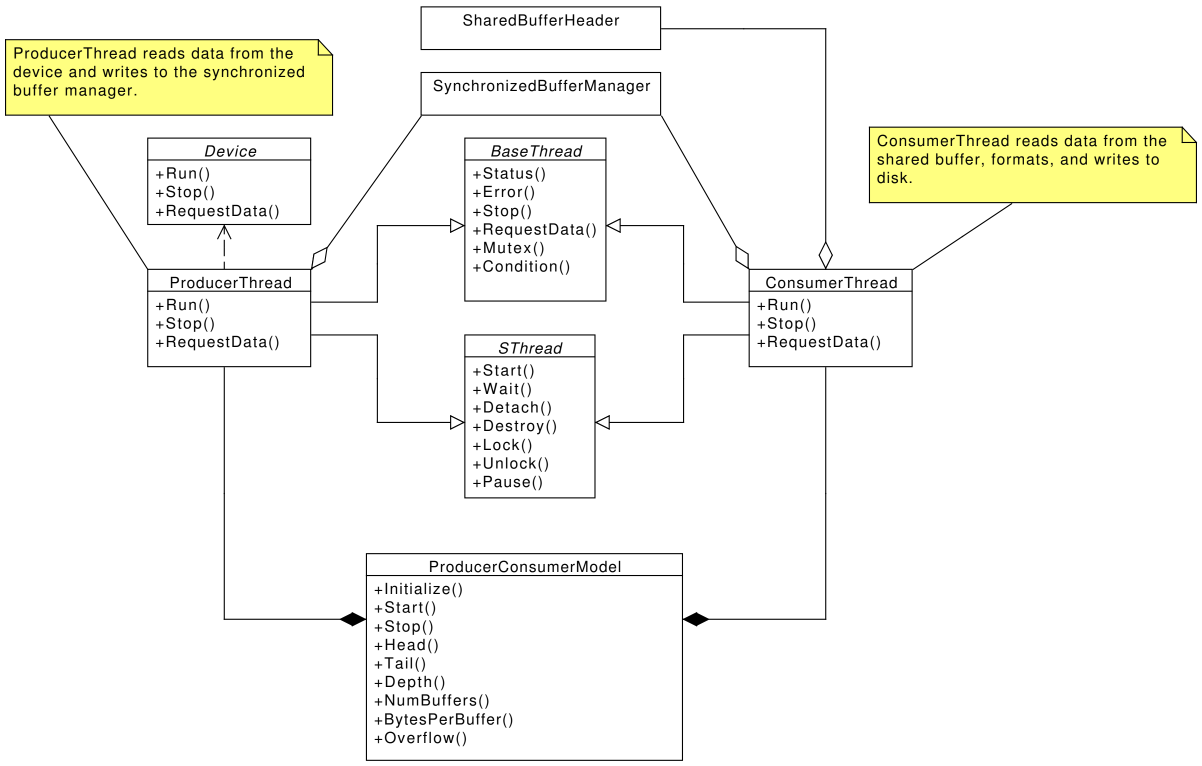
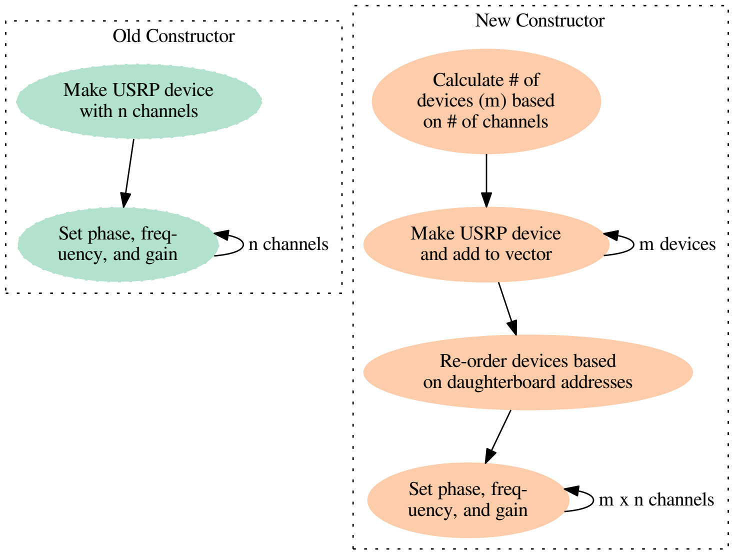


Figure 3.25: Class diagram of `gradar-run-server` backend [42].



Old Constructor
Make USRP device with n channels
Set phase, frequency, and gain
n channels
New Constructor
Calculate # of devices (m) based on # of channels
Make USRP device and add to vector
m devices
Re-order devices based on daughterboard addresses
Set phase, frequency, and gain
m x n channels

Figure 3.26: `GnuRadarDevice` constructor before (left) and after (right) multi-device modifications.

multiple devices, based on the number of channels set in the GnuRadar configuration file, and adds them to a standard template library (STL) vector. Because of the uncertainty in USB device ordering by the operating system, the USRP device vector must be sorted based on the hardware addressing scheme described in Section 3.2.2.1.4. Finally, the frequency, phase, and gain for each channel is set; however, some code accommodations were made to select the correct device for each channel.

The second component of `GnuRadarDevice` that was modified for multi-device operation is the `GnuRadarDevice::RequestData()` member function, inherited from the `ProducerThread` object class (through `Device`). As illustrated in Figure 3.27, instead of reading data from the USRP device into a single buffer, an STL vector of buffers is created to store a buffer for each device. At this point, the buffers are sample-interleaved in the same fashion that the USRP channel data streams are interleaved (I0 Q0 I1 Q1 I2 Q2 I3 Q3 ...), and then placed into shared memory, where `ConsumerThread` can store the data to disk. This approach takes advantage of the abstraction of the `GnuRadarDevice` class, hiding the number of devices from the rest of the system, and eliminating the need for higher-level architecture modifications.

Many of the finer details have been omitted, but the reader is encouraged to reference [3, 4, 42] for a more thorough understanding.

3.2.2.2.3.3 Receiver Control As described in Section 3.2.2.2.3.2, a network-based command system is used for controlling data collection with GnuRadar. While `gradar-run-server` is a network listener, `gradar-run` generates and sends the command packets over the network to the data collection GPC. Figure 3.28 shows a screenshot of the `gradar-run` program during data collection (after the configuration has been loaded and verified).

In the current PARIS implementation, both `gradar-run-server` and `gradar-run` are

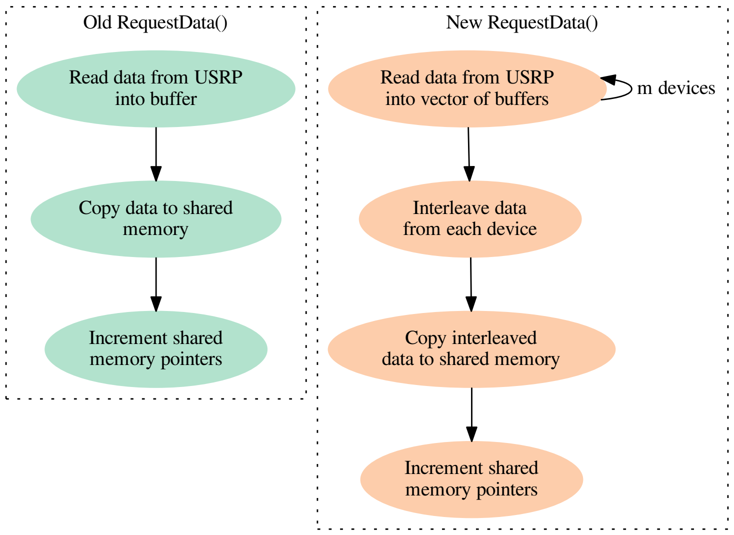
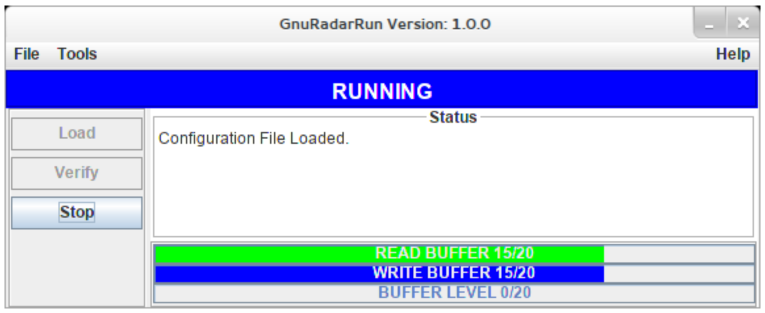

Old RequestData()
Read data from USRP into buffer
Copy data to shared memory
Increment shared memory pointers
New RequestData()
Read data from USRP into vector of buffers
m devices
Interleave data from each device
Copy interleaved data to shared memory
Increment shared memory pointers

Figure 3.27: `GnuRadarDevice::RequestData()` before (left) and after (right) multi-device modifications.



Configuration File Loaded.

Figure 3.28: Screenshot of the `gradar-run` program.

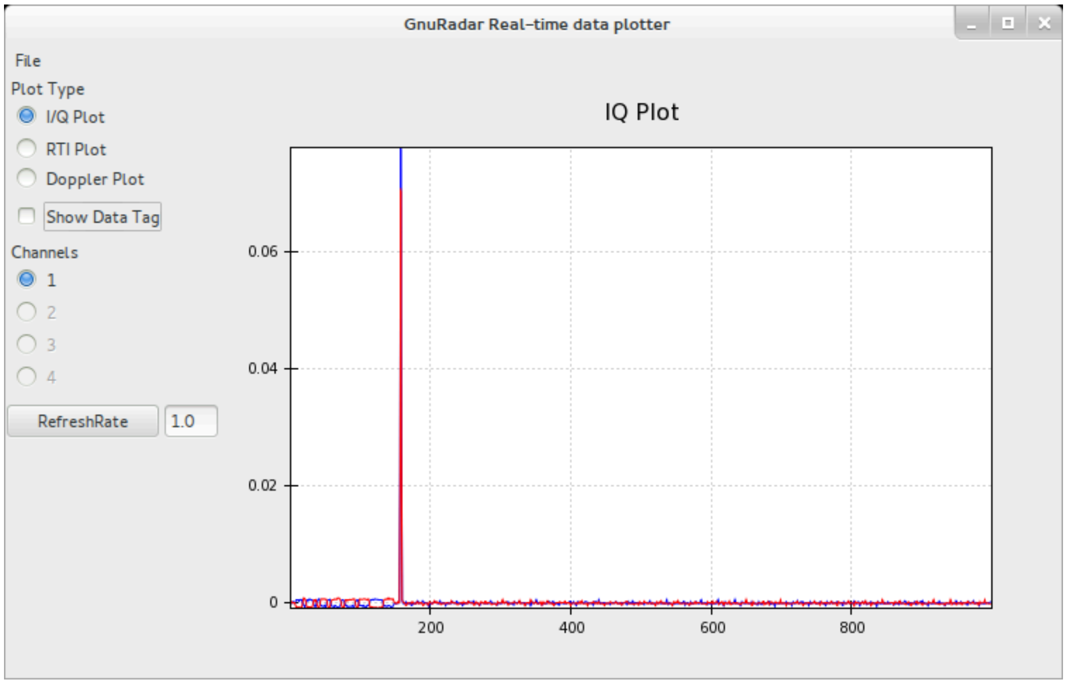


Figure 3.29: Screenshot of the `gradar-plot` program.

executed on the same GPC, although the system was designed to support separate control and server machines if desired.

3.2.2.2.3.4 Data Display GnuRadar also features a basic realtime data plotter, `gradar-plot`, for preliminary data display prior to post-processing routines. The plotter grabs the raw data from the buffers written to host GPC memory by `gradar-run-server`, prior to writing to the data file, and makes it available for display. Both oscilloscope-type (I/Q Plot) and range-time-intensity (RTI) plots are currently supported, and Figure 3.29 shows a screenshot of `gradar-plot` running in the I/Q Plot view.

3.2.2.2.3.5 Data Processing A preliminary data processing program, `rti_big.py`, was developed for batch offline processing and plotting data files taken by the PARIS system. Two different plotting modes are available: a single image showing data for entire specified range, or multiple images, each showing a regular time interval within the specified range.

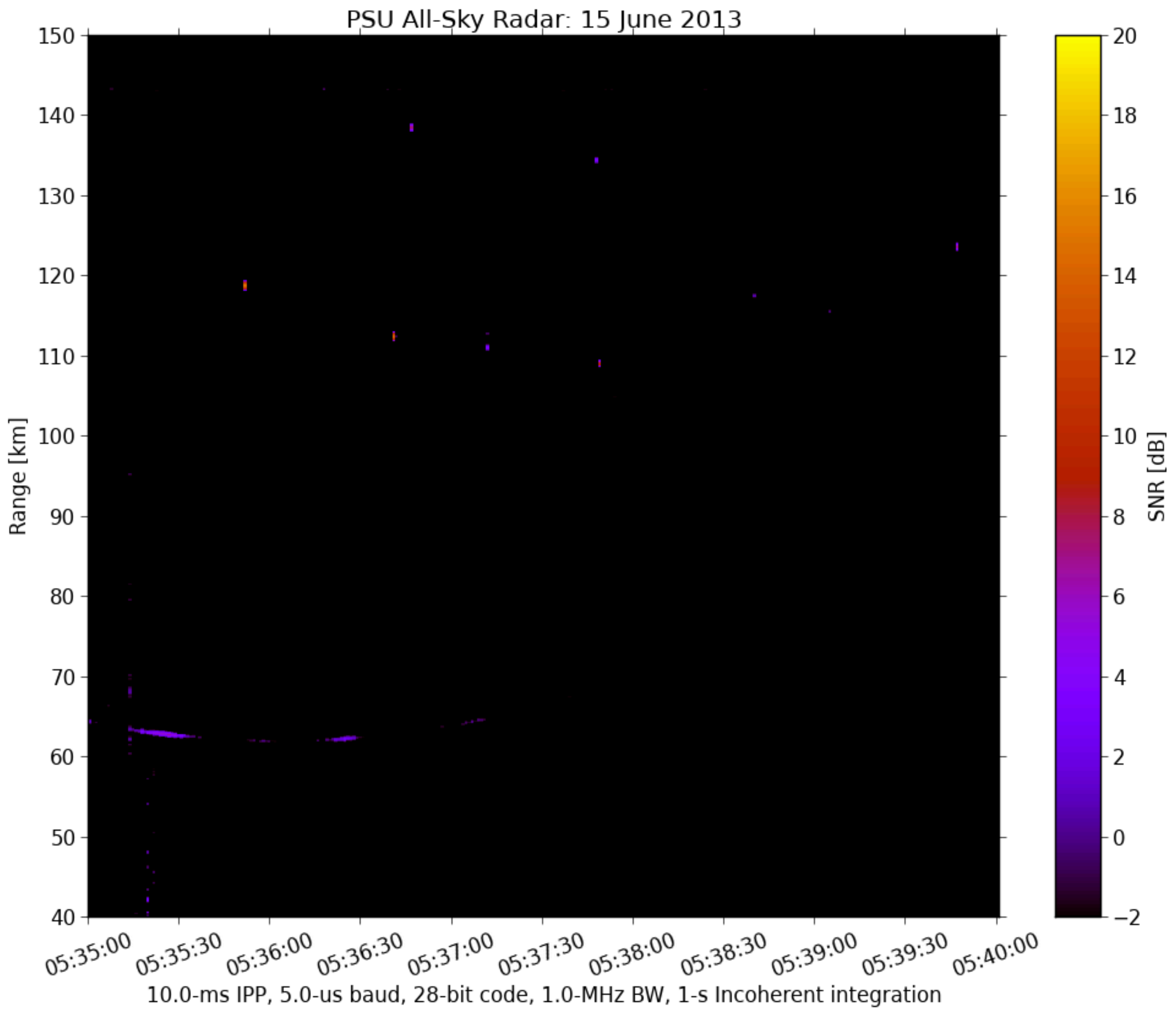


Figure 3.30: Example data plot from `rti_big.py` showing reflections from several specular meteors (reflections above 100 km) and an airplane (curved reflection appearing between 60 and 70 km).

There are many configuration options listed in both the script and configurable via the command line, including integration time,[9] integration type, data channel to plot, phase code, range window, and program verbosity. When operating in multi-image mode, the output images are named according to the start time of the data plotted in each image. Figure 3.30 shows an example output image from `rti_big.py`.

Figure 3.31 shows the general software process taken by the `rti_big.py` script. The program begins by collecting all the options from both the command line and parameters defined within the script. In multi-image mode, the plotter runs two functions — `reader()` (shown in blue) and `rti_plotter()` (shown in pink) — within a loop that checks if all images within the defined range have been plotted. Assuming this has not occurred, `reader()` first gathers several radar parameters from the HDF5 data file, including IPP, sampling rate, and start time. An indexing variable is used to traverse through the different tables in the data file. If the table index is within the calculated table index range, `reader()` then checks if enough tables have been read to integrate (if desired). Assuming more tables are needed, a data table is read from the HDF5 file. An FIR filter is applied to the data table for decoding the BPSK-coded data. This decoded data is then appended to a large array containing the decoded I and Q samples for one integration period. The table index is then incremented and the process repeats until enough tables have been read to integrate, at which point, the big IQ array is time integrated and added to the final IQ array. This integration process is then repeated until the final IQ array is full, indicating that the entire specified data range has been read, integrated, and added to the final IQ array. At this point, the table index is now beyond the table index range, and a power map from the final IQ array is generated. The power map is returned and then passed to `rti_plotter()`. After basic denoising, the power map is plotted in decibel units and saved to disk. At this point, `reader()` is called again and the process repeats until all images have been plotted.

[9]Integration time is currently limited to a minimum of 1 s.

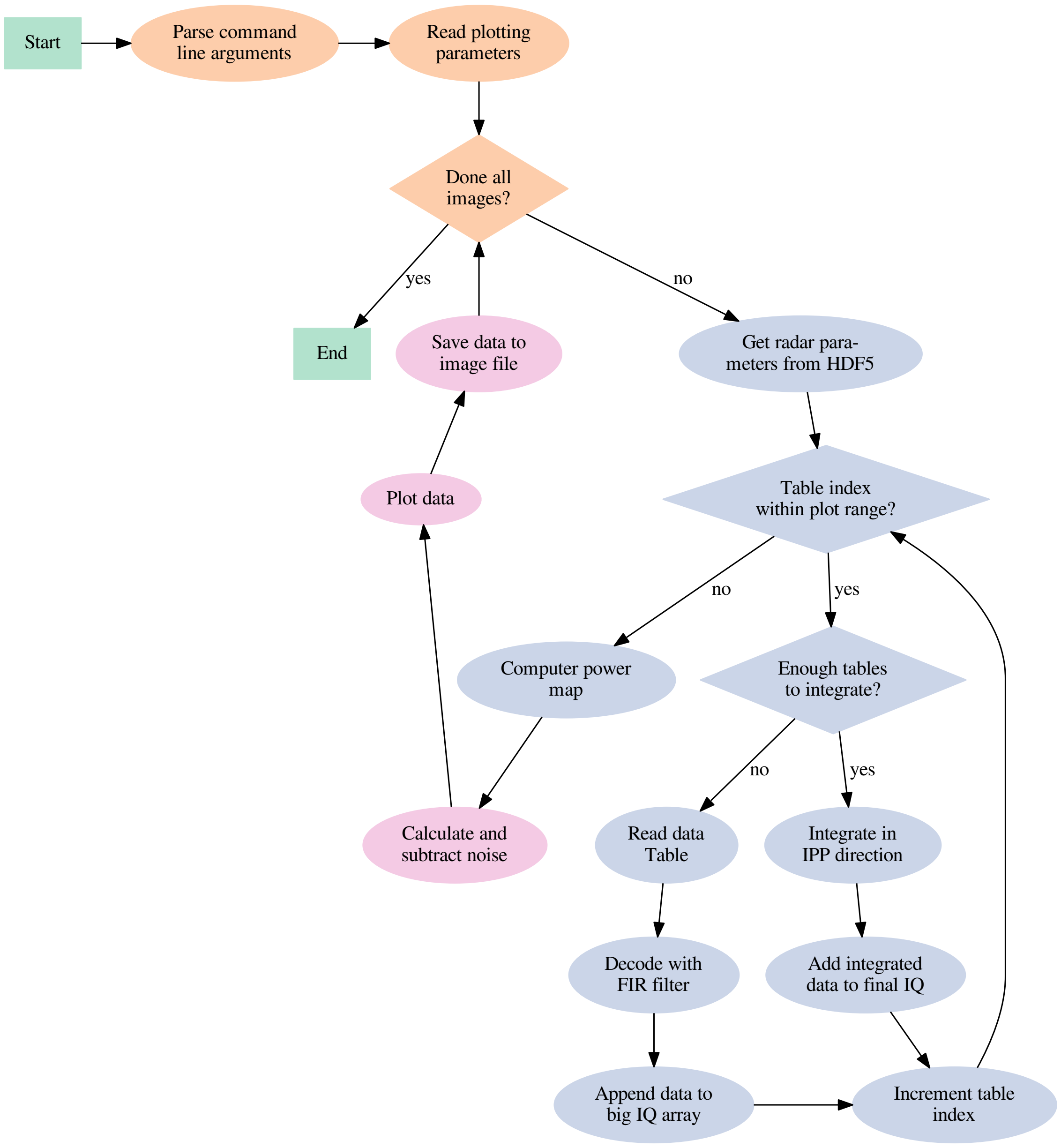

Start
Parse command line arguments
Read plotting parameters
Done all images?
yes
no
End
Save data to image file
Get radar parameters from HDF5
Plot data
Table index within plot range?
no
yes
Computer power map
Enough tables to integrate?
no
yes
Calculate and subtract noise
Read data Table
Integrate in IPP direction
Decode with FIR filter
Add integrated data to final IQ
Append data to big IQ array
Increment table index

Figure 3.31: General flow diagram of the RTI processing plotter for HDF5 data taken with GnuRadar.

3.2.3 Summary

This section has presented the overall hardware architecture of the PARIS software-defined radar system, in addition to many of the critical details. Additionally the basic software architecture of GnuRadar, used for data capture and control, was discussed. Preliminary operating procedures of PARIS and basic troubleshooting can be found in Appendix B.

3.3 Cognitive Interferometry Radar Imager

The Cognitive Interferometry Radar Imager (CIRI) is an advanced 50-MHz digital radar system designed for study of ionospheric phenomena, including ionospheric layers, specular and non-specular meteor trails, and other ionospheric plasma instabilities, using reconfigurable hardware and cognitive radio techniques to adapt to the various radar targets of interest. The system was developed out of the Illinois Radar Interferometer System (IRIS), designed at the University of Illinois (Urbana–Champaign), and is now a joint design and operations effort between students and faculty at The Pennsylvania State University and the University of Illinois. CIRI@PSU has been deployed and is currently in operation near The Pennsylvania State University (University Park) campus, at the Rock Springs Radio Space Observatory, both providing important magnetic mid-latitude ionospheric data and acting as a testbed for enhancements and upgrades to the CIRI system. A second deployment of the system, CIRI@Andes, is currently under construction at Observatorio de Huancayo in the Peruvian Andes. Deployment of CIRI@PSU, including set up of antenna arrays, receiver hardware, and transmitter hardware, integration of software, and preliminary operations, as well as system documentation, were the primary duties of the author.

3.3.1 System Overview

Unlike PARIS (discussed in Section 3.2), CIRI is not a traditional radar with full transmitter and receiver synchronization through logic-level signals in hardware. However, the transmitter and receiver are colocated and share two coaxial–colinear (COCO) antenna arrays, and like PISCO (discussed in Section 3.1), it uses a "reconstructive" synchronization technique in the receiver software. Figure 3.32 shows a high-level systems diagram of CIRI. The transmit segment (outlined in gold) of this radar system is composed of five main high-level components: 1) the direct digital synthesizer responsible for generating the RF carrier signal, as

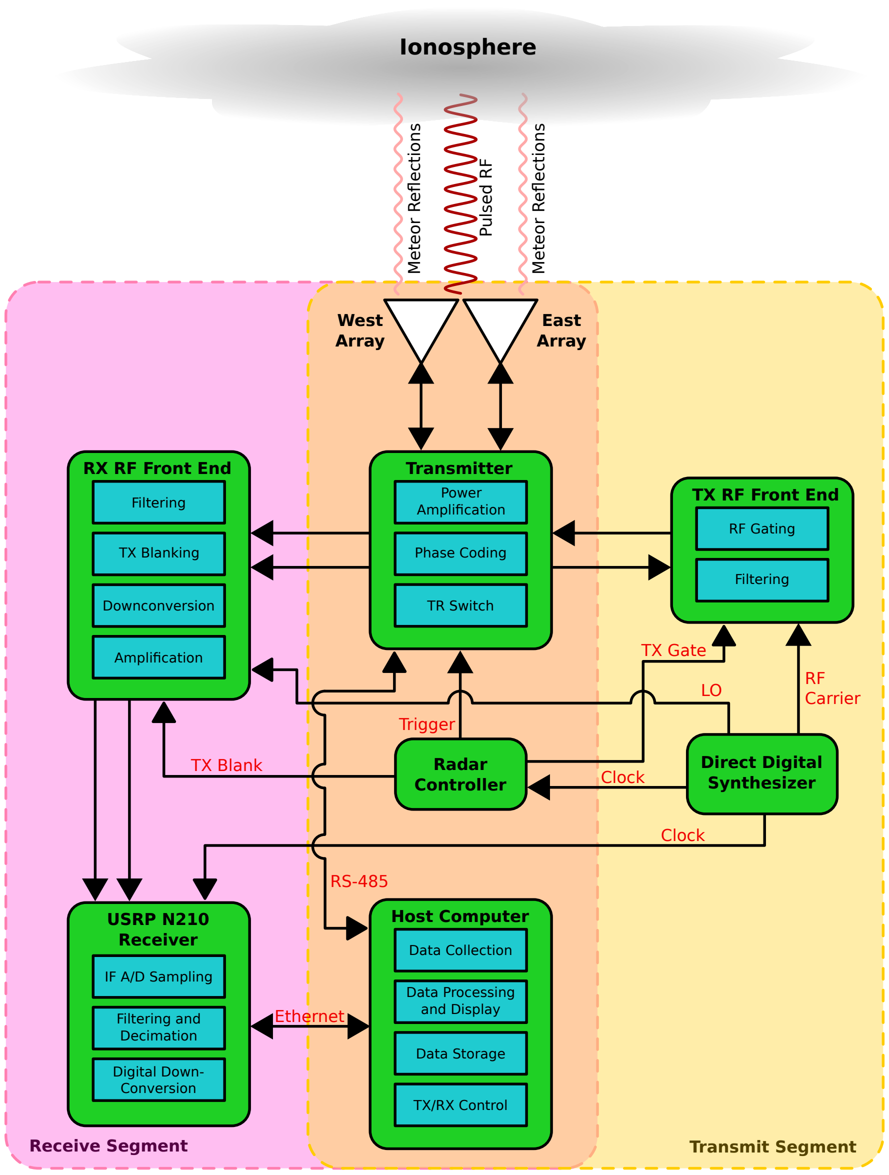


Figure 3.32: Overview of the Cognitive Interferomtry Radar Imager.

well as several clock signals for other components; 2) the radar controller responsible for generating several logic-level pulses necessary for radar operation, including the transmit trigger signal; 3) the transmit RF front end responsible for gating the RF carrier, as well as signal conditioning prior to transmission; 4) the transmitter responsible for power amplifying the pulsed RF carrier in preparation for transmission, as well as housing the transmit–receive (TR) switch; and 5) the antenna arrays responsible for transducing the electrical signal to RF energy. The antenna arrays radiate the pulsed RF signal through the atmosphere, where it then reflects and scatters off various targets, including meteor trails, ionospheric layers, and airplanes. Some of this reflected energy is captured by the antenna arrays, beginning the signal of the receive segment.

The six main components of the receive segment (outlined in pink in Figure 3.32) are: 1) the antenna arrays responsible for transducing the captured RF energy from target reflections into electrical signals; 2) the receive RF front end responsible for some basic hardware signal conditioning and downconversion to intermediate frequency (IF); 3) the USRP N210 receiver device responsible for digitization and preliminary signal processing of the received signals; 4) the host computer responsible for controlling the USRP N210 device and transmitter, as well as data collection, processing, display, and storage; 5) the radar controller responsible for generating a logic-level signal for receiver protection; and 6) the direct digital synthesizer responsible for providing a stable clock source for the USRP N210 device and a local oscillator for IF downconversion in the receive RF front end. All of the components in both the transmit and receive segments are further detailed in Section 3.3.2.

3.3.2 Implementation

The following sections discuss the various hardware and software components of CIRI in detail.

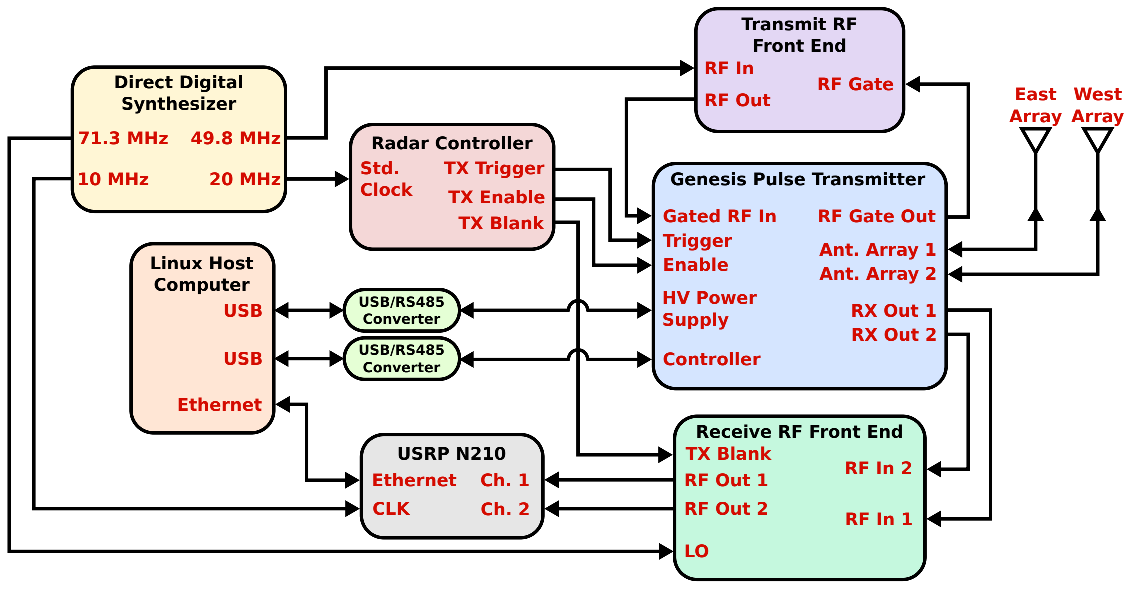


Figure 3.33: CIRI systems diagram (low-voltage DC and AC power connections omitted).

3.3.2.1 Hardware

A systems diagram of the different hardware components used by CIRI is shown in Figure 3.33. The following sections detail each of these components and discuss their interactions with the others.

3.3.2.1.1 Antennas As previously mentioned, CIRI uses two COCO antenna arrays (east and west) for both pulse transmission and interferometric reception. Each array consists of four individual COCO antennas suspended above the ground on polyvinyl chloride (PVC) pipe and steel masts, as illustrated in Figure 3.34 and shown partially in Figure 3.35. The coaxial cable segments of each antenna are fastened to a high-tension mounting cable with cable ties, so as to eliminate (potentially damaging) horizontal tension on the antennas themselves. The end of each mounting cable is attached to a hoisting rope, and wound around a pulley attached to a vertically mounted steel mast. Additionally, similar hoisting ropes are attached to the mounting cable at one-quarter length intervals, and threaded through eye bolts (acting as make-shift pulleys) attached to vertically mounted PVC pipes.

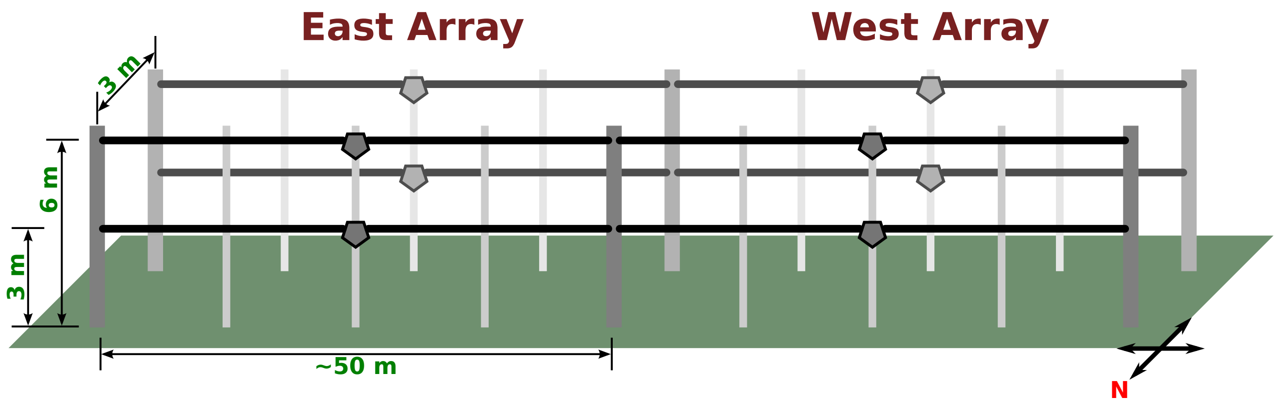


Figure 3.34: COCO antenna arrays layout diagram used by CIRI (not to scale).

Horizontal leveling of the antenna is accomplished by pulling all of the hoisting ropes tight and wrapping them around rope cleats on each of the steel masts and PVC pipes.

Viewed end-on from the east or west, the antennas are arranged in a square pattern, at heights of 3 and 6 m above the ground (corresponding to free-space electrical distances of $\lambda/2$ and λ, respectively), and separated by 3 m ($\lambda/2$) in the North–South direction. The two arrays are stationed next to each other in the East–West direction. When each of the four antennas of each array are combined[10] through the use of a quarter-wave ($\lambda/4$) transformer, this arrangement results in steering of the array's radiation pattern, such that the main lobe is directed to approximately a 16° elevation angle [37]. This specific elevation angle is important to ensure that the pulsed RF signals transmitted intersect Earth's magnetic field lines at right angles, which enables a host of ionospheric plasma phenomena to be visible to the radar system. In the azimuth direction, the antenna arrays' beamwidths are fairly narrow (due to the large azimuth aperture of the arrays), on the order of 2–5°, whereas in the elevation direction, the radiation patterns are much wider, approximately 20° [37].

3.3.2.1.2 Transmitter The transmitter used by CIRI is a custom-designed system from the Genesis Software PTS (Pulse Transmitter System) family, tuned specifically for the

[10]The north-most pair each have an extra $\lambda/4$ phasing cable on their feedlines.

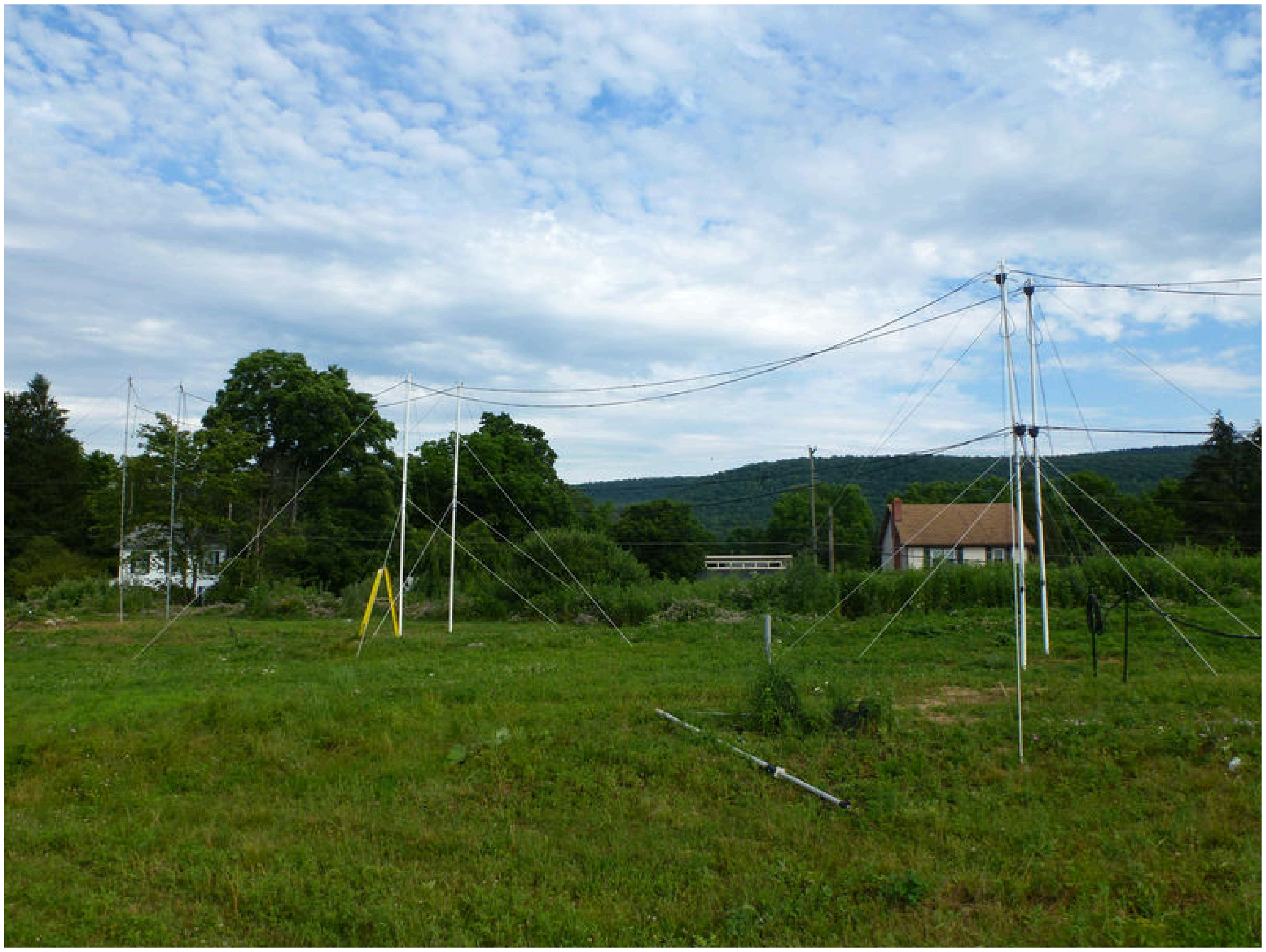


Figure 3.35: East half of the East COCO array used by CIRI, viewed from the northwest.

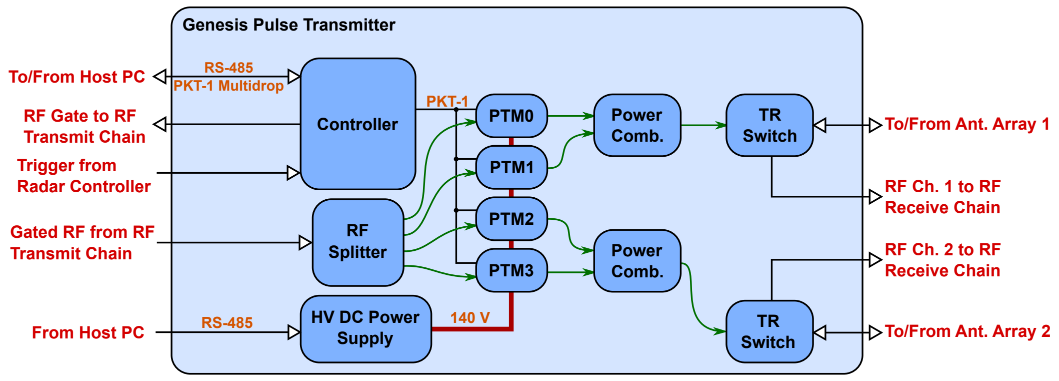


Figure 3.36: Block diagram of the Genesis Pulse Transmitter System.

49.8-MHz RF carrier used by CIRI. It is a solid-state transmitter that uses four software-controllable, 7.5-kW Pulse Transmitter Modules (PTMs), as shown in the transmitter block diagram (Figure 3.36). Additionally, a number of passive RF components (e.g., RF splitter, power combiners, and TR switches) and a high-voltage DC power supply are also housed in the rack-mount chassis of the transmitter, shown in Figure 3.37.

Using a multidrop communications protocol designed by Genesis Software (PKT-1) over an RS-485 interface, a host computer can issue commands to and listen for responses from the transmitter's interface controller module, called the Transmitter Supervisor Module (GTS). These commands include functions such as enabling transmitter output, defining the parameters of the transmitter pulse, and reading back diagnostic information from the system. The same PKT-1 multidrop protocol is used internally for the GTS to issue a slightly altered set of commands and control the four PTMs via their Transmitter Controller Modules (GTCs). The entire command listing for the PKT-1 protocol can be found in the Genesis PTS User Manual [45].

The pulsed RF signal path begins with the transmit trigger, generated by an external system, such as the radar controller. From the transmit pulse length stored in memory, the GTS generates and outputs the RF Gate signal, aligned to the beginning of the trigger signal. The external transmit RF front end uses this signal to gate the RF carrier (discussed

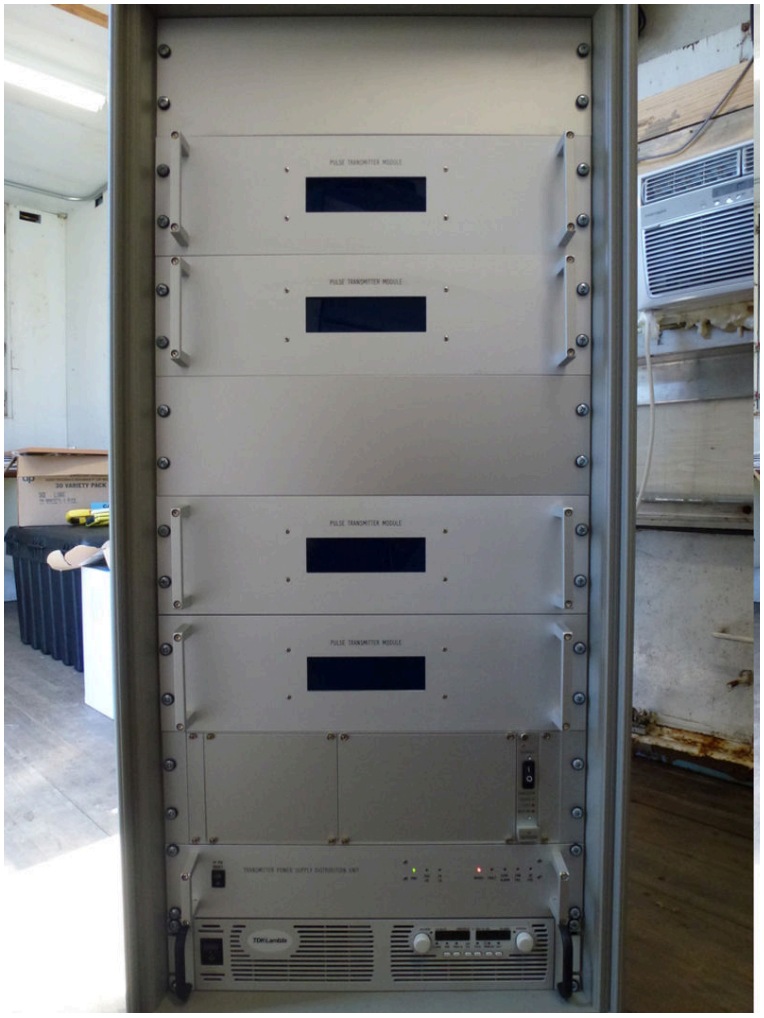


Figure 3.37: Front view of the Genesis PTS.

in Section 3.3.2.1.3). The Gated RF signal is then returned to the transmitter[11] and split out to each of the PTMs. Each of the GTCs uses the other transmit pulse parameters (e.g., pulse amplitude, BPSK coding, baud length, and pulse shape) to modify the Gated RF signal prior to power amplification and output by each of the PTMs. The PTM outputs are paired and summed by two power combiners, and the output of each power combiner is fed into a TR switch. During transmission, the TR switch connects the antenna array feedlines to the pulsed RF output, isolating the receive port on the TR switch in order to protect the sensitive receiver equipment.

The high-voltage power supply included with the transmitter supplies each of the solid-state power amplifiers in the PTMs with a $\sim$140-V bias in order to amplify gated RF signal up to 7.5 kW. Like GTS, this power supply can be remotely programmed via an RS-485 serial interface, although the only command used in normal operation of CIRI is the output enable command. A full command listing for this power supply can be found in the unit's manual [46].

3.3.2.1.3 Transmit RF Front End As previously mentioned, the transmit RF front end is responsible for gating the RF carrier signal in order to supply the transmitter with a pulsed RF signal. Figure 3.38 illustrates how this is accomplished, with the individual components listed in Table 3.9. The RF carrier signal generated by the direct-digital synthesizer is supplied to a gating switch, controlled by the RF Gate signal generated by the transmitter. This results in a Gated RF signal, which is then amplified and filtered to reduce out-of-band spurs and harmonics generated as a result of the gating operation. The signal amplitude of the RF carrier input was experimentally adjusted in order for the Gated RF output to meet the +10-dBm signal level input required by the transmitter.

[11]The required signal level for the Gated RF signal is +10 dBm for proper operation of the transmitter.

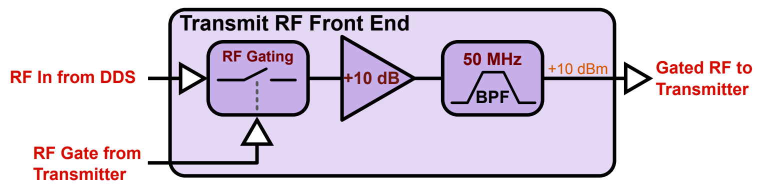


Figure 3.38: Block diagram of the CIRI transmit RF front end.

Table 3.9: Transmit RF front-end components used by CIRI.

Manufacturer	Part Number	Description
MiniCircuits	ZX80-DR230-S+	RF Gating Switch
MiniCircuits	ZX60-43-S+	RF Amplifier
MiniCircuits	SIF-50+	50-MHz Band-pass Filter

3.3.2.1.4 Receive RF Front End After the antenna arrays capture signal reflections from the ionosphere (and airplanes), the TR switches on the transmitter connect the antenna arrays to the two signal inputs of the receive RF front end, which provides preliminary signal conditioning and analog downconversion of the signal to IF. Figure 3.39 illustrates the configuration of the receive RF front end, with the individual components listed in Table 3.10.

The input signals from the antenna arrays begin by each passing through two band-pass filters, reducing the amplitude of out-of-band signals from the desired 49.8-MHz signal reflections. The first filter is a high-quality, high-order Butterworth band-pass filter, with a 5-MHz bandwidth and very steep rolloff. The second filter is a lower order filter with a slightly wider bandwidth of approximately 17 MHz. As second layer of receiver protection

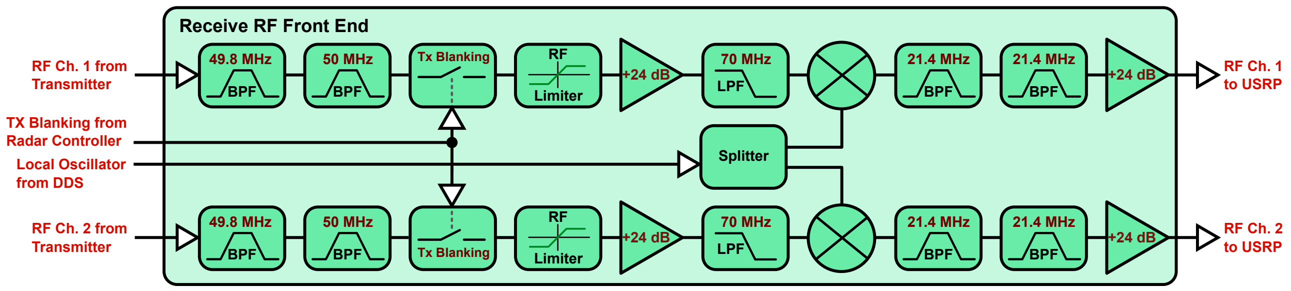


Figure 3.39: Block diagram of the CIRI receive RF front end.

Table 3.10: Receive RF front-end components used by CIRI.

Manufacturer	Part Number	Description
TTE, Inc.	KB8-49.8M-5M-50-720A	49.8-MHz Band-pass Filter
MiniCircuits	SIF-50+	50-MHz Band-pass Filter
MiniCircuits	ZX80-DR230-S+	RF Gating Switch
MiniCircuits	VLM-33-S+	RF Limiter
MiniCircuits	ZFL-500LN+	+24-dB RF Amplifier
MiniCircuits	SLP-70+	70-MHz Low-pass Filter
MiniCircuits	ZX05-1L-S+	RF Mixer
MiniCircuits	SIF-21.4+	21.4-MHz Band-pass Filter
MiniCircuits	SBP-21.4+	21.4-MHz Band-pass Filter
MiniCircuits	ZFL-500LN+	+24-dB RF Amplifier

(in addition to the transmitter's TR switches), an RF blanking switch is used to isolate the input from the remainder of the receive RF front end during the transmit pulse, controlled by a transmitter blanking signal generated by the radar controller. Following the blanking switch is an RF limiter, clamping the input signal to a maximum of +10 dBm, in the event that any large-amplitude signal spikes make it through both the TR switch and TX blanking switch. Following the RF Limiter is a +24-dB wideband amplifier, used to increase the in-band return signal level. A 70-MHz low-pass filter is then used to reduce any high-frequency signal content that may have made it through the signal chain, prior to the IF downconversion in order to avoid adding mixing products into the desired signal band. Using the ∼71.3-MHz local oscillator signal generated by the direct-digital synthesizer, a mixer is used to demodulate the RF signal to the 21.4-MHz IF, with the output filtered by two 21.4-MHz band-pass filters to remove the ∼121-MHz mixer product content from the IF signal. Finally, the output of the two band-pass filters is passed through another +24-dB amplifier, further increasing the in-band signal content. The output of each channel's final amplifier is each connected one input channel of the USRP device for digitization and further signal processing.

Using the same MATLAB script noise figure and overall system gain as used with PARIS

(Appendix A), the noise figure and gain calculated to be 10.4 dB and 32.2 dB, respectively, for CIRI's receive RF front end, including the USRP N210's ADC (see Section 2.2.1.2), but excluding the antenna array (as the noise temperature/noise figure for the COCO arrays are unknown).

3.3.2.1.5 USRP The core of the receiver hardware is the USRP N210 device, which digitizes the incoming RF signals from the receive RF front end. Like the PISCO (Section 3.1) and PARIS (Section 3.2) USRP receivers, the USRP also uses a BasicRX daughterboard that provides 50-Ω termination and transformer coupling of the signal to the USRP's ADCs. Unlike the USRP1 devices used with PISCO and PARIS, the USRP N210 has a sampling rate of 100 MHz, yielding a total sampling bandwidth of 50 MHz. This 100-MHz clock is generated by an onboard oscillator locked to the external 10-MHz input with a phase-locked loop (PLL). Instead of using the USB 2.0 interface used by the USRP1 device for data transfer and configuration, the USRP N210 uses a gigabit Ethernet interface, allowing a much higher signal bandwidth to be transferred to the host computer. Also unlike the USRP1, the USRP N210 has a non-volatile memory for storing the FPGA bitstream, allowing the device to run without downloading the FPGA configuration every time it is powered up.

In the configuration currently used by CIRI (400-kHz sampling rate), a USRP1 device could easily be substituted for the USRP N210 with minimal software impact, due to the common UHD software interface provided by Ettus Research. However, such a substitution could potentially result in a lower overall system dynamic range, as the USRP1 samples with a 12-bit ADC, whereas the USRP N210 samples with a 14-bit ADC.

3.3.2.1.6 Radar Controller The radar controller used by CIRI is the same system used in PARIS (see Section 3.2.2.1.2 for more details). It is responsible for generating the synchronized timing pulses used to control both the transmitter and the receive RF front end in CIRI. Table 3.11 lists the radar controller output signals and Figure 3.40 graphically

Table 3.11: Radar controller output signal descriptions used by CIRI.

Name	Description	Destination
TX Trigger	Transmit pulse trigger	Transmitter
TX Enable	Transmitter enable signal	Transmitter
TX Blanking	Receiver disable for protection from transmitter	RF RX Chain

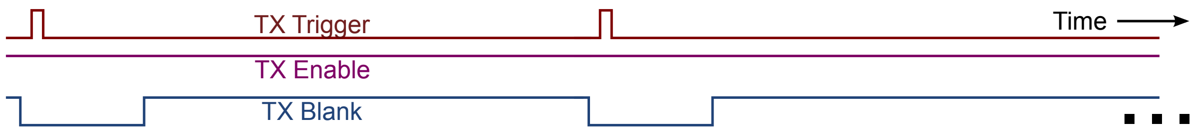


Figure 3.40: Radar controller output signals timing.

illustrates the timing relationship between the different output signals.

The TX Trigger signal is used to signal the transmitter to begin a new gate pulse, and repeats at the specified inter-pulse period (IPP) interval. The pulse width of the TX Trigger signal has no effect on transmitted pulse, as the transmitter only detects the rising edge of the pulse to generate and align the TX Gate signal.[12] The TX Enable signal is normally set to logic high for all time, as it is one of the three necessary enable signals for the transmitter (discussed further in Section 3.3.2.2.2). The TX Blanking signal is similar to the corresponding signal used in PARIS; it disables a portion of the receive RF front end using an active-low logic signal slightly wider than the width of the transmitted RF pulse (accounting for transient effects).

3.3.2.1.7 Direct Digital Synthesizer For proper operation of the radar system, all of the components must be fully synchronized, with all clock signals derived from a single master clock. The Novatech 409B (the same device used in PISCO — see Section 3.1.2.1.3) is a four-channel programmable oscillator that is used by CIRI to provide stable and phase-locked clock signals for the various components of the radar system. Table 3.12 lists the output signals of the Novatech 409B device, all with a programmed phase of 0°.

[12]The minimum recommended TX Trigger pulse width is 1 µs in order to ensure the transmitter properly recognizes pulse as a true trigger signal.

Table 3.12: DDS output signals.

Channel	Frequency (MHz)	Amplitude (Counts)	Destination
0	20.000000	1023	Radar Controller Std. Clock
1	10.000000	1023	USRP Reference Clock
2	49.800000	160	RF Input to Transmit RF Front End
3	71.284375	1023	Local Oscillator Input to Receive RF Front End

Table 3.13: Basic hardware specifications of the CIRI receiver GPC.

Processor Family	AMD Phenom 9950, 2.6 GHz
RAM	8 GB DDR2 (4×2 GB)
Hard Drive (OS + Software)	250 GB SATA
Removable Hard Drive (Data)	4 GB SATA via USB 2.0 dock
Network Interface (USRP)	1000 Mbps Ethernet
Network Interface (Internet)	100 Mbps Ethernet

As is the case with PARIS, the only output signal with an amplitude dependence is the RF carrier input signal, whose amplitude was experimentally adjusted such that the amplitude of pulsed RF output for the transmit RF front end meets the +10-dBm level requirement. All other DDS output signals (20 MHz for the radar controller, 10 MHz for the USRP, and ∼71.3 MHz for the local oscillator) are set to the maximum amplitude count, as their exact signal amplitude is not critical.

3.3.2.1.8 General-Purpose Computer The host general-purpose computer used by CIRI is responsible for data capture, processing, and storage, as well as control of the transmitter. The basic hardware specifications of the GPC are listed in Table 3.13, although most modern laptop or desktop computers (with >6 GB of RAM) should be suitable for acting as the CIRI GPC. The minimum RAM limitation stems from the fact that raw data files are stored in system RAM prior to data processing (discussed in Sections 3.3.2.2.3 and 3.3.2.2.4).

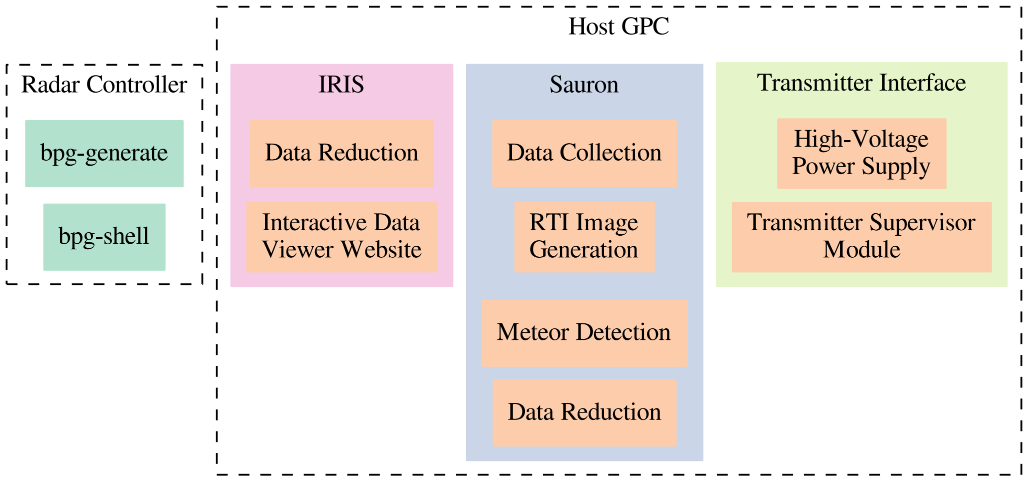


Figure 3.41: Overview of the CIRI software.

3.3.2.2 Software

As a software-defined radar system, much of the flexibility of CIRI lies within the host software that runs the CIRI system. The following sections discuss the operation of each of the various software components (overviewed in Figure 3.41) that are used for transmitter and receiver control, as well as data collection, processing, and display for CIRI.

3.3.2.2.1 Radar Controller As previously mentioned, the radar controller used in CIRI is the same system used by PARIS. Thus, the software interface for defining and generating output modes (`bpg_generate`) and controlling the signal outputs (`bpg_shell`) are the same, and the reader is referred to Section 3.2.2.2.1 for a detailed discussion of this software.

3.3.2.2.2 Transmitter Interface In order to control the Genesis transmitter, the host computer must communicate with two different devices on the transmitter: the high-voltage power supply and the GTS. Both devices use an RS-485 interface for communication, which is a differential hardware interface, eliminating unwanted common-mode interference (e.g., 60-Hz mains hum) that could potentially interfere with proper communications and put the transmitter in an unwanted operational mode. The software used to communicate with these devices is discussed in the following sections.

3.3.2.2.2.1 High-Voltage Power Supply The Genesis transmitter rack houses a high-voltage DC power supply, which is used by the power amplifiers in each of the PTMs. Two simple Python scripts are used to enable and disable the output of the high-voltage DC power supply in the Genesis PTS: `hv_enable` and `hv_disable`. In order for the transmitter to transmit, the high-voltage power supply outputs must be enabled. When the transmitter is not in use, it is recommended that the high-voltage power supply output be disabled as well for safety.

3.3.2.2.2.2 Transmitter Supervisor Module In order to control the behavior of the transmitter (e.g., output enable, transmit pulse parameter definition, etc.), the host computer must relay commands over the GTS RS-485 interface. While the PKT-1 packet structure developed by Genesis is fairly simple (defined in [45]), manually constructing the command packets is tedious and an inefficient method of transmitter control. To alleviate this, a Python script, `txcli.py`, was written to issue transmitter commands by name, automatically forming the correct PKT-1 packet for each command.

This program uses a Python dictionary structure as a command database (`TxRxDefs.py`), storing all of the commands listed in the Genesis PTS User Manual [45]. The `txcli.py` script parses the command line arguments and attempts to find the requested command in the command database. If successful, `txcli.py` opens the serial port and sends the command to the transmitter, along with any data bytes required by the command. The script then waits and listens for a command response from the GTS, parsing and displaying the responses on the host GPC, as defined in `TxRxDefs.py`.

Table 3.14 lists the most commonly used GTS commands (along with a short description) in the current CIRI configuration. Typically, the user defines a pulse configuration, saves

Table 3.14: Most common GTS commands used for CIRI.

Command Name	Description
GTS_ENABLE	Enable or disable RF output.
GTS_SETPULSE	Set the transmit pulse parameters (amplitude, phase code, baud length, pulse shape, etc.)
GTS_SAVPULSE	Save the defined transmit pulse parameters to non-volatile memory
RESET	Reset the transmitter GTS and PTMs

it to non-volatile memory, and then resets the transmitter, at which point the new pulse configuration is loaded by default and the signal output can be enabled as desired.

3.3.2.2.3 Sauron "One program to rule them all" is the philosophy behind the naming of the Sauron program, and indeed it does accomplish many different software tasks necessary for CIRI (but not quite all of them — see Section 3.3.2.2.4). Sauron provides functionality to capture data from the USRP device, plot RTI power map images in realtime, use image-processing techniques to detect a variety of meteor events, and perform data reduction. The data capture functionality of the program is implemented through the use of a GnuRadio flowgraph. Two channels of I/Q data are streamed from the USRP device using the `uhd.usrp_source` signal block, which in turn is passed through a matched filter for BPSK decoding. The taps used by this filter are simply the BPSK code coefficients (+1 or −1) with the correct multiplicity for the specific sampling rate and baud length. Finally, the samples from each channel are interleaved into the output binary data file, stored in the host GPC's RAM (making use of shared memory).

Sauron's RTI plotter functionality then reads these raw data files and produces RTI power maps in realtime, allowing the user to view the data as they are streamed in from the USRP device. Based on the power map that is generated, the transmitter pulse is detected (averaging across several time bins and looking for the peak power amplitude) and the data are realigned with the transmitter pulse at a range of 0 km. Additionally, a noise level estimate is made and used for calculating SNR, which is plotted and saved to image files at regular time intervals. These RTI power maps are also used for meteor echo detection, one of the most distinguishing features of Sauron. The meteor detection code utilizes a Gaussian Mixture Model (GMM) algorithm that employs image-processing and machine learning techniques in order to identify and classify several different types of meteor event echos captured by CIRI. Further information on the theory and implementation behind Sauron's meteor detection can be found in [47, 48].

Additionally, several different data reduction techniques can be used with the data captured by Sauron. The simplest form of data reduction implemented by Sauron uses thresholding on the RTI power map. A data mask is created from the power map, assigning a '1' only to pixels with an SNR above a specified threshold, with '0's elsewhere. The mask is multiplied by the raw I/Q data array, preserving only the significant data samples. This array is written to a data file of the same format as the raw I/Q samples, resulting in the same overall data file size. However, an external compression utility (e.g., `bzip2`) can take advantage of the large number of 0's and significantly reduce the file size. Typical compression ratios seen when operating the system in this mode have been >9:1, although this figure is dependent on the number of ionospheric reflections as well as any type of interference above the power threshold.

A second, more involved data reduction technique is also available for use with Sauron. This method makes use of the meteor detection and classification capabilities. For each event detected, a rectangular "window" around the event is created, from which the classifier determines the type of event (e.g., specular, non-specular, interference, etc.) and saves event statistics. The window generated is the key for this method of data reduction. If the classifier determines the event is not interference, the bounding coordinates of the window are saved and applied to the raw I/Q data array, and a MATLAB/Octave workspace (*.mat) file is generated containing the event I/Q channel data, start time, duration, and range information, allowing full reconstruction of the event for further processing. After all processing of the raw I/Q data file has been completed, it is removed, leaving only the *.mat event information, which is on the order of several hundred kilobytes. Like the previously described data compression technique, this method is also dependent on the number of ionospheric reflections; however, during typical operation, a 1.8-GB data file has been shown to reduce to around 5 MB.

3.3.2.2.4 IRIS As used by CIRI, the IRIS software provides two important functions: data reduction and an interactive data viewing website.[13] In order to reduce the raw I and Q data files sampled from the USRP (by Sauron), the IRIS software computes a power map (synchronized using the transmitter pulse, as in Sauron), Doppler periodogram, and magnitude and phase coherence between the two receive data channels for interferometry, and stores these results in MATLAB/Octave workspace (*.mat) files. Each of these workspace files contain one second of the aforementioned computed data products, used for further processing by the interactive website tools.

Several interactive tools (developed by researchers at UIUC) are available to explore the data taken by CIRI, using the previously discussed *.mat workspace files. `realtime` allows users to view a power map of the most recent five minutes of data, as well as plot range slices of the power map for more detailed inspection. `datascope` is a more advanced tool, allowing the user to zoom in on specific regions of interest, as well as view phase coherence and spectrogram plots in addition to the standard power map. The `archive` tool is used to view static images of previously generated power maps (eliminating the need to recompute these for old data sets). Finally, `rtitool` allows the user to concatenate and display power maps from multiple data files, which is useful for viewing a complete picture of longer duration events, such as ionospheric layers.

[13]The IRIS software also provides a data collection feature, but it is not used in the CIRI configuration, as Sauron is responsible for data collection.

3.3.3 Summary

In this section, the hardware design and basic data collection, processing, and control programs of CIRI has been presented. Preliminary operating procedures and basic troubleshooting checklists can be found in Appendix B.

Chapter 4

Preliminary Results

This chapter presents a brief overview of some of the key results from preliminary operations of the three modern ionospheric sensor systems discussed in Chapter 3. The data products (i.e., RTI power map images, event statistics, etc.) generated by each of these systems in their early stages of operation are two-fold in purpose. First, they help to validate system functionality within typical expected operating parameters and environments. Second, these results help to discover and diagnose bugs, limitations, and failure situations of these systems. The results from all three systems validate functionality; however, in each case, there are multiple issues to be resolved before reliable and meaningful science operations should begin. These issues are discussed both in this chapter and in Section 5.2.

4.1 PSU Ionospheric Sounder for Chirp Observations

As mentioned in Section 3.1, the first deployment of the PISCO receiver is located at Arecibo Observatory in Puerto Rico, operating in conjunction with the previously installed CADI system. At the time of writing, the system has been operating continuously for several months, storing both an ionogram image and compressed HDF5 data file for each ionosonde sweep by CADI. For the reader's convenience, the radar parameters of the two systems are

Table 4.1: Radar parameters of the PISCO/CADI ionosonde system.

Parameter	Value	Units
IPP	25	ms
Baud Length	40	µs
Phase Code	Barker 13	—
Start Frequency	1	MHz
Stop Frequency	20	MHz
Number of Frequencies	300	—
Frequency Dwell	200	ms
Repetition Interval	15	min
Transmit Power (peak)	$\sim$600	W
PISCO Sampling Rate	500	kSps

repeated in Table 4.1.

The following sections present a few of the ionogram plots generated by the system, specifically highlighting the sensitivity of the PISCO receiver by demonstrating its ability to resolve several ionospheric structures typically detected by ionosondes. Additionally, several of the challenges and shortcomings of the system are discussed with the aid of ionogram plots in which the reconstructive synchronization technique failed.

4.1.1 Positive Results

Since PISCO has been collecting and processing data, many correctly synchronized ionogram plots have been generated. As expected, these plots show slowly time-varying ionospheric structures and phenomena familiar to ionosonde users, including hop reflections, single and multiple F-region layers, Spread-F layers, the ordinary and extraordinary (O+X) mode split, and Sporadic-E layers. The following sections show and discuss examples of each of these, as recorded by PISCO.

4.1.1.1 Hop Reflections

Often, especially when D-region absorption is low (e.g., at night), multiple "hop" reflections can be seen on the ionogram plots generated by PISCO. This phenomenon occurs when ionospheric reflections bounce off the Earth and ionospheric regions one or more times before being attenuated below the noise floor, appearing on the ionogram at intervals of the original reflection height [5, 6]. In a sense, this phenomenon is similar to range aliasing, except that it is visible because a longer IPP is used, rather than too short of an IPP. Figure 4.1 shows an example of at least 12 hop reflections, not only showing low ionospheric absorption, but also demonstrating the high sensitivity of the receiver.

4.1.1.2 Multiple F-region Layers

Sometimes, multiple layers within the F region of the ionosphere appear, and both can be seen on ionogram plots, with F_2 appearing at higher range and frequency than F_1 [6]. This can be illustrated by Figure 4.2, where the first layer appears between around 3.3 MHz and 4.5 MHz, and the second layer extends from 4.5 MHz beyond 6 MHz. Both layers are thin in nature.

4.1.1.3 Spread-F Layer

Spread-F layers are a rarer ionospheric phenomenon whose occurrence and physical process is still under active research today, particularly by the CIRI@Andes system. In ionogram plots, Spread-F can be observed as an F-region layer that appears "smeared" across many ranges, with a thickness much greater than the typically observed thin F-region layers [6]. Figure 4.3 illustrates a Spread-F layer observed by PISCO, ranging from approximately 350 km to 500 km.

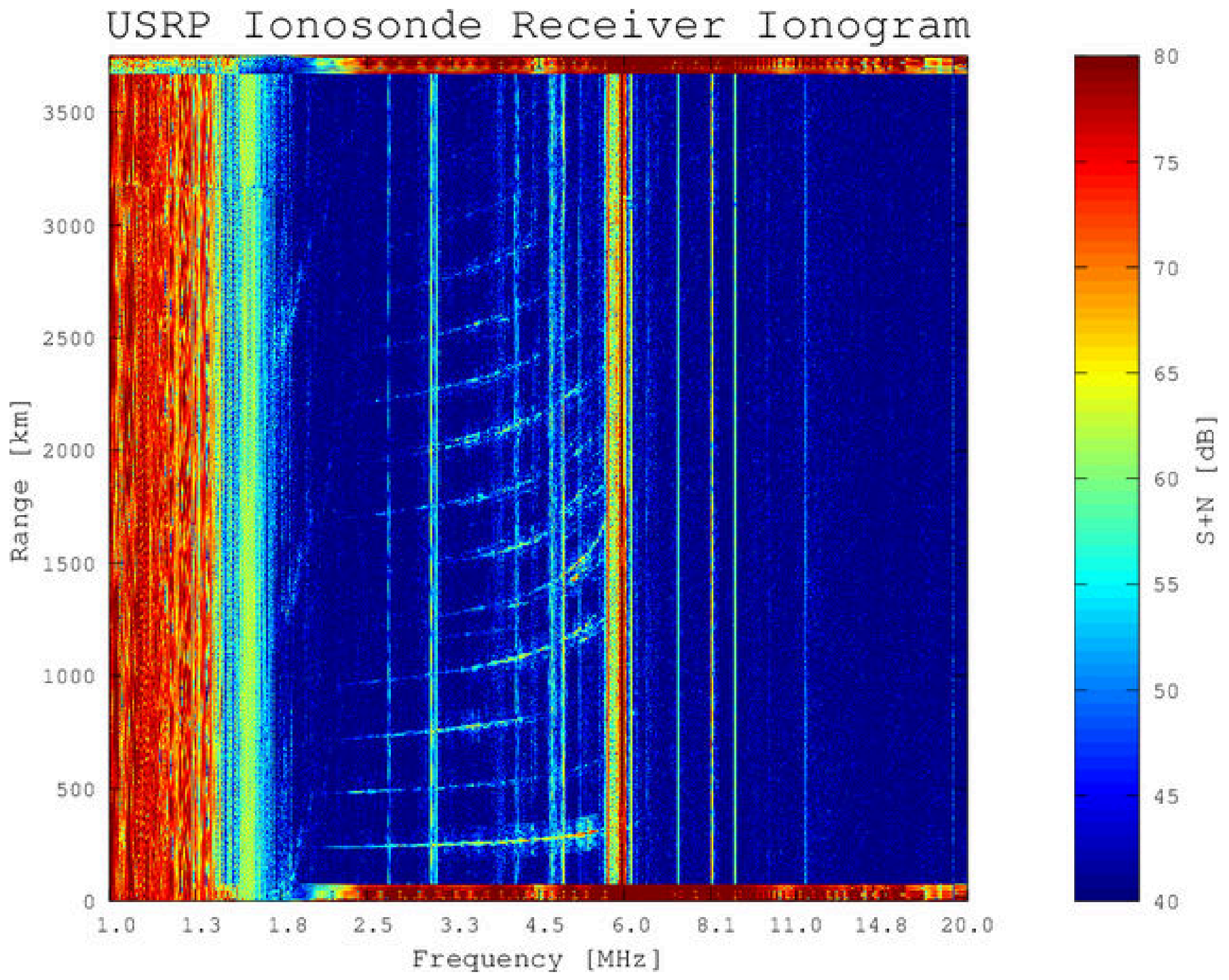


Figure 4.1: Ionogram power map generated by PISCO on 3 June 2013 at 5:15 AST, with 12+ "hop" reflections are visible.

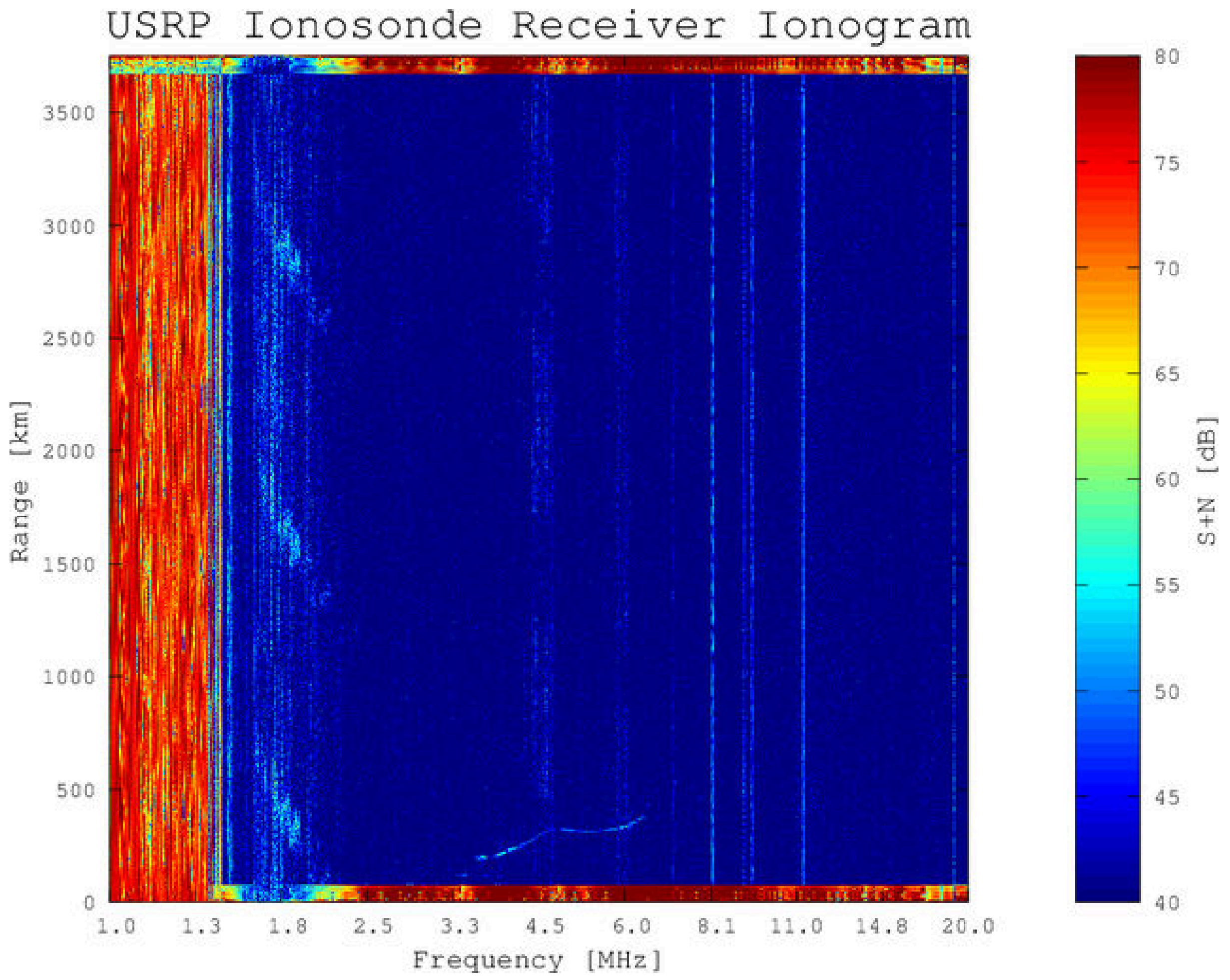


Figure 4.2: Ionogram power map generated by PISCO on 4 July 2013 at 8:30 AST, showing reflections from multiple F-region layers, F_1 and F_2.

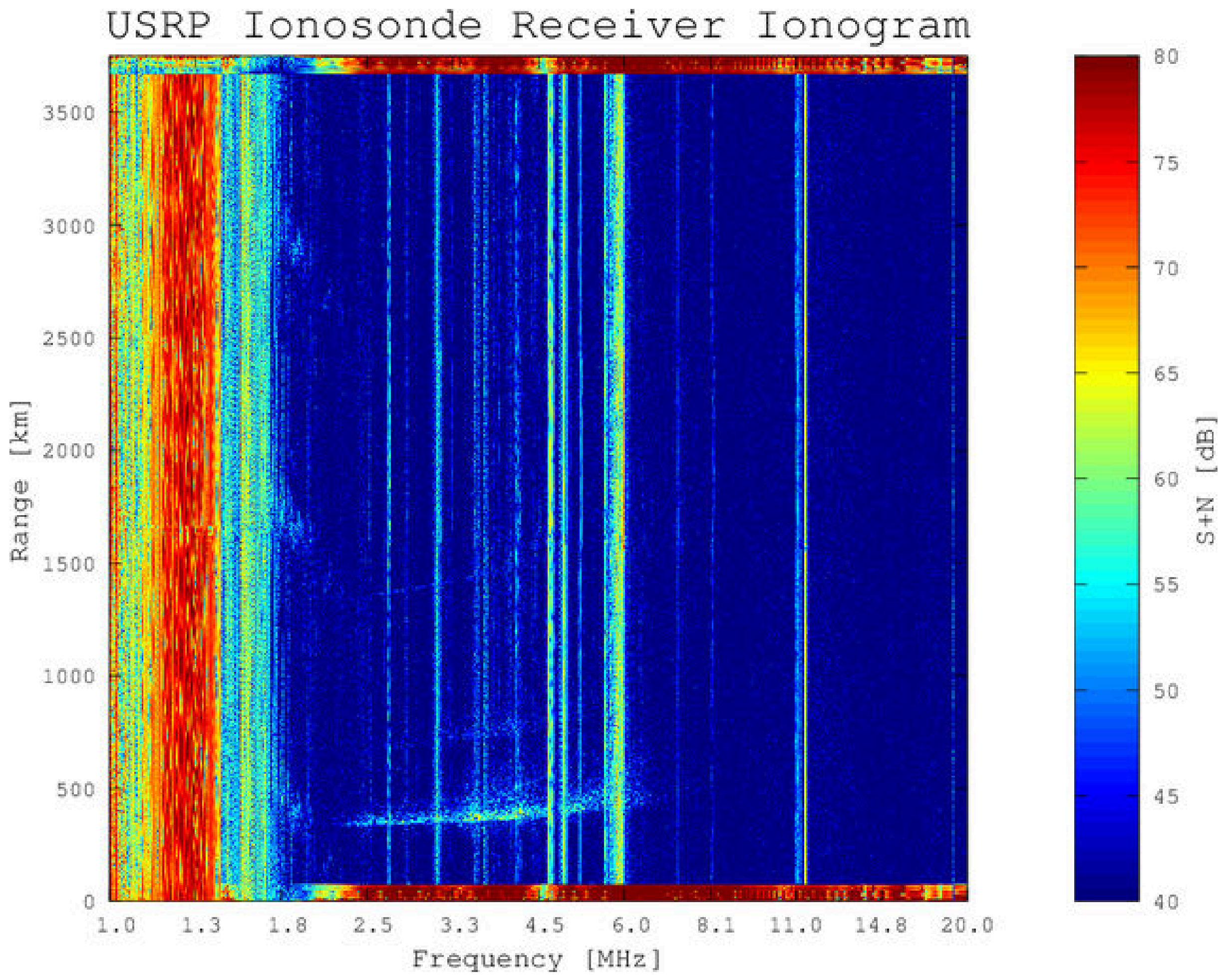


Figure 4.3: Ionogram power map generated by PISCO on 29 June 2013 at 4:45 AST, showing reflections from a Spread-F layer.

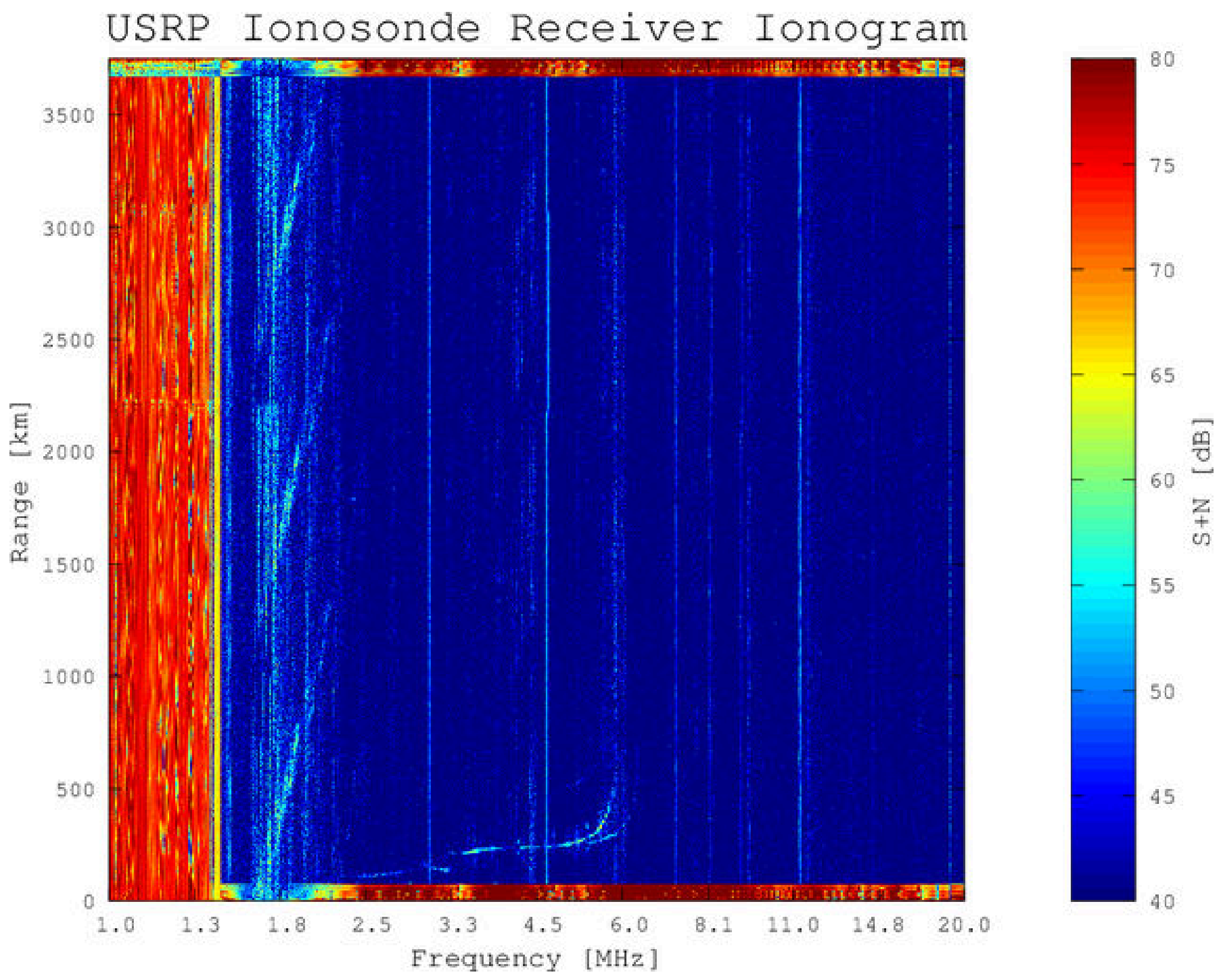


Figure 4.4: Ionogram power map generated by PISCO on 5 July 2013 at 7:45 AST, showing the ordinary and extraordinary mode split.

4.1.1.4 O+X Mode Split

Under the right conditions, interaction of the Earth's magnetic field with the ionosphere can increase ionospheric reflection, resulting in a second reflection trace (called the extraordinary (X) mode) that "splits" out of the typical F-region reflection trace (called the ordinary (O) mode) [49]. This phenomenon can be seen in Figure 4.4, with the O+X mode split occurring at approximately 5 MHz.

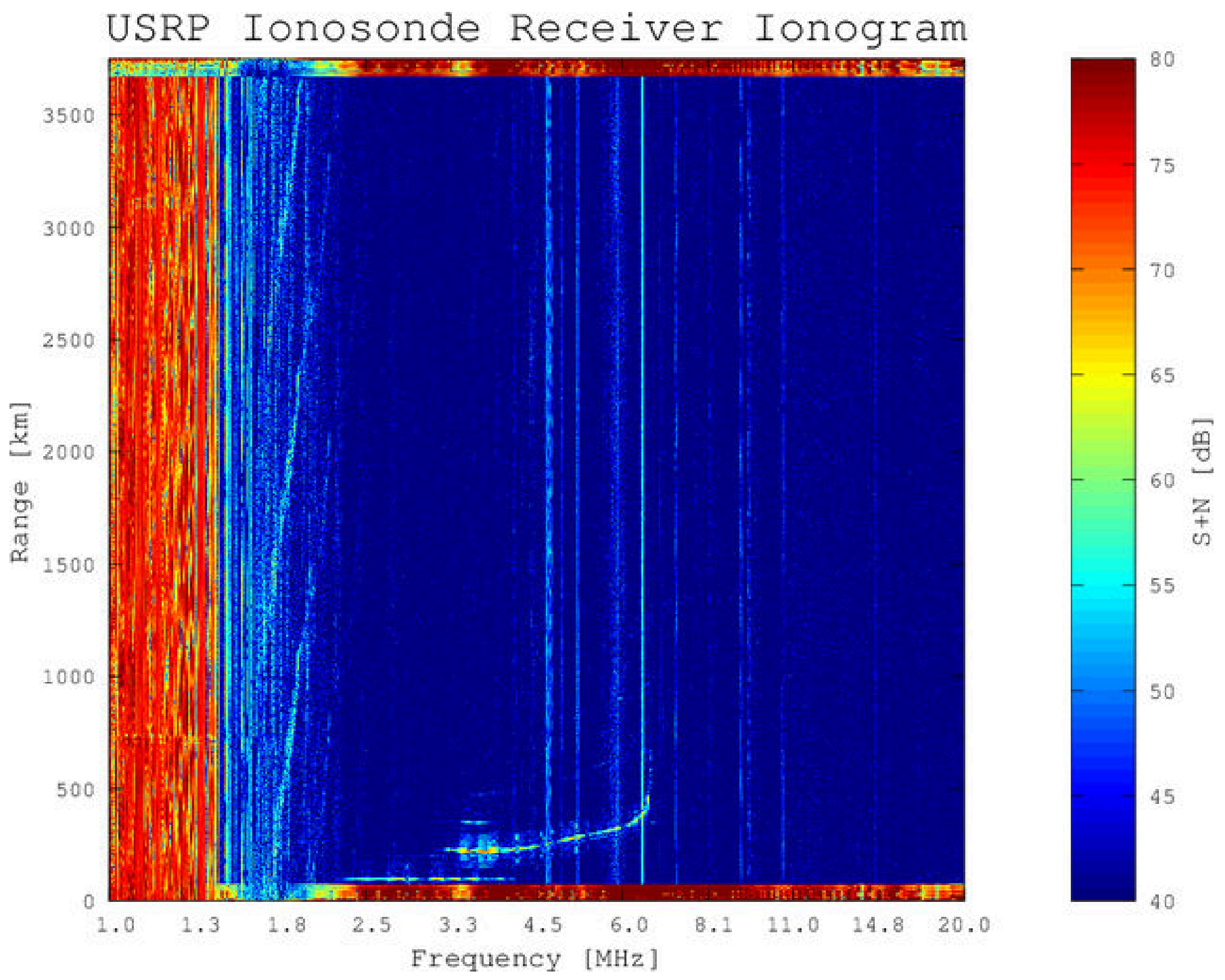


Figure 4.5: Ionogram power map generated by PISCO on 22 June 2013 at 7:00 AST, illustrating a Sporadic-E layer.

4.1.1.5 Sporadic-E Layer

Sporadic-E layers, layers that form occasionally in the E region, are typically limited to around 100 km in range and appear as thin, flat reflection traces on ionogram plots [6]. Figure 4.5 illustrates a Sporadic-E layer with characteristic thin, flat appearance, spanning from about 2 MHz to 4 MHz, in addition to the upper F-region layer.

4.1.2 Issues

While many correctly synchronized ionogram plots have been generated by PISCO, there are still a few issues with the system that range from a being a minor nuisance in correctly synchronized plots to completely rendering the output plot useless. The following sections detail these issues loosely in order of increasing severity. Finally, some error statistics from the collected data are presented.

4.1.2.1 Coding Sidelobes

Like the PARIS and CIRI systems, CADI uses binary phase encoding (i.e., BPSK modulation) as a pulse compression radar technique to retain good range resolution while transmitting a high power RF pulse. The decoding technique used by PISCO involves applying a matched filter to the baseband data, ideally resulting in the autocorrelation function of the code for perfect reflections. However, for large amplitude reflections, the code sidelobes begin to appear above the background noise level. Figure 4.6 shows these sidelobes on the strong reflection between around 5 and 6 MHz, and it appears to widen the reflection in range by several times. Although these sidelobes are easy to identify, there is the possibility of masking weak reflections behind strong sidelobes.

4.1.2.2 Interference

Background interference is a pronounced feature common to all of the ionogram plots generated by PISCO. It tends to appear as vertical lines (often with striations) within the plotted data, as these radio signals are generally not pulsed as radar signals are, but rather more continuous in nature (at least on the timescale of the radar IPP). The background interference certainly varies over the course of the day, as different radio operators and services use different parts of the radio spectrum. Additionally, as ionospheric absorption varies with the time of day, so do the propagation of radio waves.

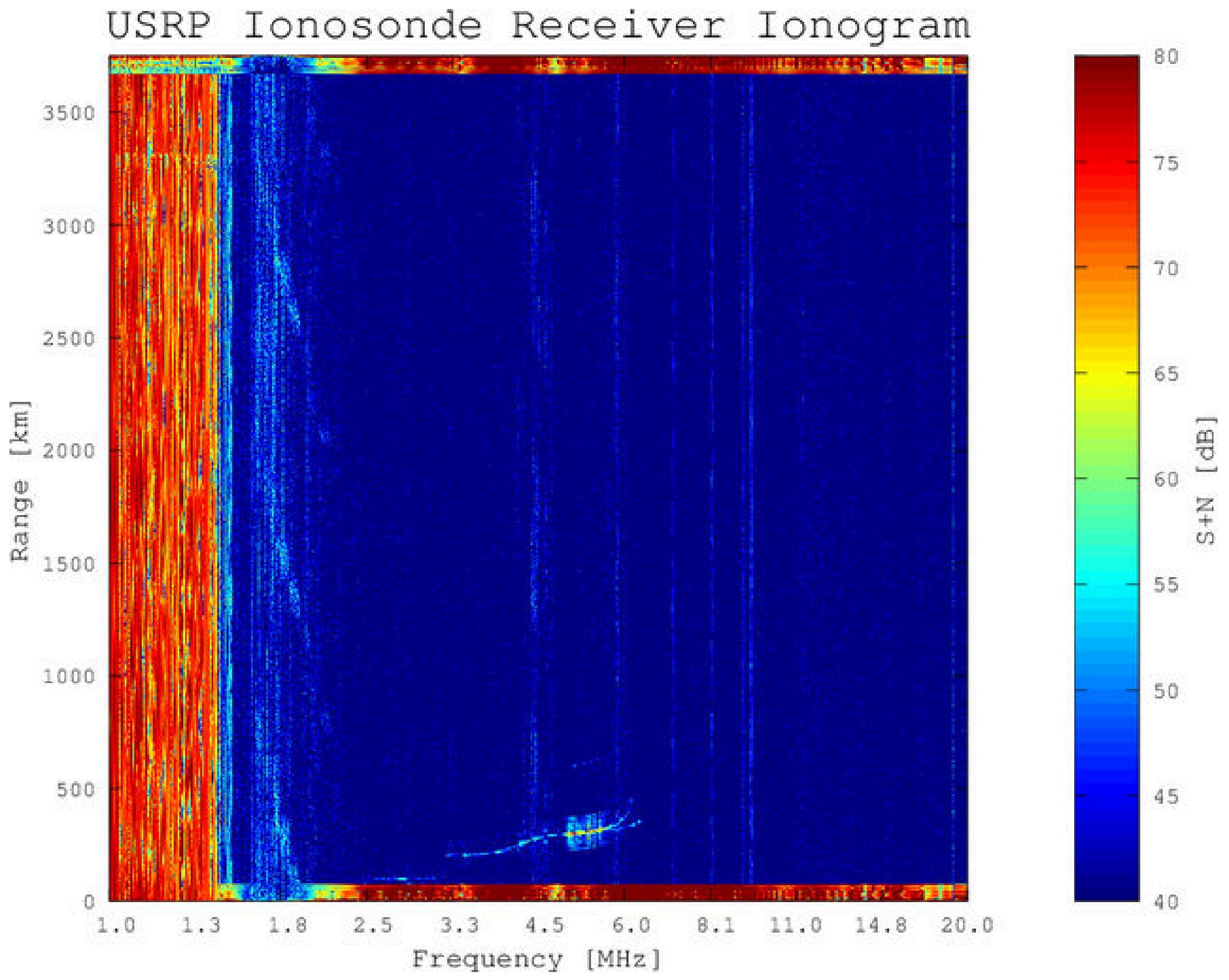


Figure 4.6: Ionogram power map generated by PISCO on 2 July 2013 at 8:15 AST, illustrating phase coding sidelobes.

Fortunately, the most prominent interference resides in the 1.0-to-1.6-MHz AM radio broadcast band, which is typically well below the frequencies at which ionospheric reflections tend to occur. However, several amateur radio, broadcast, fixed, and mobile communication bands are located above the AM radio band [50], and well within the typical reflection frequency range. In most cases, the background interference is not enough to fully mask even medium to low amplitude reflections, although the denser and wider the bandwidth of the interference, the more difficult it could be to discern reflection features.

4.1.2.3 Transmitter Timing Errors

Occasionally, the groundwave pulse from the CADI transmitter does not form the simple sloped line that the PISCO processing software is expecting (see Section 3.1.2.2.4), but rather it appears scattered around, as shown in Figure 4.7. This was confirmed to be due to timing errors within the transmitter, and the erratic and often seemingly random transmitter pulse position (within the IPP) results when there is too heavy of a CPU load on the CADI host computer. The frequency of this type of error occurring has been drastically reduced after a failing process on the CADI host (`rsync`) was identified and corrected; however, it still does occur, though infrequently. Often, the processing software cannot correctly identify the groundwave pulse line in these situations (i.e., a groundwave slope or offset error occurs), but in the case of Figure 4.7 it has, and the ionospheric reflections can be seen from 2.5 MHz to just past 6 MHz.

4.1.2.4 Groundwave Slope Error

On occasion, the line detected by the data processing software does not have the correct slope (i.e., it chooses a line that runs diagonally through the groundwave pulse). When this occurs, the alignment, downconversion, and decoding steps cannot properly operate, as illustrated by Figure 4.8. This error can be easily identified because the groundwave pulse

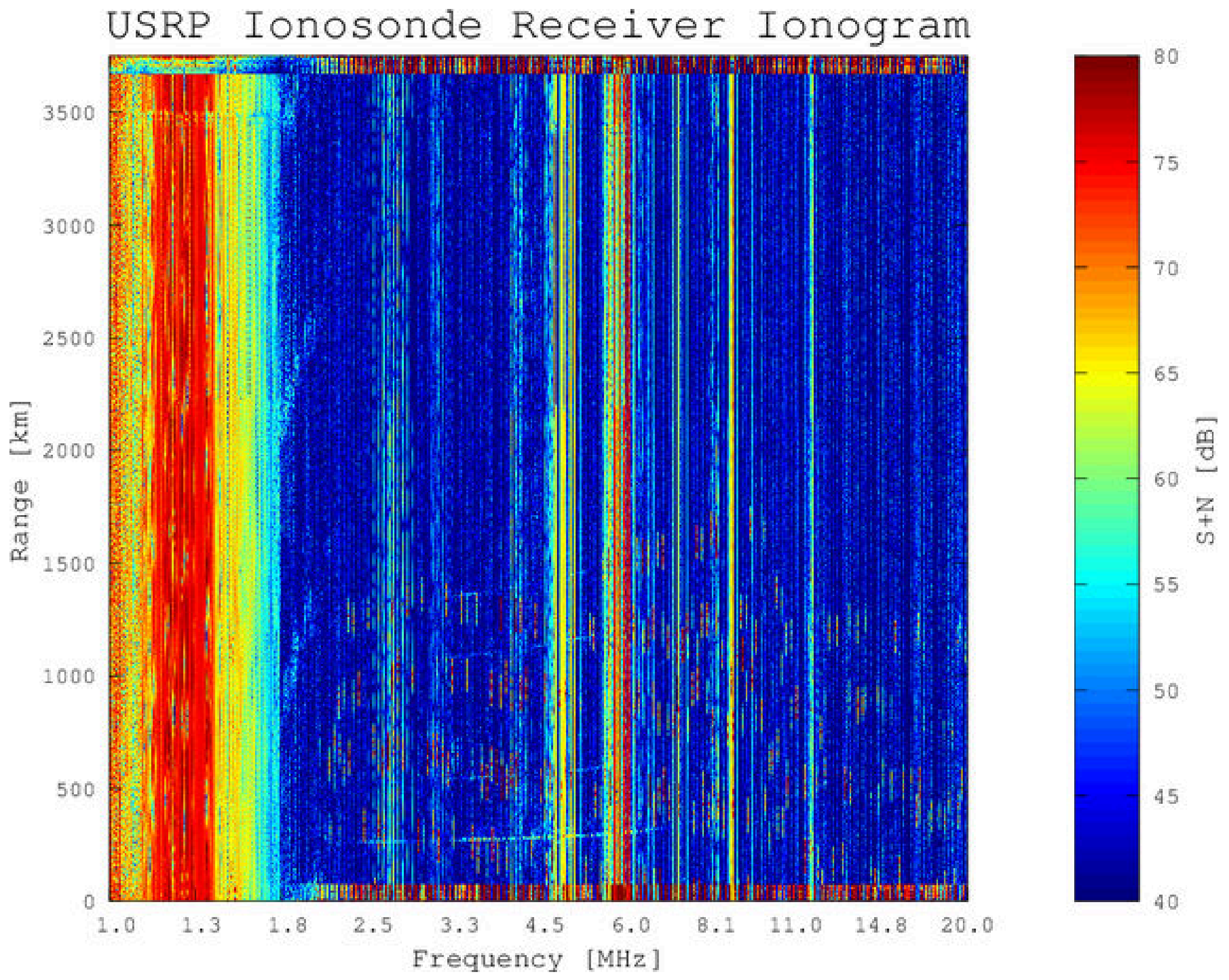


Figure 4.7: Ionogram power map generated by PISCO on 17 June 2013 at 4:15 AST, illustrating the effect of transmitter timing errors on the output plot.

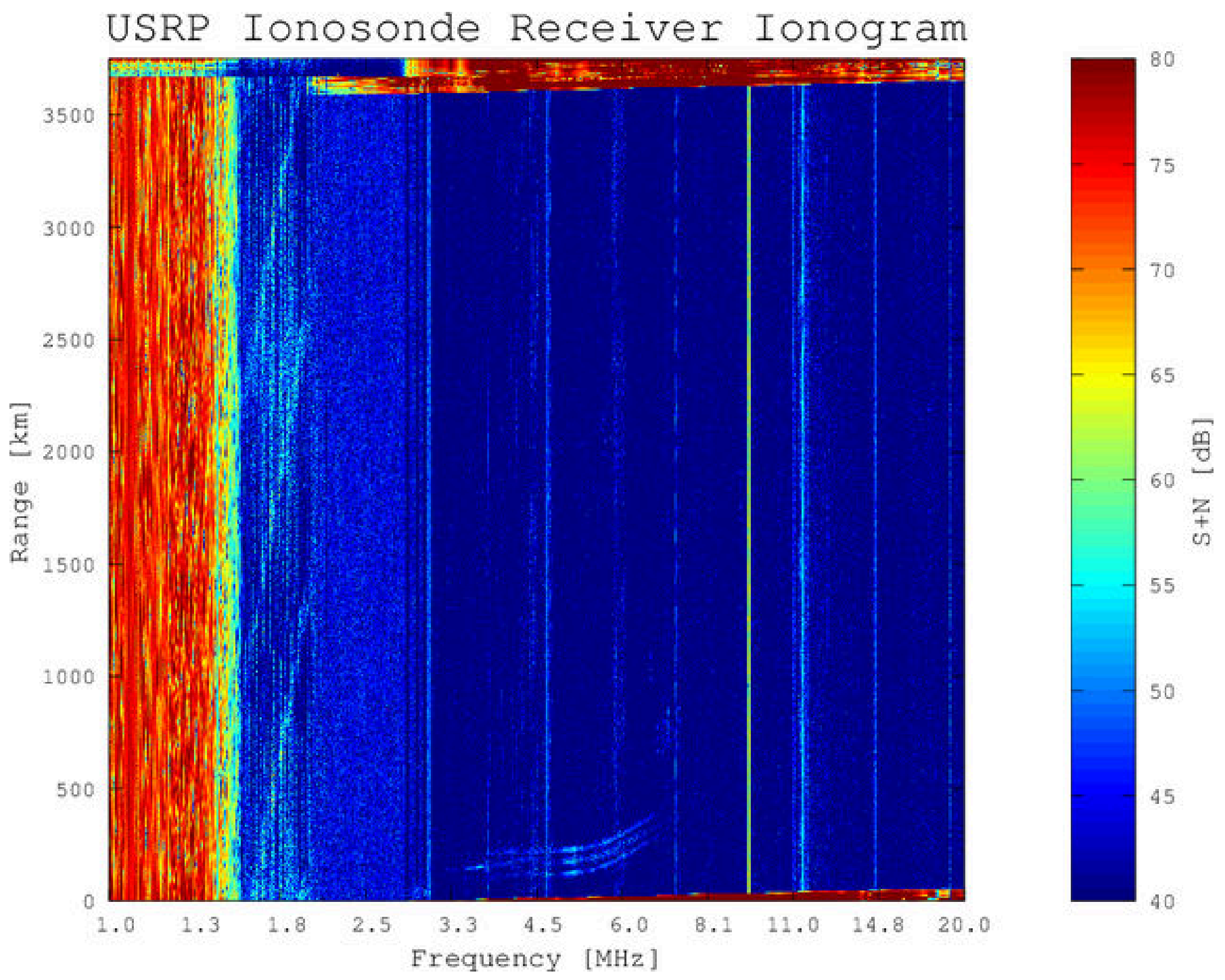


Figure 4.8: Ionogram power map generated by PISCO on 18 June 2013 at 7:45 AST, illustrating the effect of groundwave slope calculation error on the output plot.

seen at the top and bottom of the image has a non-zero slope. The resulting ionospheric reflection that is visible was improperly downconverted and decoded, causing the triplicated appearance.

4.1.2.5 Groundwave Offset Error

Similar to the slope error discussed in Section 4.1.2.4, the groundwave offset error results from an improperly detected groundwave. However, in this case the processing software detects the line on the "wrong" (i.e., bottom) edge of the groundwave pulse instead of the "correct" (i.e.,

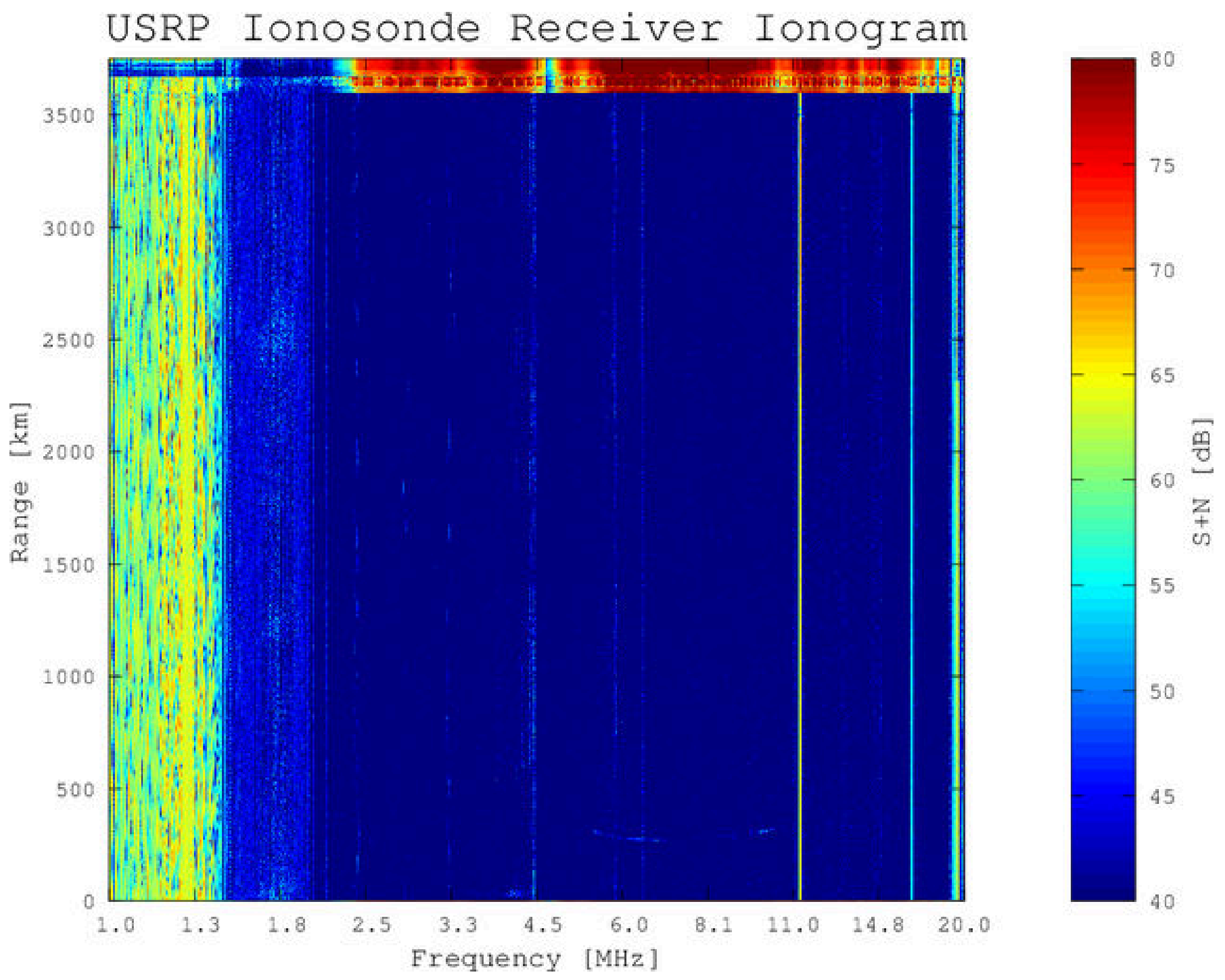


Figure 4.9: Ionogram power map generated by PISCO on 19 June 2013 at 13:45 AST, illustrating the effect of groundwave offset calculation error on the output plot.

top) edge. Since the alignment algorithm relies on the top edge being detected, the alignment is performed improperly, likely resulting in improper downconversion and decoding of the signal. Figure 4.9 shows an ionogram plot resulting from an offset error. The asymmetric split of the groundwave between the top and bottom of the image[1] is indicative of this type of error.

[1]The symmetric split is actually caused by coding sidelobes.

4.1.2.6 Error Statistics

In order to understand the frequency of the different types of errors discussed, a one-week data set (6/21/13 through 6/27/13) was manually analyzed, with each ionogram plot labeled as one of the following:

- *Correct* – No errors, plot can be trusted for correctness
- *Slope Error* – Groundwave slope error occurred, plot should not be trusted
- *Offset Error* – Groundwave offset error occurred, plot should not be trusted
- *TX Error, Correct* – Transmitter timing error occurred, but processing occurred correctly, plot can be trusted with discretion
- *TX Error, Slope Error* – Transmitter timing and groundwave slope errors occurred, plot should not be trusted
- *TX Error, Offset Error* – Transmitter timing and groundwave offset errors occurred, plot should not be trusted

The results from this analysis are tabulated in Table 4.2 (raw counts) and Table 4.3 (percents). The most common type of error was the groundwave offset error, accounting for 6% of the data files processed, closely followed by the groundwave slope error, accounting for 5%. Approximately 89% of the data files were processed correctly, a substantial percentage of the entire data set. It also can be seen that, in many cases, transmitter timing errors result in a secondary processing type of error.

Table 4.2: Error counts by type for PISCO receiver data during the week of 21 June 2013 to 27 June 2013. Note, the host computer experienced a power disruption on 25 June and 27 June, resulting in a missed ionosonde sweep.

Date	# of RTIs	Correct	Slope Error	Offset Error	TX Error, Correct	TX Error, Slope Error	TX Error, Offset Error
6/21/13	96	82	5	8	0	0	1
6/22/13	96	78	5	12	1	0	0
6/23/13	96	83	4	8	0	0	1
6/24/13	96	88	4	3	0	0	1
6/25/13	95	89	3	2	0	0	1
6/26/13	96	86	7	3	0	0	0
6/27/13	95	89	3	3	0	0	0
Totals	670	595	31	39	1	0	4
Average	96	85	4	6	0	0	1
Std. Dev.	0	4	1	4	0	0	1

Table 4.3: Error percents by type for PISCO receiver data during the week of 21 June 2013 to 27 June 2013 (calculated from Table 4.2). Note, the averages sum to >100% due to rounding.

Date	Correct	Slope Error	Offset Error	TX Error, Correct	TX Error, Slope Error	TX Error, Offset Error
6/21/13	85	5	8	0	0	1
6/22/13	81	5	13	1	0	0
6/23/13	86	4	8	0	0	1
6/24/13	92	4	3	0	0	1
6/25/13	94	3	2	0	0	1
6/26/13	90	7	3	0	0	0
6/27/13	94	3	3	0	0	0
Average	89	5	6	0	0	1
Std. Dev.	5	1	4	0	0	1

4.2 PSU All-sky Radar Interferometry System

The first deployment of PARIS is located at the Rock Springs Radio Space Observatory, near The Pennsylvania State University (University Park) campus. The system is still actively under development; however, some preliminary radar experiments have been run (with one receive channel), mainly for testing and verification of the system hardware and software components. Table 4.4 lists the radar parameters used for the radar experiment with PARIS during 5–6 May 2013. This section presents several RTI plots from this experiment. It should be noted that, during this experiment, the receive antennas were configured to connect directly to the first RF limiter on the receive RF front end (bypassing the bias tee), without the use of the preamplifier and bias tee as discussed in Section 3.2.2.1.1. Additionally, the +26-dB preamplifier was moved into the receive RF chain, just prior to the final +11-dB wideband amplifier (see Section 3.2.2.1.3 for details). The system has only been tested with the antenna preamplifier configuration for short durations (although initial results are very promising), so those results are not presented here.

During the PARIS radar experiment run on 5–6 May 2013, approximately 24 hours of data were collected in order to provide an understanding of the system's sensitivity, as well as assess, based on the meteor count, if modifications to the receive RF front end should be made. The subsequent sections present several of the commonly seen targets by PARIS, including airplanes, specular meteors, and non-specular meteors. Additionally, a meteor flux histogram is also presented illustrating the hourly meteor count.

Table 4.4: Radar parameters of PARIS used during the 5–6 May 2013 radar experiment.

Parameter	Value	Units
IPP	2	ms
Baud length	1	µs
Phase Code	28 bit	—
Carrier Frequency	49.8107	MHz
Transmit Power (peak)	∼15	kW

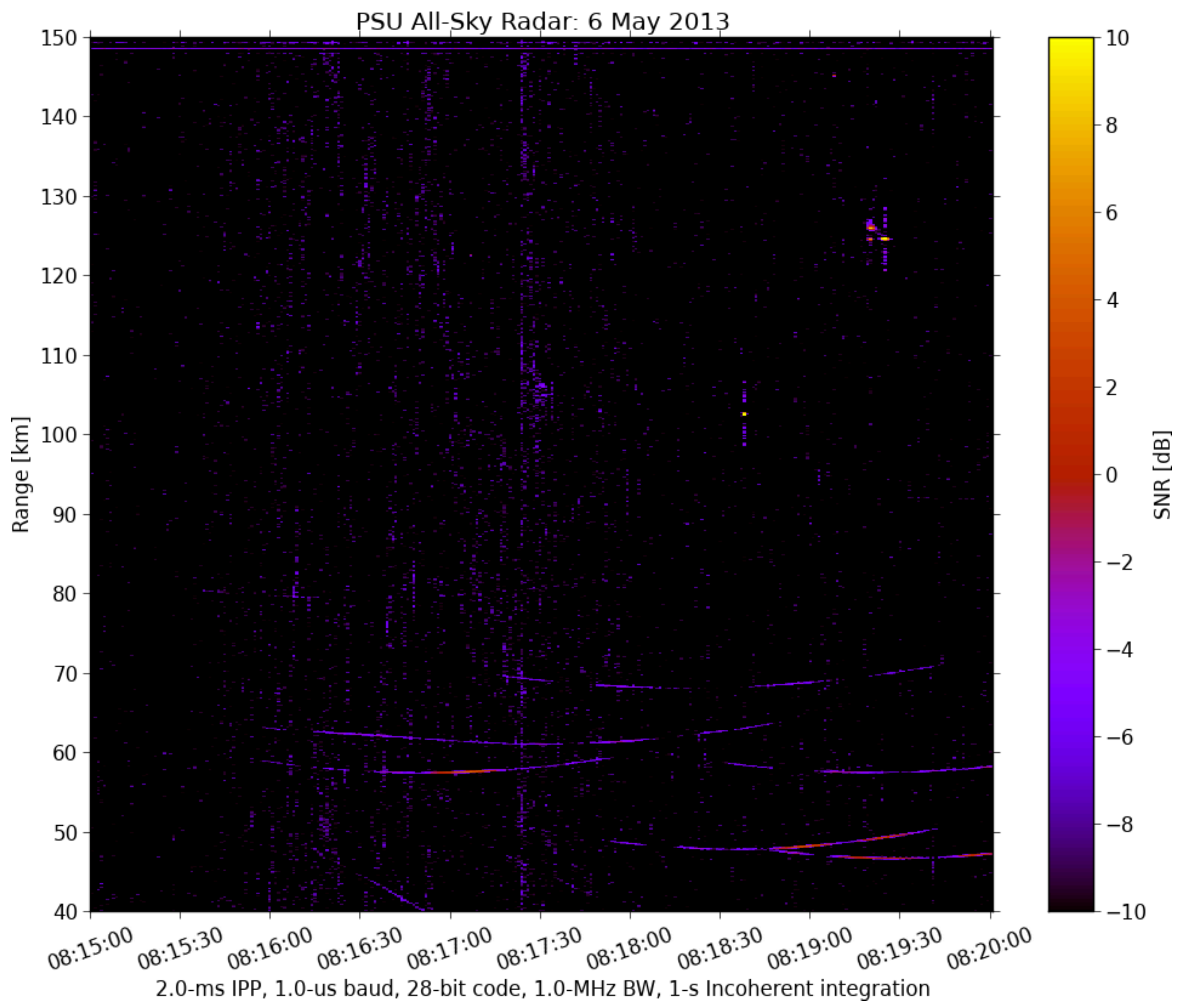


Figure 4.10: RTI from data captured by PARIS between 5–6 May 2013, showing at least six airplanes (bottom traces) and several meteor events (top right, point-like).

4.2.1 Airplanes

As large, metal moving objects in the sky, airplanes can help to provide a good first order test of a radar system. They tend to appear at low ranges (possibly picked up by antenna sidelobes) as thin, straight or slightly curved lines, lasting from tens of seconds up to several minutes in duration. Figure 4.10 shows reflections from at least six airplanes between a range of 40 km to 75 km.

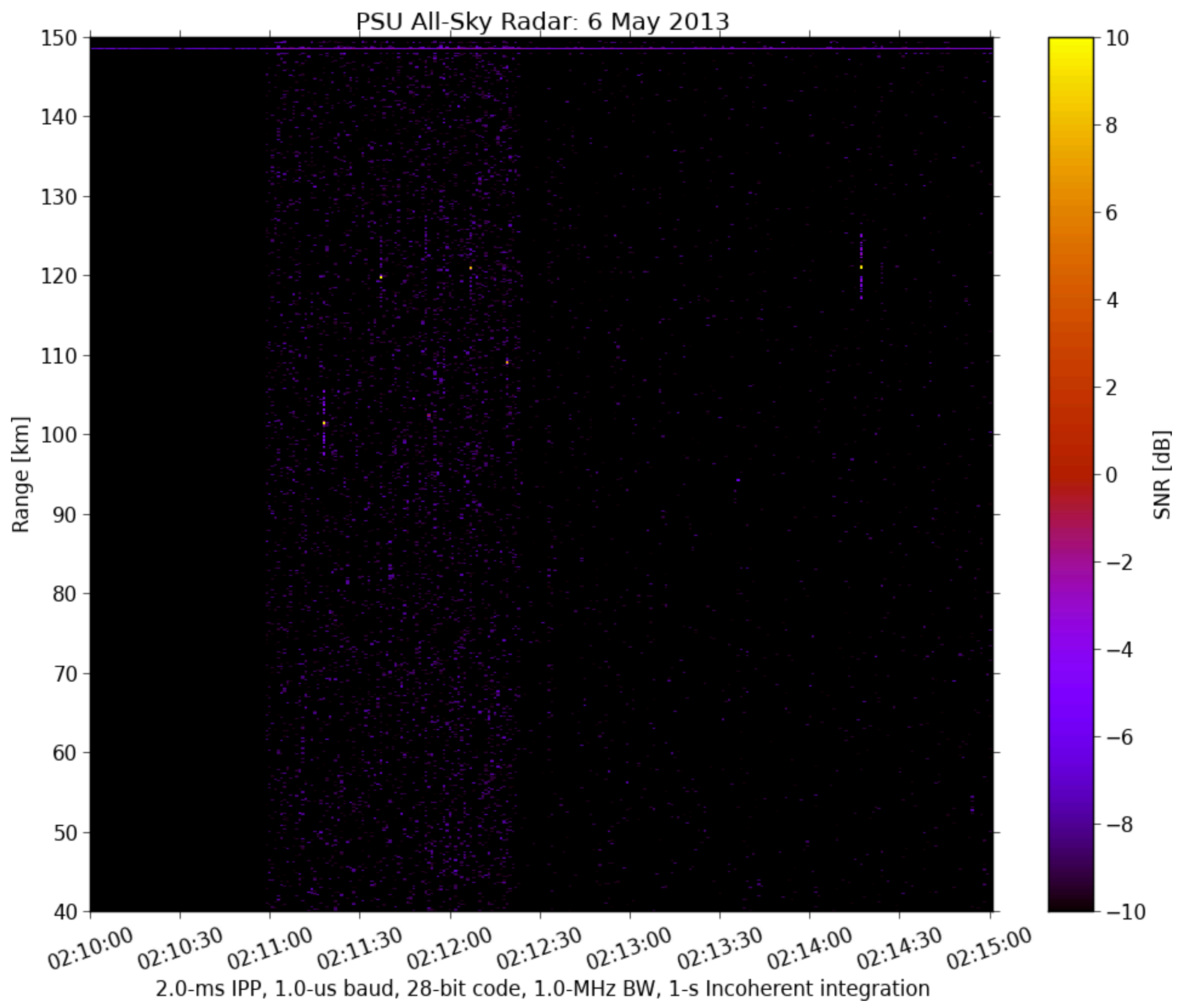


Figure 4.11: RTI from data captured by PARIS between 5–6 May 2013, showing at least six specular meteor events (point-like) between 100 and 130 km.

4.2.2 Specular Meteors

Echoes from the plasma created by specular meteors are what PARIS was designed to observe. These reflections can be seen at all times of the day, are very short in duration (< 1 second), and are usually visible on a timescale of seconds to tens of seconds, depending on time of day, time of year, transmitter power, and receiver sensitivity [37]. Figure 4.11 shows at least six specular meteor trails (yellow and red dots) in the range of 100 to 130 km.

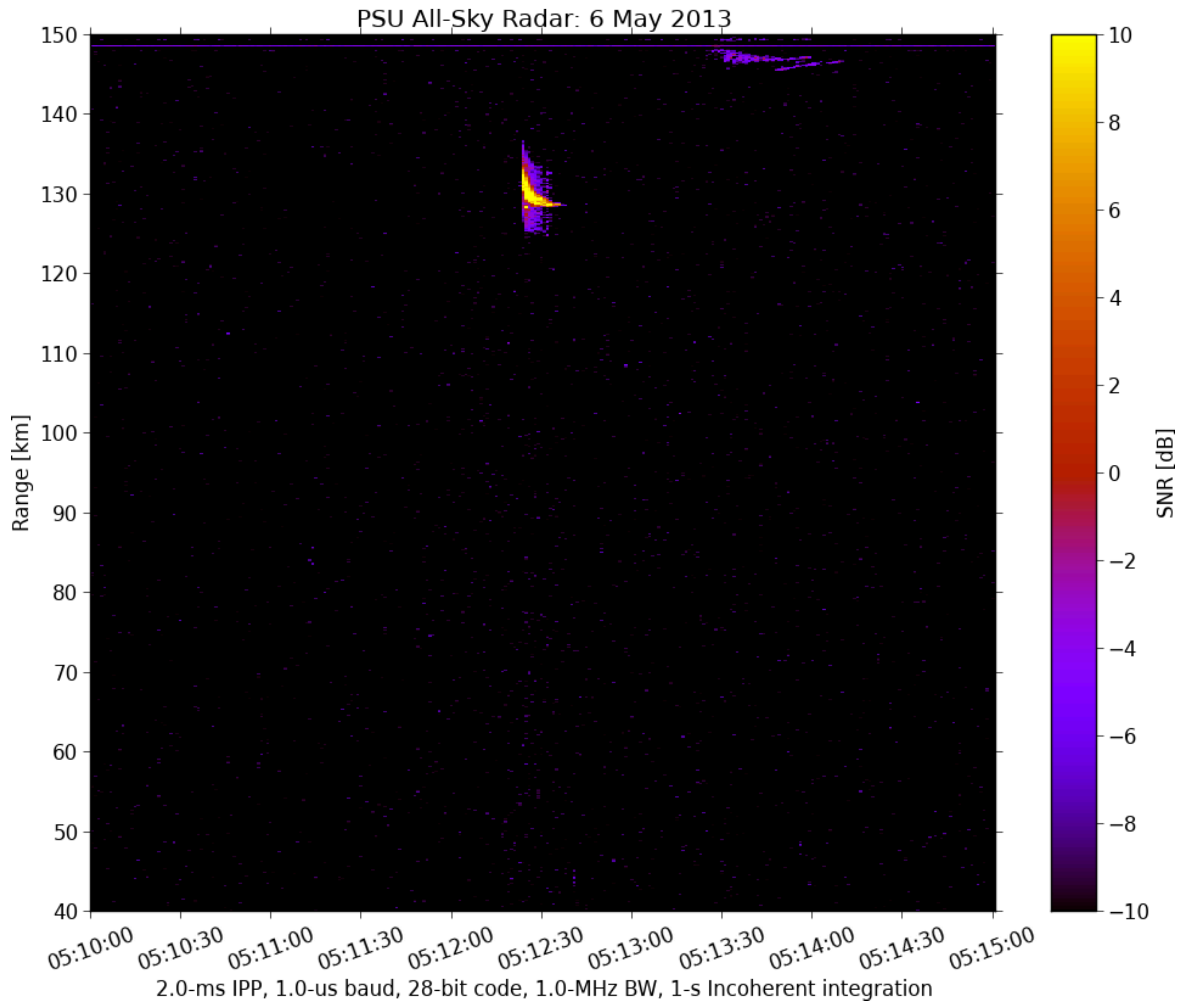


Figure 4.12: RTI from data captured by PARIS between 5–6 May 2013, showing a very strong non-specular meteor event at around 5:12:20 and 130 km in range.

4.2.3 Non-specular Meteors

Unlike specular meteors, plasma trails created by non-specular meteor events can last much longer and do not necessarily appear point-like in RTI plots and are most typically observed with radar trajectories orthogonal to Earth's magnetic field [37]. Figure 4.12 shows a very strong non-specular meteor event detected by PARIS at around 5:12:20 and 130 km in range.

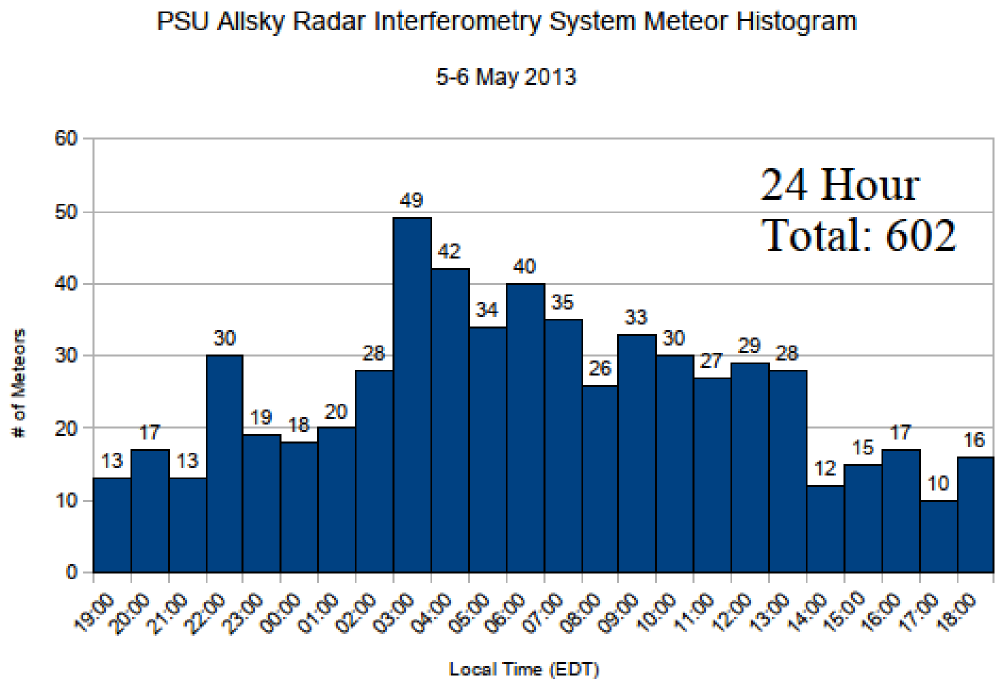


Figure 4.13: Histogram showing meteor flux detected by PARIS between 5 and 6 May 2013.

4.2.4 Meteor Flux

One of the goals of PARIS is to study specular meteors statistically, in terms of count variation diurnally and annually. A histogram with 1-hour bins showing total meteor count over the course of the radar experiment performed by PARIS is shown in Figure 4.13. As expected, the histogram shows a general increase in the number of meteors during the morning hours before and during sunrise. Although not shown here, preliminary tests have shown around a 40% increase in meteor count during the early morning hours with the antenna preamplifier configuration discussed in Section 3.2.2.1.1.

4.3 Cognitive Interferometry Radar Imager

As discussed in Section 3.3, the first deployment of CIRI (CIRI@PSU) is located at the Rock Springs Radio Space Observatory, along with PARIS. The system ran nearly continuously from March 2013 through June 2013, and a very large data set is available for viewing (RTI images) or further processing. Table 4.5 lists the radar parameters of CIRI@PSU during this time frame. The following sections present a few of the typical and more interesting results from data obtained by CIRI@PSU, including specular and non-specular meteor targets, and results from a power sweep experiment run between 28 April and 4 May 2013.

Table 4.5: Radar parameters of CIRI used from March 2013 through June 2013.

Parameter	Value	Units
IPP	4	ms
Baud length	5	µs
Phase Code	28 bit	—
Carrier Frequency	49.800	MHz
Transmit Power (peak)	30	kW
Pulse Shape	Square	—

4.3.1 Specular Meteors

The short-duration specular meteor trails are detected by CIRI@PSU in quantities of hundreds daily, providing many data points for the mass and velocity statistics that CIRI was designed in part to study. Figure 4.14 shows reflections from nine specular meteor events over the course of 30 seconds, ranging from 140 km to 350 km,[2] and Figure 4.15 shows six additional specular meteors ranging from 150 km to 400 km. Note, the vertical strips surrounding several of the meteor echos in Figure 4.14 are due to sidelobes from phase coding used for pulse compression radar, and do not represent physical ionospheric phenomena.

[2]Because of the low elevation angle of the antenna arrays, these ranges correspond to approximately 40 to 100 km in altitude. The low end of this range is very low to observe meteors, so several of these meteors are likely range aliased.

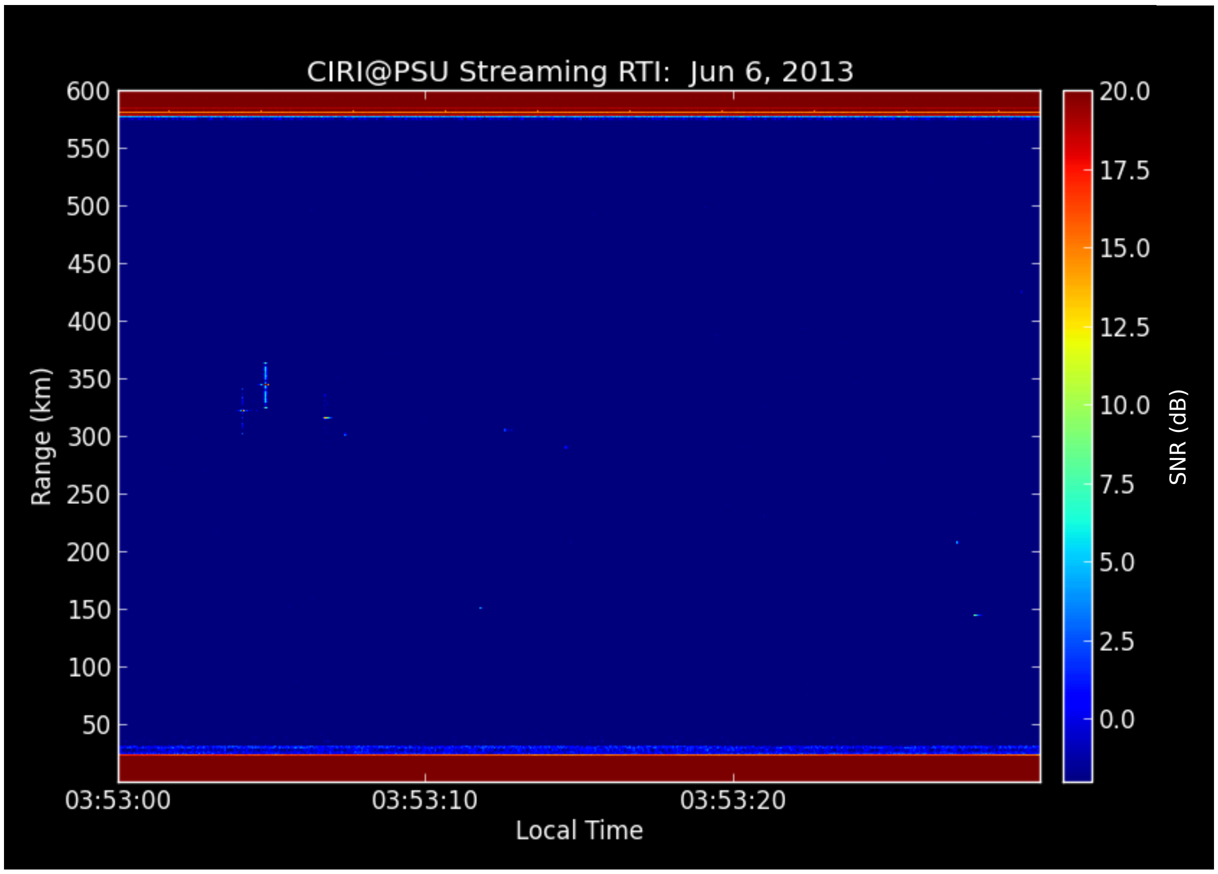


Figure 4.14: RTI plot generated by CIRI on 6 June 2013, showing nine specular meteor events.

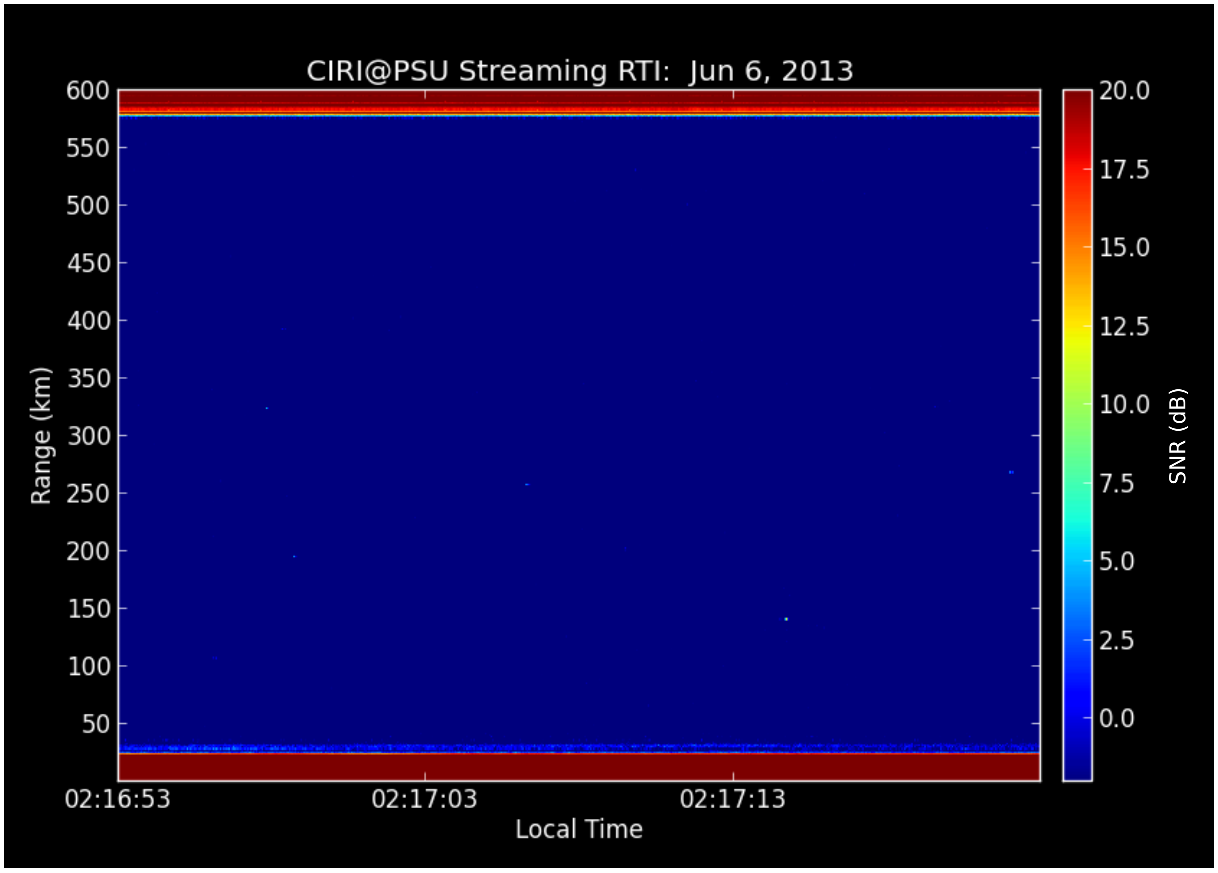


Figure 4.15: RTI plot generated by CIRI on 6 June 2013, showing six specular meteor events.

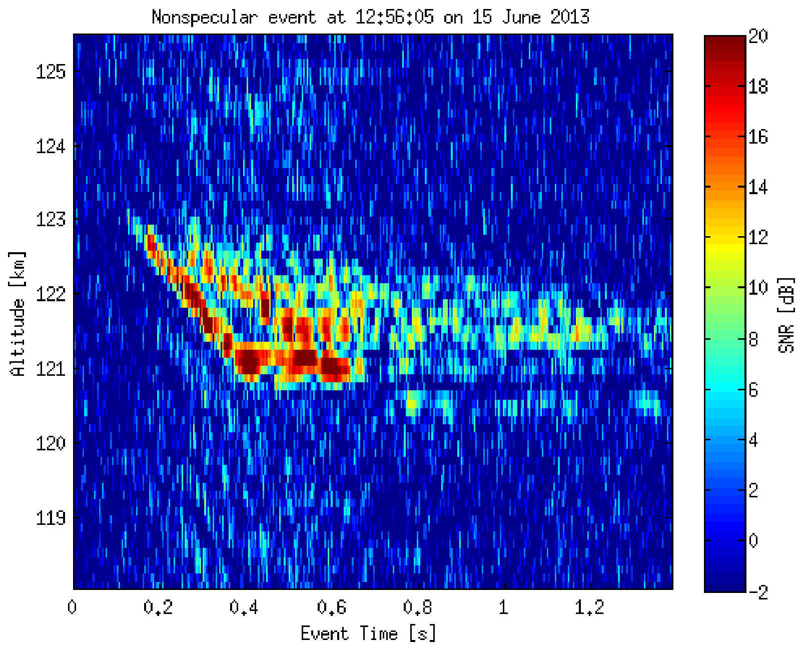


Figure 4.16: Close-up of meteor-head and nonspecular meteor detected by CIRI on 15 June 2013 at 12:56:05 EDT.

4.3.2 Non-specular Meteors

In addition to specular meteors, CIRI is also capable of detecting the larger and longer duration non-specular events, as well as meteor-head echos. Figure 4.16 shows a close-up view of a meteor-head and non-specular meteor event from 15 June 2013. This image was plotted from the event data saved by Sauron using the meteor detection/event windowing data compression technique, discussed in Section 3.3.2.2.3. Figures 4.17, 4.18, and 4.19 also show several interesting reflections from meteor-heads and non-specular meteor trails, all captured on 6 June 2013.

4.3.3 Power Sweep Experiment

In an attempt to test the different power levels of the transmitter, as well as provide preliminary proof-of-concept meteor flux results for future, more extensive experiments, a power

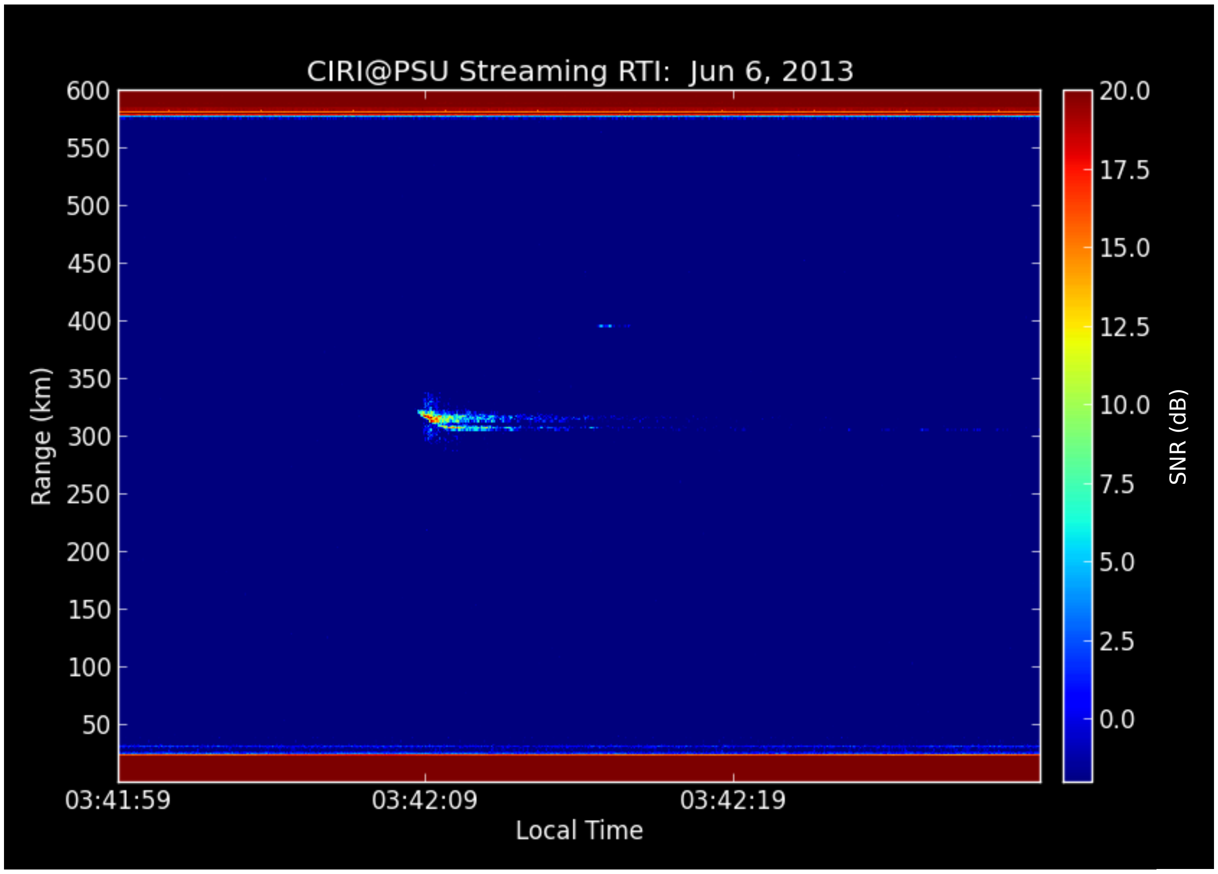


Figure 4.17: RTI plot generated by CIRI on 6 June 2013, showing a meteor-head and non-specular meteor at 3:42:08 EDT.

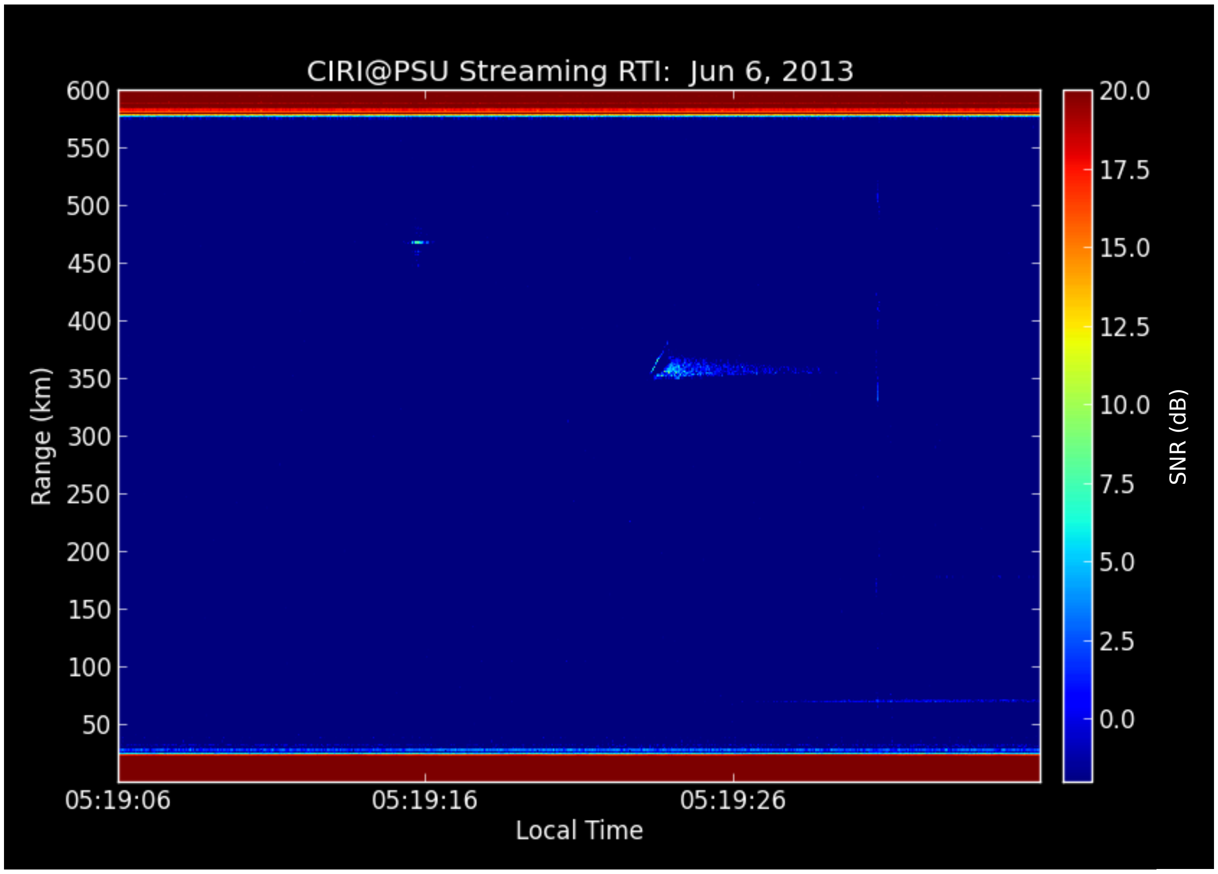


Figure 4.18: RTI plot generated by CIRI on 6 June 2013, showing a meteor-head and non-specular meteor at 5:19:22 EDT.

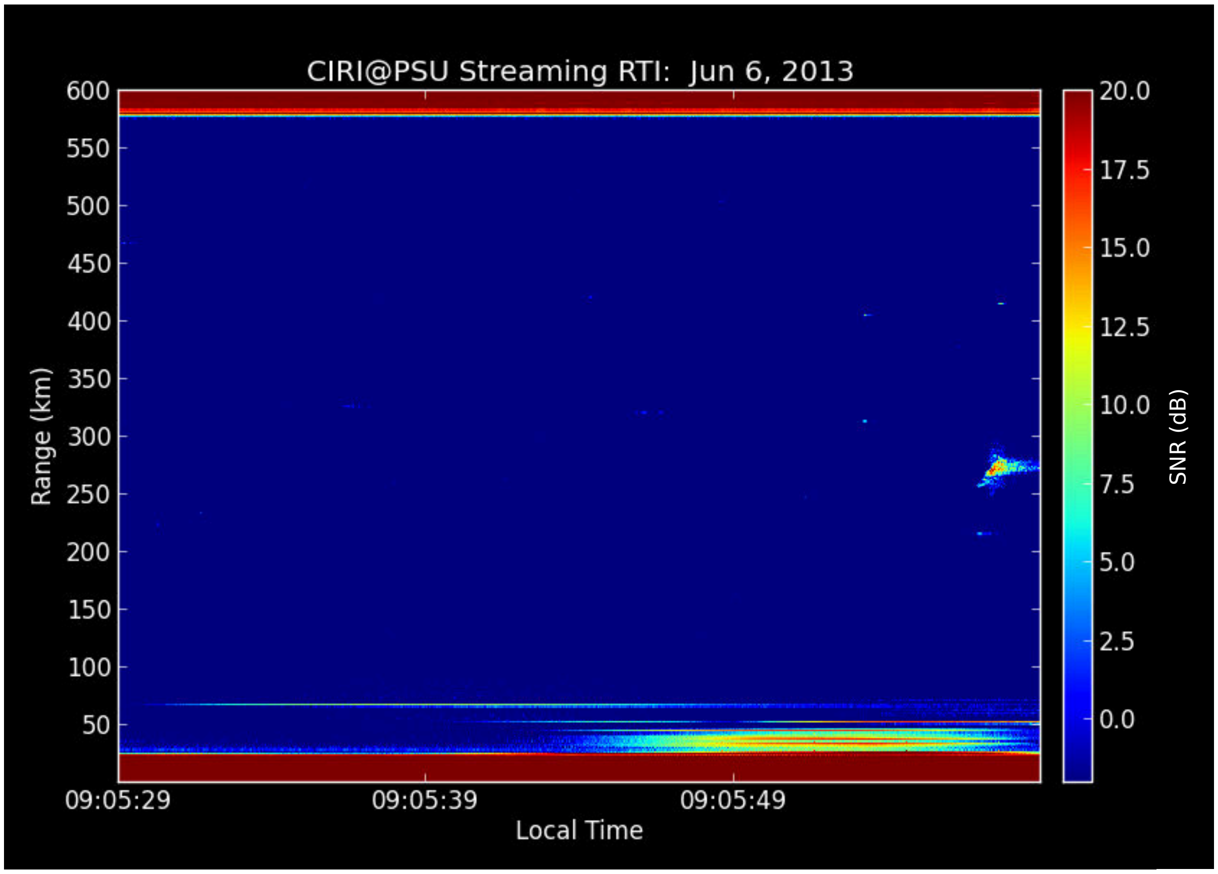


Figure 4.19: RTI plot generated by CIRI on 6 June 2013, showing several specular meteor trails, and a meteor-head and non-specular meteor at 9:05:56 EDT.

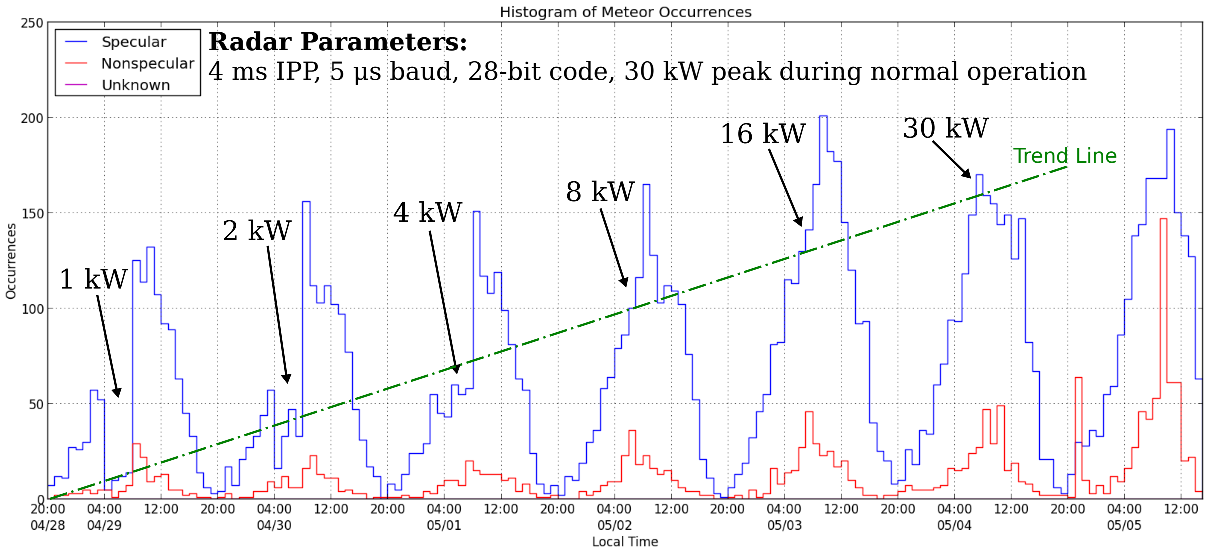


Figure 4.20: Meteor fluxes observed by CIRI during a week-long power sweep experiment from 4 to 8 am (EST) each day (as denoted above).

sweep experiment was performed from 28 April through 4 May 2013. During this experiment, the peak transmitter power was progressively increased from 1 kW to 30 kW over the course of the week, but only between the hours of 4 to 8 am each day. During the rest of the day, the system resumed normal operation (30-kW peak power). A histogram of the hourly meteor fluxes during this experiment are shown in Figure 4.20. As expected, the number of meteor events detected increased with increasing power levels. The relationship between meteor count and transmitted power appears to be potentially logarithmic, although the unknown flux variation from day to day makes it difficult to assess error bar size. The experiment will need to be run multiple times with better knowledge of flux variation to draw any significant conclusions; however, these preliminary results certainly appear promising.

Chapter 5

Conclusions

A wide spectrum of material relating to the three software-defined ionospheric sensors discussed has been examined up to this point. This chapter briefly recaps the content presented, and provides direction for continued and future efforts on unresolved issues and new features for each of these projects.

5.1 Summary

In this thesis, a variety of topics have been discussed, beginning with the foundations. In Chapter 1, some of the the scientific and engineering motivations behind the development of these different ionospheric sensor systems were examined. Additionally, because the complexities involved with each project could fill volumes, the extents of the author's efforts and of the information provided in this document was limited to a reasonable scope.

Despite limiting the scope of these projects, there is still a large amount of information from multiple domains with which the reader must be familiar in order to grasp the full essence of the project designs. Chapter 2 attempted to shed some light in many of the different science and engineering disciplines and topics covered by these projects, placing readers of all backgrounds on common ground. The motivational ionospheric science topics were briefly discussed, followed by an overview of several classes of instruments used to make observations in this field. Then, the emerging topic of software-defined radio was examined, exploring both the fundamental philosophy and principles behind its operation. The USRP, as a software-defined radio platform, was presented and some of its basic features, especially those relevant to the projects in this thesis, were reviewed. Finally, the science and technology topics were united through a brief survey of previous efforts in the software-defined ionospheric sensor arena.

Chapter 3 sequentially presented the design of the PISCO, PARIS, and CIRI projects to a moderate level of detail. Each system was overviewed at a high level for context, both in operation and in the science results expected to be observed, followed by an organized breakdown of the various hardware and software components that comprise each sensor system. PISCO was presented as a software-defined ionosonde receiver with minimal hardware components and a number of external challenges to overcome in software. On the larger scale, PARIS was presented as part of an ongoing effort to construct a traditional-style meteor radar system, with software-defined elements. Also on the larger scale, the CIRI system was detailed, a radar system utilizing some different aspects of software-defined radio than PARIS.

Preliminary testing of each of the systems has validated at least the concept of software-defined ionospheric sensors, if not already provided important scientific results. During PISCO's operation, a number of ionosonde-familiar phenomena have been observed, including hop reflections, multiple F-region layers, Spread-F, the ordinary and extraordinary mode split, and Sporadic-E. In their initial deployments at the Rock Springs Radio Space Observatory, both PARIS and CIRI@PSU have successfully detected reflections from specular and non-specular meteor trails. These results demonstrate the validity of applying software-defined radio technologies to ionospheric science and will hopefully motivate and inspire new, innovative solutions both within field and outside of this field of study.

5.2 Future Work

Although each of the three systems presented have shown positive preliminary results, none of the systems are completely ready for long-term, reliable, and user-friendly operation (despite some parts being very close). The following sections provide a brief overview of some of the design work still ahead for PISCO, PARIS, and CIRI. Additionally, some potential, alternate applications for these systems are suggested and discussed.

5.2.1 PSU Ionospheric Sounder for Chirp Observations

As illustrated by the results presented in Section 4.1, the PISCO receiver has been fairly successful in observing several ionospheric layers; however, there are still a few issues with the system that should be resolved before the system is ready to provide reliable scientific results. The following sections discuss a potential solution to the groundwave offset and slope errors that occasionally affect the PISCO receiver, as well as present several ideas for future efforts with this system.

5.2.1.1 More Robust Groundwave Detection

Several of the issues discussed in Section 4.1.2 (i.e., groundwave slope error and groundwave offset error) potentially could be solved by the implementation of a more robust groundwave detection algorithm. Currently, the PISCO data processing program identifies the strongest line candidate within a given range of angles in an attempt to find the groundwave pulse. The software makes the assumption that this line lies along a specific edge of the groundwave pulse, which, according to the error statistics presented in Section 4.1.2.6, works fairly well for basic operation of the system. However, the software processing is essentially operating open loop, without the ability to identify that the groundwave has not been found correctly and recover.

A smarter algorithm might attempt to identify both edges of the groundwave pulse, instead of just one. The "thickness" of the pulse (in range) could then be checked to be relatively constant (for the duration of the pulse), matching an expected value based on the radar parameters. If not, further attempts could be made to find the groundwave by even more robust image processing algorithms, based on the *a priori* knowledge of the shape of the groundwave pulse.

5.2.1.2 Fixed-Frequency Meteor Radar Mode

While both the PISCO receiver and the CADI system were designed as and intended to operate in an ionosonde configuration, some preliminary work has been started on using the PISCO/CADI system as a fixed-frequency radar system to search for meteor reflections. Because the carrier frequency is software programmable (to some extent) on both systems, only a few minor changes are necessary for the two systems to operate in a basic fixed-frequency mode, although current implementations of both the CADI software and PISCO software prohibit continuous radar operation (like PARIS and CIRI). Instead, radar operation is constrained to one-minute intervals, scheduled as often as every other minute.[1] Results so far have been inconclusive as to the feasibility of such a system, and further analysis and testing are required.

5.2.1.3 Ionosonde Transmitter

The PISCO receiver presented only represents one half of a complete ionosonde system, and thus, an appropriate future effort would be to design and construct the PISCO transmitter. The flexibility of the USRP platform enables, and even invites, development of a transmitter using the same platform, even the same physical device.[2] The automatic frequency sweeping

[1]Even this requires offline data processing on PISCO, as the processing software cannot keep up with such a high data rate.

[2]The USRP1 platform supports two transmit and two receive channels simultaneously.

capability added for the receiver design is also applicable for the transmitter, which needs to generate the RF carrier for transmission. An added benefit of using a single device for both receiver and transmitter is perfect frequency synchronization between the two, eliminating the need for the frequency offset detection and downconversion processing step (step 5 described in Section 3.1.2.2.4). Additionally, if using an external clock source that also provides a clock to the radar controller (or other device that provides radar timing signals), the reconstructive groundwave synchronization can be drastically simplified, as it should have a slope of zero.

5.2.2 PSU All-sky Radar Interferometry System

As a system that is still actively under development, there are a few features that have not been fully implemented, affecting both the hardware and software of PARIS. The following sections discuss these features and describe the potential steps towards realizing them.

5.2.2.1 Five-channel Receive RF Front End

At the time of writing, only one channel of the receive RF front end has been constructed and tested, mainly due to the frequency change issue discussed in Section 5.2.2.2. Before interferometry operation can occur, all five RF signal chains will need to be constructed, tested, and integrated into the PARIS receive segment.

5.2.2.2 Operating Frequency Change

The wideband spectral content that results from narrow, square pulsed, and phase-coded radar signals is problematic when operating with a carrier close to frequency bands of other services, as the transmitted signals can radiate unwanted spectral content and cause interference within other bands. In fact, recent operations of both PARIS and CIRI have caused interference in the 6-m amateur radio band (whose lower end starts at 50.000 MHz), which led to a congress and discussions between ASPIRL researchers and active amateur radio users. The decision was made to move the carrier frequency of PARIS away from this band to 40.2 MHz; however, not without consequences: retuning of the transmitter, purchase/construction of new antennas, and modification of the receive RF front end will all be necessary for operation at the new frequency.

5.2.2.3 Automated Meteor Detection

The meteor flux counts shown in Figure 4.13 were performed manually, cataloging the time of each of the events in a spreadsheet and totaling the results. Not only is this a tedious process for large data sets, but it introduces the potential for biases and error, especially when multiple or poorly trained people catalog independently. The problem will be only escalated when PARIS begins operating continuously for months at a time.

In CIRI, this issue is solved by using the automated meteor detection and classification routines built into the Sauron software; however, PARIS/GnuRadar does not have an equivalent functionality. Although the two systems use different data file types, it may still be possible to modify and utilize the meteor detection and classification portion of Sauron with GnuRadar. If this is the approach taken, it would likely need to work from the data files written to disk and already closed out by GnuRadar,[3] introducing a significant delay between the time of data collection and detected meteors within that data. If a separate detection and classification algorithm was written (in C/C++) it could potentially be injected into GnuRadar's `ConsumerThread`, processing data from shared memory before writing to the HDF5 file on disk.

[3]HDF5 files cannot easily be simultaneously written to and read from [51].

5.2.2.4 Synchronization and Cooperation with CIRI@PSU

In order to operate the Rock Springs Radio Space Observatory to its full potential, both PARIS and CIRI@PSU will need to be operated simultaneously, a non-trivial task that will require careful planning and implementation. The large distance between the two systems' operating facilities (∼125 m) and site layout makes hard-wired synchronization between the two systems difficult and impractical. However, the use of GPS timing appears to provide a viable synchronization solution. Both systems use Novatech oscillators as the master system clock, which can be synchronized to an external input clock. This external clock signal could be supplied by a GPS-disciplined oscillator (GPSDO), such as the work presented in [52], which generates a standard clock signal that is phase- or frequency-locked to the one pulse-per-second (1PPS) provided by GPS receivers. Additionally, the same radar controller is used by both PARIS and CIRI, and it features a PPS input signal for synchronization, which could be provided by the same GPS receiver's output PPS. There are undoubtedly other details to consider; however, this basic approach, if carefully implemented, should be successful.

5.2.3 Cognitive Interferometry Radar Imager

Although the CIRI@PSU system has been operating continuously for several months, with minimum operator interaction, there are still a few tasks that need to be completed before the system is ready for basic science operations. The following sections provide a brief overview of these tasks and suggestions on how to complete them. Additionally, a brief preview of some of the "cognitive" features are presented.

5.2.3.1 Antenna Array Beam Pattern

Although the COCO antenna arrays were constructed as closely as possible to the design shown in Section 3.3.2.1.1, there invariably are differences between the design and the constructed arrays, namely cable droop between the support poles and mismatched array heights. It is unknown how much the shape and direction of the arrays' beam pattern is distorted from these differences, and this could have a large impact on experimental results if the arrays are misaligned.

Because the setup is so large (∼100 m in length) beam pattern measurement is challenging; however, one method has been discussed that should provide a first-order of beam pattern of the arrays. This method involves the use of a mobile receiver in a car driving approximately perpendicular to and within the expected main lobe of the antenna arrays. The receiver would listen to the transmitted pulse from CIRI@PSU, recording signal strength and GPS location as the car drives through the main lobe. Using the recorded data, an approximate 2-D beam pattern could be estimated, ensuring that the arrays are indeed pointing towards magnetic north. Because the antenna arrays are expected to have a fairly large beamwidth in the elevation direction, the mobile receiver should not have difficulty picking up the transmitted pulsed RF.

5.2.3.2 Pulse Shaping

As mentioned in Section 5.2.2.2, interference issues with the amateur radio community (caused by CIRI@PSU) have led to discussions and examination of the radar signals' spectra. Previously operated with a short baud length (5 µs), long code (28-bit), and a square pulse envelope severely disrupted amateur radio operations in the 6-m band. The very low end of this band is used exclusively by the amateur radio community for transmitting very long distances and recovering very weak signals.

After working on-site with several amateur radio operators, along with others at remote stations, it was determined that, due to the close proximity of the 49.8-MHz carrier to the 6-m radio band, only the Gaussian shaped pulse may be used with pulse lengths of 10 µs or longer, as long as any phase-coding used for pulse compression does not widen the spectrum beyond 50.000 MHz. These radar signal restrictions will affect data quality for future experiments, increasing the minimum discernible range (i.e., range resolution) and significantly lowering average power output of the transmitter.

5.2.3.3 Cognitive Functionality

Even though most of the components of the system are software-configurable, CIRI@PSU still operates as a "dumb" radar system, with no "cognitive" functionalities. After the system has proven itself with valuable science data in the dumb radar mode, development attention will be focused on adding functionality to reconfigure the radar system automatically and on the fly. Upon detecting and identifying different ionospheric layers and events, the system will automatically reconfigure itself for the best radar parameters for that target (within predefined limited sets) and hone in on specific details about the target.

The Genesis transmitter has many different features designed to enable cognitive radar applications. These include programmable pulse shaping (e.g., ramp, triangle, etc. in addition to square and Gaussian) and switching between four user-programmable pulse configurations on an IPP by IPP basis, suggesting alternation between multiple pulse configurations each IPP. These features could potentially result in a broader scope of data on a particular event, although much study and analysis will need to accompany these advanced experiments.

5.3 Final Remarks

Software-defined radio is an exciting technology that is constantly finding applications in new fields as engineers and scientists around the world continue to innovate. With the amount of big picture scientific questions about the ionosphere still unanswered, the low cost and reconfigurability of software-defined radio platforms (such as the USRP) provide an excellent tool to build and deploy widespread sensor networks, enabling previously impractical (or even impossible) experiments to be performed. These systems can make use of emerging signal processing algorithms to analyze and piece together data sets in ways never done before. It is hoped that the utilization of software-defined radio continues to permeate throughout the ionospheric sciences community, as well as throughout other scientific communities to better explain the unsolved mysteries of the Universe.

References

[1] M. Greenman. “An Introduction to HF Propagation and the Ionosphere.” (1999). `http://www.qsl.net/zl1bpu/IONO/iono101.htm`

[2] “Space Weather and the Ionosphere.” `http://www2.naic.edu/aogeo/frames/weather_info.htm`

[3] R. Seal. *VHF Software Defined Radar for Atmospheric Research at The Pennsylvania State University.* (2013). M.S. Thesis. The Pennsylvania State University.

[4] R. Seal and J. Urbina. *VHF Radar Design Using Software Defined Radio Receiver Platform.* (2013).

[5] R. D. Hunsucker. (1991). *Radio Techniques for Probing the Terrestrial Ionosphere.* Springer-Verlag, Berlin.

[6] C. Davis. (1996). “Interpreting an ionogram.” `http://www.wdc.rl.ac.uk/ionosondes/ionogram_interpretation.html`

[7] W. R. Piggott and K. Rawer. (1961). *URSI Handbook of Ionogram Interpretation and Reduction.* Elsevier Publishing Company, Amsterdam.

[8] J. K. Shi, *et al.* (2011). “Properties of Spread-F in High and Low Latitude Ionospheres.” *PIERS Proceedings*, Marrakesh.

[9] P. Colestock, S. Close, and J. Zinn. “Theoretical and Observational Studies of Meteor Interactions with the Ionosphere.” Los Alamos National Laboratory, Los Alamos.

[10] L. Dyrud, *et al.* (2007). “Plasma and Electromagnetic Simulations of Meteor Head Echo Radar Reflections.” *Earth, Moon, and Planets*, Vol. 102.

[11] M. Tsutsumi, *et al.* (1999). “Meteor Observations with an MF Radar.” *Earth Planets Space*, Vol. 51.

[12] S. Palo. (2007) “Meteors, Meteor Radar and Mesospheric Winds.” CEDAR Workshop, Santa Fe. `http://sisko.colorado.edu/palo/CEDAR-2007/PALO_CEDAR%20meteor%20radar.pdf`

[13] C. Wolff. “Radar Basics – Pulse Compression.” `http://www.radartutorial.eu/08.transmitters/tx17.en.html`

[14] V. C. Ramasami. (2006). “Principle of the Pulse Compression Radar.” `https://www.cresis.ku.edu/~rvc/documents/pulsecomp.pdf`

[15] K. Nozaki. “Application of FM/CW Techniques to Ionosondes.” `http://www.ursi.org/files/CommissionWebsites/INAG/uag-104/text/nozaki1.html`

[16] S. Bilén. (2009). “SDR: The Future of Radio.” Class Lecture, EE 497E, The Pennsylvania State University.

[17] C. R. Johnson and W. A. Sethares. (2003). *Telecommunication Breakdown: Concepts of Communication Transmitted via Software-Defined Radio*. Prentice Hall, Upper Saddle River.

[18] (2013). “Ettus Research.” Ettus Research. `http://ettus.com/`

[19] S. Franke. (2011). Unpublished Manuscript. University of Illinois.

[20] “GNU Radio.” `http://gnuradio.org/redmine/projects/gnuradio/wiki`

[21] (2009). “SKiYMET Meteor Radar.” Genesis Software. `http://www.gsoft.com.au/productsandservices/skiymet`

[22] J. Vierinen. “GNU Chirp Sounder.” Sodankylä Geophysical Observatory. `http://www.sgo.fi/~j/gnu_chirp_sounder/`

[23] F. Lind. “OpenRadar - The Open Radar Initiative.” `http://www.openradar.org/`

[24] “DX Engineering Active Horizontal Receive Antennas DXE-ARAH3-1P.” DX Engineering. `http://www.dxengineering.com/parts/dxe-arah3-1p`

[25] “Universal Serial Bus Specification Revision 2.0.” USB Implementers Forum, Inc. `http://www.usb.org/developers/docs/usb_20_070113.zip`

[26] “Hardware Setup Notes: External Clock Modification.” `http://files.ettus.com/uhd_docs/manual/html/usrp1.html`

[27] “FPGA.” GNU Radio. `http://gnuradio.org/redmine/projects/gnuradio/wiki/UsrpFAQIntroFPGA`

[28] "Bench Top Direct Digital Synthesized (DDS) Signal Generators." Novatech Instruments, Inc. http://www.novatechsales.com/Bench-Signal-Generator.html

[29] GNU Radio Source Code. http://gnuradio.org/redmine/projects/gnuradio/repository/revisions/b5709abe52928f0b701cd5f5eedefc2c1665123e/entry/usrp/host/lib/usrp_standard.cc

[30] "HDF5." The HDF Group. http://www.hdfgroup.org/HDF5/

[31] "h5py." https://code.google.com/p/h5py/

[32] "HDF5 Files – MATLAB & Simulink." The Mathworks, Inc. http://www.mathworks.com/help/matlab/hdf5-files.html

[33] "Hough transform." Planetmath.org. http://planetmath.org/HoughTransform

[34] J. D. Mathews. (2013). Personal Communication. The Pennsylvania State University.

[35] P. Vixie. (1996). "CRON." System Manager's Manual. 4th Berkley Distribution.

[36] J. Jones, *et al.* (1998). "An improved interferometer design for use with meteor radars." *Radio Science*, Vol 33.

[37] J. Urbina. (2013). Personal Communication. The Pennsylvania State University.

[38] (2012). "Noise Figure." Microwaves101.com. http://www.microwaves101.com/encyclopedia/noisefigure.cfm

[39] A. Hackett, *et al.* (2013). "Assembly and Operation Manual for the Bit Pattern Generator." The Pennsylvania State University.

[40] "Board Signal Generators, Clock Generators, & Locking Programmable Oscillators." Novatech Instruments, Inc. http://www.novatechsales.com/Bench-Signal-Generator.html

[41] "VHF Pulse Transmitter Manual." Manual for WPT-50 Transmitter. Tycho Technologies.

[42] R. Seal. "GnuRadar." Source Code Repository. https://github.com/rseal/GnuRadar

[43] R. Seal and J. Urbina. (2010). "Reconfigurable Virtual Instrumentation Design for Radar using Object-Oriented Techniques and Open-Source Tools." *Radar Technology*, Chapter 18. InTech. http://www.intechopen.com/books/radar-technology/reconfigurable-virtual-instrumentation-design-for-radar-using-object-oriented-techniques-and-open-so

[44] R. Seal. "BitPatternGenerator." Source Code Repository. https://github.com/rseal/BitPatternGenerator

[45] "Pulse Transmitter System User Manual PTS Series." Genesis Software. Rev. 1.02.

[46] "Technical Manual for 2U GENESYS 3.3 kW Programmable DC Power Supplies." TDK-Lambda Americas Inc. Rev. E. `http://www.us.tdk-lambda.com/hp/pdfs/Product_manuals/83503001.pdf`

[47] Z. Stephens, *et al.* (2012). "Automated Classification of Meteor Reflections." Conference Poster. *Coupling, Energetics, and Dynamics of Atmospheric Regions*, Santa Fe.

[48] Z. Stephens. (2012). "Some Mild Sauron Documentation." The Pennsylvania State University.

[49] "Ionosonde." (2012). HF Underground. `http://www.hfunderground.com/wiki/Ionosonde`

[50] "United States Frequency Allocations: The Radio Spectrum." (2003). National Telecommunications and Information Administration, Office of Spectrum Management. `http://www.ntia.doc.gov/files/ntia/publications/2003-allochrt.pdf`

[51] (2013). "HDF5 FAQ – Questions About the Software." The HDF5 Group. `http://www.hdfgroup.org/hdf5-quest.html#grdwt`

[52] T. Boehmer. (2013). "Design and Verification of a Low-Power GPS-Disciplined Oscillator For Use In Distributed Sensor Arrays." Master's Thesis. The Pennsylvania State University.

[53] A. Hackett, *et al.* (2012). "A 50-MHz Digital Radar System for Ionospheric Studies." Conference Poster. *Coupling, Energetics, and Dynamics of Atmospheric Regions*, Santa Fe.

[54] R. Sorbello, *et al.* (2012). "An Overview of a Cognitive Radar System to Study Plasma Irregularities near the Peruvian Andes." Conference Poster. *Coupling, Energetics, and Dynamics of Atmospheric Regions*, Santa Fe.

[55] A. Hackett, *et al.* (2013). "Development of a Reconfigurable Ionosonde Receiver Using a Software-defined Radio Hardware Platform." Conference Poster. *Coupling, Energetics, and Dynamics of Atmospheric Regions*, Boulder.

[56] A. Hackett, *et al.* (2013). "Development of an Advanced Digital Radar Network for Mid-latitude Ionospheric Studies." Conference Poster. *Coupling, Energetics, and Dynamics of Atmospheric Regions*, Boulder.

[57] R. Sorbello, *et al.* (2013). "First steps towards the implementation of a cognitive radar to study plasma instabilities near the Peruvian Andes." Conference Poster. *Coupling, Energetics, and Dynamics of Atmospheric Regions*, Boulder.

Appendix A

Selected Code Listings

This appendix provides listings of original and modified code and utility configurations (e.g., `cron`) for PISCO, PARIS, and CIRI@PSU, as well as general-purpose tools.

A.1 PISCO

Listing A.1: `IonosondeRxRun.cpp` – data capture program for PISCO.

```
#include <iostream>
#include <sys/time.h>

#include <boost/shared_ptr.hpp>
#include <boost/lexical_cast.hpp>

#include <usrp/usrp/standard.h>

#include <gnuradar/GnuRadarDevice.h>
#include <gnuradar/GnuRadarTypes.hpp>
#include <gnuradar/GnuRadarSettings.h>
#include <gnuradar/SynchronizedBufferManager.hpp>
#include <gnuradar/SharedMemory.h>
#include <gnuradar/ProducerThread.h>
#include <gnuradar/ConsumerThread.h>
#include <gnuradar/ProducerConsumerModel.h>
#include <gnuradar/yaml/SharedBufferHeader.hpp>

#include <hdf5r/HDF5.hpp>
#include <hdf5r/Complex.hpp>
#include <hdf5r/Time.hpp>
#include <hdf5r/Complex.hpp>

#include "IonosondeRxDevice.h"
#include "Scheduler.h"
#include "timer_us.h"
```

```

using namespace boost;
using namespace gnuradar;


int main( int argc, char** argv ) {

    typedef boost::shared_ptr<gnuradar::IonosondeRxDevice>
        IonosondeRxDevicePtr;
    typedef boost::shared_ptr<SynchronizedBufferManager>
       SynchronizedBufferManagerPtr;
    typedef boost::shared_ptr<SharedMemory>
        SharedBufferPtr;
    typedef std::vector<SharedBufferPtr>
        SharedArray;
    typedef boost::shared_ptr<HDF5>
        Hdf5Ptr;
    typedef boost::shared_ptr<ProducerThread>
        ProducerThreadPtr;
    typedef boost::shared_ptr<ConsumerThread>
        ConsumerThreadPtr;
    typedef boost::shared_ptr<ProducerConsumerModel>
        PCModelPtr;
    typedef boost::shared_ptr< ::yml::SharedBufferHeader>
        SharedBufferHeaderPtr;

    // FIXME Ionosonde parameters -- should be changed to a config file
    const int CLKRATE = 64000000;
    const int SAMPRATE = 500000;
    const int CHANNELS = 2;
    const int PRF = 40;
    const int SAMPBYTES = 4;
    const int NUMBUFFERS = 10;
    const int BYTESPERBUF = SAMPRATE*SAMPBYTES*CHANNELS;
    const int NUMFREQ = 300;
    const float BAUD = 40e-6;
    const std::string CODE = "1111100110101";

    // GnuRadar Settings
    gnuradar::GnuRadarSettingsPtr grSettings( new gnuradar::GnuRadarSettings() );
    grSettings->numChannels = CHANNELS;
    grSettings->decimationRate = CLKRATE / SAMPRATE;
    grSettings->fpgaFileName = "usrp1_iono_rx_300.rbf";
    grSettings->fUsbBlockSize = 0;
    grSettings->fUsbNblocks = 0;
    grSettings->mux = 0xf3f2f1f0;
    grSettings->firmwareFileName = "std.ihx";

    // Set up the HDF5 data tags -- These should come from the config file
    Hdf5Ptr h5File = Hdf5Ptr ( new HDF5 ( "/data/IonosondeRx", hdf5::WRITE ) );
    h5File->Description( "USRP Ionosonde Receiver" );
    h5File->WriteStrAttrib( "INSTRUMENT", "USRP Rev4.5" );
    h5File->WriteStrAttrib( "IPP_s", boost::lexical_cast<std::string>( 1.0/float(PRF) ) );
    h5File->WriteStrAttrib( "SAMP_BW_Hz", boost::lexical_cast<std::string>( SAMPRATE ) );
    h5File->WriteStrAttrib( "SAMP_FORMAT", "Complex 32-bit Integer" );
    h5File->WriteStrAttrib( "CHANNELS", boost::lexical_cast<std::string>( grSettings->
        numChannels ) );
    h5File->WriteStrAttrib( "SWEEP_TIME_s", "0.2" );
    h5File->WriteStrAttrib( "FPGA_BITSTREAM", grSettings->fpgaFileName );
    h5File->WriteStrAttrib( "BAUD_s", boost::lexical_cast<std::string>( BAUD ) );
    h5File->WriteStrAttrib( "CODE", CODE );

    // The receiver device (inherits from GnuRadarDevice)
    IonosondeRxDevicePtr myUSRP( new gnuradar::IonosondeRxDevice( grSettings, "/home/radar/
        pisco/config/igram300.txt" ) );

    // Create the 1-s data buffers in /dev/shm
    SharedArray array;

    for ( int i = 0; i < NUMBUFFERS; ++i ) {
        std::string bufferName = "GnuRadar" + boost::lexical_cast<std::string>(i) + ".buf";

        SharedBufferPtr myBufPtr (
            new SharedMemory (
                bufferName,
                BYTESPERBUF,
                SHM::CreateShared,
                0666 )
            );

        array.push_back( myBufPtr );
    }

    // Set up the buffer manager
    SynchronizedBufferManagerPtr bufferManager = SynchronizedBufferManagerPtr(
        new SynchronizedBufferManager( array, NUMBUFFERS, BYTESPERBUF ) );

    std::vector<hsize_t> dimVector;
    dimVector.push_back( static_cast<int> ( PRF ) );
    dimVector.push_back( static_cast<int> ( SAMPRATE / PRF * CHANNELS ) );

    SharedBufferHeaderPtr header = SharedBufferHeaderPtr (
        new ::yml::SharedBufferHeader (
            NUMBUFFERS,                    // # of buffers
            BYTESPERBUF,             // bytes per buffer
            SAMPRATE,              // sample rate
            CHANNELS,                    // # of channels
            PRF,                   // ipps per buffer
            SAMPRATE/PRF*CHANNELS         // samples per buffer
            )
        );


    // Set up the producer/consumer model
    ProducerThreadPtr producer = ProducerThreadPtr (
        new ProducerThread ( bufferManager, myUSRP )
        );

    header->Write( 0, 0, 0 );

    ConsumerThreadPtr consumer = ConsumerThreadPtr (
        new ConsumerThread ( bufferManager, header, h5File, dimVector )
        );

    PCModelPtr pcModel ( new ProducerConsumerModel() );
    pcModel->Initialize( bufferManager, producer, consumer );


    // Wait until the system time (synchronized via NTP) is at the next minute
    //  --> This will be scheduled with cron
    Scheduler myScheduler(60);

    uint64_t tic, toc;

    // Start the ionosonde data collection and retuning
    std::cout << ">>> Starting ionosonde receiver... " << std::endl;
    myUSRP->Start();
    pcModel->Start();

    tic = timer_us();
    myUSRP->Wait();
    toc = timer_us();

    // Finish and clean up
    std::cout << ">>> Ionosonde finished." << std::endl;
    std::cout << ">>> Total elapsed time: " << toc-tic << " us." << std::endl;
    pcModel->Stop();
    myUSRP->Stop();

    return 0;
};
```

Listing A.2: `IonosondeRxDevice.h` – inherited ionosonde device for PISCO.

```
#ifndef IONOSONDERXDEVICE_H
#define IONOSONDERXDEVICE_H

#include <fstream>
#include <iostream>

#include <gnuradar/GnuRadarDevice.h>
#include <gnuradar/SThread.h>

#include <boost/lexical_cast.hpp>
#include <usrp/usrp/basic.h>
#include <usrp/fpga/fpga_regs_standard.h>

#include "timer_us.h"

namespace gnuradar{

const double DEFAULT_START_FREQ = 2000000;
const double DEFAULT_END_FREQ = 20000000;
const double DEFAULT_FREQ_STEP = 2000000;
const int DEFAULT_STEP_TIME_US = 200000;

// FIXME: No longer doing frequency retuning on the host computer, hence
//   the the commented code.  It should all be removed at some point to
//   simplify the code.

class IonosondeRxDevice: public GnuRadarDevice, public thread::SThread {

    std::vector<double> freqList_;
    std::string freqListFilename_;
    std::vector<double>::iterator freqPos_;
    bool fixedTimeStep_;  // not used currently, but planned for variable step operation

    void ResetFreqSweep_(){

        usrp_->_write_fpga_reg( FR_USER_1, 1 );      // Pull reset high
        usrp_->_write_fpga_reg( FR_USER_0, 0 );      // Make sure frequency sweep is disabled
        usrp_->_write_fpga_reg( FR_USER_1, 0 );      // Bring reset low again
    }

    void InitFreqList_(){

        if ( !freqListFilename_.compare("") ) {
            // Load a default (linearly spaced) frequency list
            for( double i=DEFAULT_START_FREQ; i < DEFAULT_END_FREQ; i+=DEFAULT_FREQ_STEP )
                freqList_.push_back( i );

            //std::cout << "Default frequency list loaded." << std::endl;
        }
        else {
            // Load a frequency list from a file
            std::ifstream f_file;
            f_file.open( freqListFilename_.c_str(), std::ios::in );

            if( f_file.good() ) {
                std::string currFreq;

                while( !f_file.eof() ) {
                    getline( f_file, currFreq );
                    freqList_.push_back( atof( currFreq.c_str() ) );
                }
                //std::cout << "Loaded frequency list from " << freqListFilename_ << std::
                    endl;
            }
            else {
                // If there's a problem loading the file, load the default list
                //std::cout << "Error loading " << freqListFilename_ <<
                //    ". Loading default frequency list..." << std::endl;
                freqListFilename_ = "";
                InitFreqList_();
            }
        }

        freqPos_ = freqList_.begin();

        for( int i=0; i < grSettings_->numChannels; ++i ) {
            Retune( i, freqList_[0] );
            //std::cout << "### Channel " << i << " initialized to " << freqList_[0] <<
            //    " Hz." << std::endl;
        }

    };

public:

    IonosondeRxDevice( GnuRadarSettingsPtr grSettings ) :
        GnuRadarDevice ( grSettings ),
        fixedTimeStep_ ( true ),
        freqListFilename_ ( "" ) {

        std::cout << "### Instantiating IonosondeRxDevice... ";
        InitFreqList_();
        ResetFreqSweep_();
        for( int i=0; i < 4; ++i )
            usrp_->set_pga( i, 20 );
        }

    IonosondeRxDevice( GnuRadarSettingsPtr grSettings, std::string freqListFilename ) :
        GnuRadarDevice( grSettings ),
        fixedTimeStep_ ( true ),
        freqListFilename_ ( freqListFilename ) {

        std::cout << "### Instantiating IonosondeRxDevice... ";
        InitFreqList_();
        ResetFreqSweep_();
        for( int i=0; i < 4; ++i )
            usrp_->set_pga( i, 20 );
    }

    void Retune( const int channel, const double newFreq ) {

        grSettings_->Tune( channel, newFreq );
        //usrp_->set_rx_freq( channel, grSettings_->tuningFrequency[channel] );
    }

    void SetFreqList( std::string freqListFilename ) {
        freqListFilename_ = freqListFilename;
        InitFreqList_();
    }

    virtual void Run() {
        uint64_t t1, t2;

        // Enable frequency sweeping on the FPGA
        usrp_->_write_fpga_reg( FR_USER_0, 1 );

        // The end()-1 is so we don't retune to baseband at the end
        while ( freqPos_ != freqList_.end()-1 ) {

            t1 = timer_us();
            for( int i=0; i < grSettings_->numChannels; ++i )
                Retune( i, *freqPos_ );
            freqPos_++;
            t2 = timer_us();
            Sleep(thread::USEC, DEFAULT_STEP_TIME_US - (t2-t1) );
        }

        // Disable frequency sweeping on the FPGA
        usrp_->_write_fpga_reg( FR_USER_0, 0 );

    }

};

};

#endif
```

Listing A.3: `Scheduler.h` – high-precision scheduler for PISCO.

```
#ifndef SCHEULDER_H
#define SCHEDULER_H

#include <iostream>
#include <stdint.h>
#include <sys/time.h>
#include <gnuradar/SThread.h>
#include "timer_us.h"
using namespace thread;

#define NS_THRES_DEFAULT 1*ONE_E6;       // default of 1 ms seems to be okay for our system
//#define NS_THRES_DEFAULT 10*ONE_E6;       // default of 10 ms seems to be okay for CADI
    system

class Scheduler {

    long interval_us_;           // time interval in us
    int ns_thres_;               // "close enough" threshold in ns to complete the run
    bool debug_;         // debugging messages enabled?

public:

    Scheduler() :
        interval_us_( 15/60*ONE_E6 ), debug_( false ) { Run(); };

    Scheduler( int interval_s ) :
        interval_us_( interval_s*ONE_E6 ), debug_( false ) { Run(); };

    Scheduler( int interval_s, bool debug ) :
        interval_us_( interval_s*ONE_E6 ), debug_( debug ) { Run(); };

    void Run () {

        bool end = false;
        ns_thres_ = NS_THRES_DEFAULT;
        uint64_t target = 0;

        timeval tv;
        timespec te;

        // The idea here is that we get the current time and the target time (at multiples
            of
        //  interval_s), find the difference, and then wait half that time and repeat.  Once
             this
        //  "waiting" time is below the threshold (ns_thres_), we go ahead and let the
            scheduler
        //  finish and the calling program can execute whatever it needs to.

        while(!end)  {
            gettimeofday( &tv, NULL );

            uint64_t startTime = tv.tv_usec + (uint64_t)(tv.tv_sec)*ONE_E6;

            if( startTime > target && target > 0 ) // failsafe end condition - sometimes the
                break;                         //  threshold isn't really enough

            uint64_t waitTime = interval_us_ - (startTime % interval_us_);
            //uint64_t waitTime = interval_us_ - (startTime % interval_us_) - 2.0*ONE_E6; //
                 Fudge factor tacked on here for CADI

            if(target == 0)
                target = startTime + waitTime;

            if(debug_) std::cout << "waitTime: " << waitTime << "us" << std::endl;

            waitTime /= 2;
            te.tv_sec = waitTime / ONE_E6;
            te.tv_nsec = waitTime*ONE_E3 % ONE_E9;

            nanosleep( &te, NULL );

            if( (te.tv_sec < 1 && te.tv_nsec < ns_thres_) ) end = true; // desired end
                condition
        }

        if(debug_) {
            if(!end)
                std::cout << "Scheduler started late... :-(" << std::endl;
            else
                std::cout << "Scheduler finished. Now back to your regularly scheduled
                    program. :-)" << std::endl;
        }
    }
};
#endif
```

Listing A.4: `timer_us.h` – timer function used by high-precision scheduler for PISCO.

```
#ifndef TIMER_US_H
#define TIMER_US_H

#include <sys/time.h>

```

```
uint64_t timer_us()
{
    timeval tv;
    gettimeofday(&tv, NULL);

    uint64_t ret = tv.tv_usec;
    ret += (tv.tv_sec * 1000000);

    return ret;
}
#endif
```

Listing A.5: `Makefile` for building data capture program for PISCO.

```
all:
        g++ src/IonosondeRxRun.cpp -o bin/IonosondeRxRun deps/GnuRadar/usrp/src/
            usrp_standard.cc deps/GnuRadar/usrp/src/usrp_basic.cc deps/GnuRadar/programs/Run
            /ProducerThread.cxx deps/GnuRadar/programs/Run/ConsumerThread.cxx -fpermissive -
            lusb-1.0 -L./deps/GnuRadar/build/usrp/ -lgnuradar -lpthread -lrt -lhdf5_hl_cpp -
            lboost_system -lboost_filesystem -lyaml-cpp -lhdf5_cp
```

Listing A.6: `usrp_std.v` – FPGA design for PISCO.

```
// -*- verilog -*-
//
//  USRP - Universal Software Radio Peripheral
//
//  Copyright (C) 2003,2004 Matt Ettus
//
//  This program is free software; you can redistribute it and/or modify
//  it under the terms of the GNU General Public License as published by
//  the Free Software Foundation; either version 2 of the License, or
//  (at your option) any later version.
//
//  This program is distributed in the hope that it will be useful,
//  but WITHOUT ANY WARRANTY; without even the implied warranty of
//  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//  GNU General Public License for more details.
//
//  You should have received a copy of the GNU General Public License
//  along with this program; if not, write to the Free Software
//  Foundation, Inc., 51 Franklin Street, Boston, MA  02110-1301  USA
//

// Top level module for a full setup with DUCs and DDCs

// Define DEBUG_OWNS_IO_PINS if we're using the daughterboard i/o pins
// for debugging info.  NB, This can kill the m'board and/or d'board if you
// have anything except basic d'boards installed.

// Uncomment the following to include optional circuitry

`include "config.vh"
`include "../../common/fpga_regs_common.v"
`include "../../common/fpga_regs_standard.v"

module usrp_std
(output MYSTERY_SIGNAL,
 input master_clk,
 input SCLK,
 input SDI,
 inout SDO,
 input SEN_FPGA,

 input FX2_1,
 output FX2_2,
 output FX2_3,

 input wire [11:0] rx_a_a,
 input wire [11:0] rx_b_a,
 input wire [11:0] rx_a_b,
 input wire [11:0] rx_b_b,

 output wire [13:0] tx_a,
 output wire [13:0] tx_b,

 output wire TXSYNC_A,
 output wire TXSYNC_B,

  // USB interface
 input usbclk,
 input wire [2:0] usbctl,
 output wire [1:0] usbrdy,
 inout [15:0] usbdata,  // NB Careful, inout

 // These are the general purpose i/o's that go to the daughterboard slots
 inout wire [15:0] io_tx_a,
 inout wire [15:0] io_tx_b,
 inout wire [15:0] io_rx_a,
 output wire [15:0] io_rx_b
 //inout wire [15:0] io_rx_b
 );
   wire [15:0] debugdata,debugctrl;
   assign MYSTERY_SIGNAL = 1'b0;

   wire clk64,clk128;

   wire WR = usbctl[0];
   wire RD = usbctl[1];
   wire OE = usbctl[2];

   wire have_space, have_pkt_rdy;
   assign usbrdy[0] = have_space;
   assign usbrdy[1] = have_pkt_rdy;

   wire   tx_underrun, rx_overrun;
   wire   clear_status = FX2_1;
   assign FX2_2 = rx_overrun;
   assign FX2_3 = tx_underrun;

   wire [15:0] usbdata_out;

   wire [3:0]  dac0mux,dac1mux,dac2mux,dac3mux;

   wire        tx_realsignals;
   wire [3:0]  rx_numchan;
   wire [2:0]  tx_numchan;

   wire [7:0]  interp_rate, decim_rate;
   wire [31:0] tx_debugbus, rx_debugbus;

   wire        enable_tx, enable_rx;
   wire        tx_dsp_reset, rx_dsp_reset, tx_bus_reset, rx_bus_reset;
   wire [7:0]  settings;

   // Tri-state bus macro
   bustri bustri( .data(usbdata_out),.enabledt(OE),.tridata(usbdata) );

```

```
   assign      clk64 = master_clk;

   wire [15:0] ch0tx,ch1tx,ch2tx,ch3tx; //,ch4tx,ch5tx,ch6tx,ch7tx;
   wire [15:0] ch0rx,ch1rx,ch2rx,ch3rx,ch4rx,ch5rx,ch6rx,ch7rx;

   // TX
   wire [15:0] i_out_0,i_out_1,q_out_0,q_out_1;
   wire [15:0] bb_tx_i0,bb_tx_q0,bb_tx_i1,bb_tx_q1;  // bb_tx_i2,bb_tx_q2,bb_tx_i3,bb_tx_q3;

   wire        strobe_interp, tx_sample_strobe;
   wire        tx_empty;

   wire        serial_strobe;
   wire [6:0]  serial_addr;
   wire [31:0] serial_data;

   reg [15:0] debug_counter;
   reg [15:0] loopback_i_0,loopback_q_0;

   ////////////////////////////////////////////////////////////
   // Transmit Side
`ifdef TX_ON
   assign      bb_tx_i0 = ch0tx;
   assign      bb_tx_q0 = ch1tx;
   assign      bb_tx_i1 = ch2tx;
   assign      bb_tx_q1 = ch3tx;

   tx_buffer tx_buffer
     ( .usbclk(usbclk), .bus_reset(tx_bus_reset),
       .usbdata(usbdata),.WR(WR), .have_space(have_space),
       .tx_underrun(tx_underrun), .clear_status(clear_status),
       .txclk(clk64), .reset(tx_dsp_reset),
       .channels({tx_numchan,1'b0}),
       .tx_i_0(ch0tx),.tx_q_0(ch1tx),
       .tx_i_1(ch2tx),.tx_q_1(ch3tx),
       .txstrobe(strobe_interp),
       .tx_empty(tx_empty),
       .debugbus(tx_debugbus) );

 `ifdef TX_EN_0
   tx_chain tx_chain_0
     ( .clock(clk64),.reset(tx_dsp_reset),.enable(enable_tx),
       .interp_rate(interp_rate),.sample_strobe(tx_sample_strobe),
       .interpolator_strobe(strobe_interp),.freq(),
       .i_in(bb_tx_i0),.q_in(bb_tx_q0),.i_out(i_out_0),.q_out(q_out_0) );
 `else
   assign      i_out_0=16'd0;
   assign      q_out_0=16'd0;
 `endif

 `ifdef TX_EN_1
   tx_chain tx_chain_1
     ( .clock(clk64),.reset(tx_dsp_reset),.enable(enable_tx),
       .interp_rate(interp_rate),.sample_strobe(tx_sample_strobe),
       .interpolator_strobe(strobe_interp),.freq(),
       .i_in(bb_tx_i1),.q_in(bb_tx_q1),.i_out(i_out_1),.q_out(q_out_1) );
 `else
   assign      i_out_1=16'd0;
   assign      q_out_1=16'd0;
 `endif

   setting_reg #(`FR_TX_MUX)
     sr_txmux(.clock(clk64),.reset(tx_dsp_reset),.strobe(serial_strobe),.addr(serial_addr),.
         in(serial_data),
          .out({dac3mux,dac2mux,dac1mux,dac0mux,tx_realsignals,tx_numchan}));

   wire [15:0] tx_a_a = dac0mux[3] ? (dac0mux[1] ? (dac0mux[0] ? q_out_1 : i_out_1) : (
       dac0mux[0] ? q_out_0 : i_out_0)) : 16'b0;
   wire [15:0] tx_b_a = dac1mux[3] ? (dac1mux[1] ? (dac1mux[0] ? q_out_1 : i_out_1) : (
       dac1mux[0] ? q_out_0 : i_out_0)) : 16'b0;
   wire [15:0] tx_a_b = dac2mux[3] ? (dac2mux[1] ? (dac2mux[0] ? q_out_1 : i_out_1) : (
       dac2mux[0] ? q_out_0 : i_out_0)) : 16'b0;
   wire [15:0] tx_b_b = dac3mux[3] ? (dac3mux[1] ? (dac3mux[0] ? q_out_1 : i_out_1) : (
       dac3mux[0] ? q_out_0 : i_out_0)) : 16'b0;

   wire txsync = tx_sample_strobe;
   assign TXSYNC_A = txsync;
   assign TXSYNC_B = txsync;

   assign tx_a = txsync ? tx_b_a[15:2] : tx_a_a[15:2];
   assign tx_b = txsync ? tx_b_b[15:2] : tx_a_b[15:2];
`endif //  `ifdef TX_ON

   ////////////////////////////////////////////////////////
   // Receive Side
`ifdef RX_ON
   wire        rx_sample_strobe,strobe_decim,hb_strobe;
   wire [15:0] bb_rx_i0,bb_rx_q0,bb_rx_i1,bb_rx_q1,
           bb_rx_i2,bb_rx_q2,bb_rx_i3,bb_rx_q3;

   wire loopback = settings[0];
   wire counter = settings[1];

   always @(posedge clk64)
     if(rx_dsp_reset)
       debug_counter <= #1 16'd0;
     else if(~enable_rx)
       debug_counter <= #1 16'd0;
     else if(hb_strobe)
       debug_counter <=#1 debug_counter + 16'd2;

   always @(posedge clk64)
     if(strobe_interp)
       begin
      loopback_i_0 <= #1 ch0tx;
      loopback_q_0 <= #1 ch1tx;
       end

   assign ch0rx = counter ? debug_counter : loopback ? loopback_i_0 : bb_rx_i0;
   assign ch1rx = counter ? debug_counter + 16'd1 : loopback ? loopback_q_0 : bb_rx_q0;
   assign ch2rx = bb_rx_i1;
   assign ch3rx = bb_rx_q1;
   assign ch4rx = bb_rx_i2;
   assign ch5rx = bb_rx_q2;
   assign ch6rx = bb_rx_i3;
   assign ch7rx = bb_rx_q3;

   wire [15:0] ddc0_in_i,ddc0_in_q,ddc1_in_i,ddc1_in_q,ddc2_in_i,ddc2_in_q,ddc3_in_i,
       ddc3_in_q;
   wire [31:0] rssi_0,rssi_1,rssi_2,rssi_3;

   adc_interface adc_interface(.clock(clk64),.reset(rx_dsp_reset),.enable(1'b1),
                    .serial_addr(serial_addr),.serial_data(serial_data),.serial_strobe(
                        serial_strobe),
                    .rx_a_a(rx_a_a),.rx_b_a(rx_b_a),.rx_a_b(rx_a_b),.rx_b_b(rx_b_b),
                    .rssi_0(rssi_0),.rssi_1(rssi_1),.rssi_2(rssi_2),.rssi_3(rssi_3),
                    .ddc0_in_i(ddc0_in_i),.ddc0_in_q(ddc0_in_q),
                    .ddc1_in_i(ddc1_in_i),.ddc1_in_q(ddc1_in_q),
                    .ddc2_in_i(ddc2_in_i),.ddc2_in_q(ddc2_in_q),
                    .ddc3_in_i(ddc3_in_i),.ddc3_in_q(ddc3_in_q),.rx_numchan(rx_numchan) );
```

```verilog

  rx_buffer rx_buffer
    ( .usbclk(usbclk),.bus_reset(rx_bus_reset),.reset(rx_dsp_reset),
      .reset_regs(rx_dsp_reset),
      .usbdata(usbdata_out),.RD(RD),.have_pkt_rdy(have_pkt_rdy),.rx_overrun(rx_overrun),
      .channels(rx_numchan),
      .ch_0(ch0rx),.ch_1(ch1rx),
      .ch_2(ch2rx),.ch_3(ch3rx),
      .ch_4(ch4rx),.ch_5(ch5rx),
      .ch_6(ch6rx),.ch_7(ch7rx),
      .rxclk(clk64),.rxstrobe(hb_strobe),
      .clear_status(clear_status),
      .serial_addr(serial_addr),.serial_data(serial_data),.serial_strobe(serial_strobe),
      .debugbus(rx_debugbus) );

`ifdef RX_EN_0
  rx_chain #(`FR_RX_FREQ_0,`FR_RX_PHASE_0) rx_chain_0
    ( .clock(clk64),.reset(1'b0),.enable(enable_rx),
      .decim_rate(decim_rate),.sample_strobe(rx_sample_strobe),.decimator_strobe(
          strobe_decim),.hb_strobe(hb_strobe),
      .serial_addr(`FR_RX_FREQ_0),.serial_data(ddc_tuning_freq),.serial_strobe(sweep_strobe
          ),
      .i_in(ddc0_in_i),.q_in(ddc0_in_q),.i_out(bb_rx_i0),.q_out(bb_rx_q0),.debugdata(
          debugdata),.debugctrl(debugctrl));
`else
  assign      bb_rx_i0=16'd0;
  assign      bb_rx_q0=16'd0;
`endif

`ifdef RX_EN_1
  rx_chain #(`FR_RX_FREQ_1,`FR_RX_PHASE_1) rx_chain_1
    ( .clock(clk64),.reset(1'b0),.enable(enable_rx),
      .decim_rate(decim_rate),.sample_strobe(rx_sample_strobe),.decimator_strobe(
          strobe_decim),.hb_strobe(),
      .serial_addr(`FR_RX_FREQ_1),.serial_data(ddc_tuning_freq),.serial_strobe(sweep_strobe
          ),
      .i_in(ddc1_in_i),.q_in(ddc1_in_q),.i_out(bb_rx_i1),.q_out(bb_rx_q1));
`else
  assign      bb_rx_i1=16'd0;
  assign      bb_rx_q1=16'd0;
`endif

`ifdef RX_EN_2
  rx_chain #(`FR_RX_FREQ_2,`FR_RX_PHASE_2) rx_chain_2
    ( .clock(clk64),.reset(1'b0),.enable(enable_rx),
      .decim_rate(decim_rate),.sample_strobe(rx_sample_strobe),.decimator_strobe(
          strobe_decim),.hb_strobe(),
      .serial_addr(`FR_RX_FREQ_2),.serial_data(ddc_tuning_freq),.serial_strobe(sweep_strobe
          ),
      .i_in(ddc2_in_i),.q_in(ddc2_in_q),.i_out(bb_rx_i2),.q_out(bb_rx_q2));
`else
  assign      bb_rx_i2=16'd0;
  assign      bb_rx_q2=16'd0;
`endif

`ifdef RX_EN_3
  rx_chain #(`FR_RX_FREQ_3,`FR_RX_PHASE_3) rx_chain_3
    ( .clock(clk64),.reset(1'b0),.enable(enable_rx),
      .decim_rate(decim_rate),.sample_strobe(rx_sample_strobe),.decimator_strobe(
          strobe_decim),.hb_strobe(),
      .serial_addr(`FR_RX_FREQ_3),.serial_data(ddc_tuning_freq),.serial_strobe(sweep_strobe
          ),
      .i_in(ddc3_in_i),.q_in(ddc3_in_q),.i_out(bb_rx_i3),.q_out(bb_rx_q3));
`else
  assign      bb_rx_i3=16'd0;
```

```
   assign       bb_rx_q3=16'd0;
 `endif

`endif //  `ifdef RX_ON

   ////////////////////////////////////////////////////////////
   // Control Functions

   wire [31:0] capabilities;
   assign      capabilities[7] =   `TX_CAP_HB;
   assign      capabilities[6:4] = `TX_CAP_NCHAN;
   assign      capabilities[3] =   `RX_CAP_HB;
   assign      capabilities[2:0] = `RX_CAP_NCHAN;


   serial_io serial_io
     ( .master_clk(clk64),.serial_clock(SCLK),.serial_data_in(SDI),
       .enable(SEN_FPGA),.reset(1'b0),.serial_data_out(SDO),
       .serial_addr(serial_addr),.serial_data(serial_data),.serial_strobe(serial_strobe),
       //.readback_0({io_rx_a,io_tx_a}),.readback_1({io_rx_b,io_tx_b}),.readback_2(
           capabilities),.readback_3(32'hf0f0931a),
         .readback_0({io_rx_a,io_tx_a}),.readback_1({16'd0,io_tx_b}),.readback_2(
             capabilities),.readback_3(32'hf0f0931a),
       .readback_4(rssi_0),.readback_5(rssi_1),.readback_6(rssi_2),.readback_7(rssi_3)
       );

   wire [15:0] reg_0,reg_1,reg_2,reg_3;
   master_control master_control
     ( .master_clk(clk64),.usbclk(usbclk),
       .serial_addr(serial_addr),.serial_data(serial_data),.serial_strobe(serial_strobe),
       .tx_bus_reset(tx_bus_reset),.rx_bus_reset(rx_bus_reset),
       .tx_dsp_reset(tx_dsp_reset),.rx_dsp_reset(rx_dsp_reset),
       .enable_tx(enable_tx),.enable_rx(enable_rx),
       .interp_rate(interp_rate),.decim_rate(decim_rate),
       .tx_sample_strobe(tx_sample_strobe),.strobe_interp(strobe_interp),
       .rx_sample_strobe(rx_sample_strobe),.strobe_decim(strobe_decim),
       .tx_empty(tx_empty),
       //.debug_0(rx_a_a),.debug_1(ddc0_in_i),
       .debug_0(tx_debugbus[15:0]),.debug_1(tx_debugbus[31:16]),
       .debug_2(rx_debugbus[15:0]),.debug_3(rx_debugbus[31:16]),
       .reg_0(reg_0),.reg_1(reg_1),.reg_2(reg_2),.reg_3(reg_3) );

   io_pins io_pins
     (//.io_0(io_tx_a),.io_1(io_rx_a),.io_2(io_tx_b),.io_3(io_rx_b),
       .io_0(io_tx_a),.io_1(io_rx_a),.io_2(io_tx_b),.io_3(),
      .reg_0(reg_0),.reg_1(reg_1),.reg_2(reg_2),.reg_3(reg_3),
      .clock(clk64),.rx_reset(rx_dsp_reset),.tx_reset(tx_dsp_reset),
      .serial_addr(serial_addr),.serial_data(serial_data),.serial_strobe(serial_strobe));

   ////////////////////////////////////////////////
   // Misc Settings
   setting_reg #(`FR_MODE) sr_misc(.clock(clk64),.reset(rx_dsp_reset),.strobe(serial_strobe)
       ,.addr(serial_addr),.in(serial_data),.out(settings));

    ////////////////////////////////////////////////
    // Frequency Sweeping

    localparam SWEEP_COUNT_MAX = 32'd12799999;
    localparam NUM_FREQ = 9'd300;

    wire [31:0] ddc_tuning_freq;                 // The frequency we want to tune the DDS to
    reg [8:0] freq_list_addr = 9'd0;         // Address of the frquency we want in the
        freq_list_ram
    reg [31:0] switch_counter = 32'd0;       // Just a counter to get frequency switching
        every 1 s
    wire freq_sweep;                               // Enable frequency sweeping if high
    wire [31:0] freq_sweep_long;
    reg sweep_strobe;                              // This replaces the serial_strobe
    reg update;
    wire sweep_reset;
    wire [31:0] sweep_reset_long;

    //assign io_rx_b[7:0] = 7'd0;                                  // Debugging only
    //assign io_rx_b[15:8] = ddc_tuning_freq[7:0];      // Debugging only

    // Frequency sweep enable line
    setting_reg #(`FR_USER_0)
     sr_freq_sweep(.clock(clk64),.reset(sweep_reset),.strobe(serial_strobe),.addr(
         serial_addr),.in(serial_data),
          .out(freq_sweep_long));

    // Reset line
    setting_reg #(`FR_USER_1)
     sr_sweep_reset(.clock(clk64),.reset(1'b0),.strobe(serial_strobe),.addr(serial_addr),.in
         (serial_data),
          .out(sweep_reset_long));

    assign freq_sweep = freq_sweep_long[0];          // Temporary hack until I figure out a
        more elegant way to do this
    assign sweep_reset = sweep_reset_long[0];

    freq_list_ram   arecibo (
        .address ( freq_list_addr ),
        .clock ( master_clk ),
        .data ( 32'd0 ),
        .wren ( 1'b0 ),
        .q ( ddc_tuning_freq )
    );

    always @( posedge clk64 )
    begin
        if( sweep_reset )
        begin
            switch_counter <= #1 32'd0;
            freq_list_addr <= #1 9'd0;
            sweep_strobe <= #1 1'b0;
        end
        else if( freq_sweep )
        begin
            if( switch_counter == 32'd0 )
            begin
                sweep_strobe <= #1 1'b1;
                update <= #1 ~update;
                switch_counter <= #1 switch_counter + 1'b1;
            end
            else if( switch_counter >= SWEEP_COUNT_MAX )
            begin
                if( freq_list_addr >= NUM_FREQ )
                    freq_list_addr <= #1 9'd0;
                else
                    freq_list_addr <= #1 freq_list_addr + 1'b1;
                switch_counter <= #1 32'd0;
                sweep_strobe <= #1 1'b0;
            end
            else
            begin
                switch_counter <= #1 switch_counter + 1'b1;
                sweep_strobe <= #1 1'b0;
            end
        end
    end

endmodule // usrp_std
```

Listing A.7: `igram300.mif` – memory initialization for frequency sweeping RAM on FPGA for PISCO (1–20 MHz).

```
DEPTH = 512;            -- The size of data in bits
WIDTH = 32;             --The size of memory in words
ADDRESS_RADIX = DEC;    --The radix for address values
DATA_RADIX = DEC;       --The radix for data values
CONTENT                 --start of( address : data pairs )
BEGIN

0 : 67108864;
1 : 67784583;
2 : 68467147;
3 : 69156557;
4 : 69852945;
5 : 70556313;
6 : 71266795;
7 : 71984457;
8 : 72709300;
9 : 73441458;
10 : 74180930;
11 : 74927919;
12 : 75682424;
13 : 76444512;
14 : 77214251;
15 : 77991774;
16 : 78777082;
17 : 79570376;
18 : 80371589;
19 : 81180922;
20 : 81998375;
21 : 82824015;
22 : 83658044;
23 : 84500462;
24 : 85351335;
25 : 86210731;
26 : 87078851;
27 : 87955696;
28 : 88841399;
29 : 89735960;
30 : 90639581;
31 : 91552261;
32 : 92474136;
33 : 93405338;
34 : 94345869;
35 : 95295929;
36 : 96255519;
37 : 97224705;
38 : 98203756;
39 : 99192605;
40 : 100191454;
41 : 101200301;
42 : 102219349;
43 : 103248665;
44 : 104288316;
45 : 105338435;
46 : 106399158;
47 : 107470551;
48 : 108552748;
49 : 109645817;
```

```
50 : 110749892;
51 : 111865108;
52 : 112991530;
53 : 114129294;
54 : 115278533;
55 : 116439315;
56 : 117611841;
57 : 118796111;
58 : 119992327;
59 : 121200622;
60 : 122420996;
61 : 123653719;
62 : 124898857;
63 : 126156544;
64 : 127426915;
65 : 128710036;
66 : 130006110;
67 : 131315203;
68 : 132637448;
69 : 133973049;
70 : 135322071;
71 : 136684717;
72 : 138061053;
73 : 139451280;
74 : 140855533;
75 : 142273879;
76 : 143706519;
77 : 145153520;
78 : 146615151;
79 : 148091546;
80 : 149582705;
81 : 151088964;
82 : 152610389;
83 : 154147047;
84 : 155699275;
85 : 157267073;
86 : 158850708;
87 : 160450247;
88 : 162065893;
89 : 163697847;
90 : 165346175;
91 : 167011145;
92 : 168692894;
93 : 170391553;
94 : 172107258;
95 : 173840345;
96 : 175590812;
97 : 177358930;
98 : 179144831;
99 : 180948784;
100 : 182770857;
101 : 184611250;
102 : 186470233;
103 : 188347872;
104 : 190244436;
105 : 192160125;
106 : 194095075;
107 : 196049554;
108 : 198023628;
109 : 200017634;
110 : 202031772;
111 : 204066110;
112 : 206120983;
113 : 208196526;
114 : 210292940;
```

```
115 : 212410493;
116 : 214549387;
117 : 216709756;
118 : 218891934;
119 : 221096058;
120 : 223322395;
121 : 225571146;
122 : 227842579;
123 : 230136830;
124 : 232454233;
125 : 234794923;
126 : 237159169;
127 : 239547305;
128 : 241959399;
129 : 244395786;
130 : 246856802;
131 : 249342514;
132 : 251853258;
133 : 254389302;
134 : 256950915;
135 : 259538297;
136 : 262151718;
137 : 264791445;
138 : 267457747;
139 : 270150960;
140 : 272871218;
141 : 275618923;
142 : 278394277;
143 : 281197549;
144 : 284029073;
145 : 286889119;
146 : 289777954;
147 : 292695914;
148 : 295643201;
149 : 298620218;
150 : 301627165;
151 : 304664445;
152 : 307732259;
153 : 310830944;
154 : 313960901;
155 : 317122333;
156 : 320315641;
157 : 323541027;
158 : 326798961;
159 : 330089644;
160 : 333413479;
161 : 336770802;
162 : 340161947;
163 : 343587250;
164 : 347046981;
165 : 350541608;
166 : 354071400;
167 : 357636692;
168 : 361237955;
169 : 364875457;
170 : 368549533;
171 : 372260653;
172 : 376009153;
173 : 379795435;
174 : 383619768;
175 : 387482621;
176 : 391384398;
177 : 395325500;
178 : 399306197;
179 : 403327024;
```

```
180 : 407388385;
181 : 411490549;
182 : 415634052;
183 : 419819296;
184 : 424046685;
185 : 428316687;
186 : 432629640;
187 : 436986012;
188 : 441386206;
189 : 445830759;
190 : 450320073;
191 : 454854619;
192 : 459434799;
193 : 464061083;
194 : 468733940;
195 : 473453908;
196 : 478221322;
197 : 483036785;
198 : 487900769;
199 : 492813674;
200 : 497776106;
201 : 502788467;
202 : 507851294;
203 : 512965124;
204 : 518130426;
205 : 523347805;
206 : 528617663;
207 : 533940603;
208 : 539317097;
209 : 544747815;
210 : 550233159;
211 : 555773734;
212 : 561370144;
213 : 567022857;
214 : 572732480;
215 : 578499681;
216 : 584324865;
217 : 590208769;
218 : 596151863;
219 : 602154818;
220 : 608218238;
221 : 614342727;
222 : 620528889;
223 : 626777328;
224 : 633088648;
225 : 639463588;
226 : 645902683;
227 : 652406606;
228 : 658976027;
229 : 665611617;
230 : 672314048;
231 : 679083923;
232 : 685921981;
233 : 692828892;
234 : 699805328;
235 : 706852027;
236 : 713969728;
237 : 721159033;
238 : 728420816;
239 : 735755681;
240 : 743164365;
241 : 750647675;
242 : 758206347;
243 : 765841121;
244 : 773552802;
```

```
245 : 781342128;
246 : 789209837;
247 : 797156869;
248 : 805183827;
249 : 813291652;
250 : 821481148;
251 : 829753053;
252 : 838108308;
253 : 846547651;
254 : 855071953;
255 : 863682154;
256 : 872379060;
257 : 881163476;
258 : 890036409;
259 : 898998664;
260 : 908051113;
261 : 917194762;
262 : 926430486;
263 : 935759222;
264 : 945181910;
265 : 954699423;
266 : 964312768;
267 : 974022951;
268 : 983830912;
269 : 993737656;
270 : 1003744125;
271 : 1013851324;
272 : 1024060327;
273 : 1034372140;
274 : 1044787839;
275 : 1055308361;
276 : 1065934781;
277 : 1076668240;
278 : 1087509811;
279 : 1098460502;
280 : 1109521519;
281 : 1120693870;
282 : 1131978695;
283 : 1143377203;
284 : 1154890466;
285 : 1166519694;
286 : 1178266027;
287 : 1190130606;
288 : 1202114638;
289 : 1214219400;
290 : 1226446031;
291 : 1238795739;
292 : 1251269868;
293 : 1263869557;
294 : 1276596082;
295 : 1289450852;
296 : 1302435008;
297 : 1315549892;
298 : 1328796913;
299 : 1342177280;
[300..511] : 0;

END
```

Listing A.8: `main.py` – Python data processing script for PISCO.

```
#!/usr/bin/env python
#
# USRP-based Ionosonde Receiver Data Plotter
# Alex Hackett, Tejas Nagarmat
# Penn State University, Arecibo Observatory
# Spring 2013

import argparse
import numpy as np
import matplotlib as mp
from oct2py import octave
import scipy.io as sio

from hdf5_read import *
from hdf5_write import *


def main():

    # CLP
    parser = argparse.ArgumentParser( description='Process and plot USRP-based Ionosonde
        receiver data' )
    parser.add_argument( 'filename', metavar='datafile.h5', type=str, nargs='+', default='/
        data/data_latest.h5',
                         help='the HDF5 file to process and plot' )
    parser.add_argument( '-m', '--multi_channel', action='store_true', help='process
        multiple channels together', dest='multi', default=False )
    parser.add_argument( '-c', '--channel', type=int, default=0, help='channel number to
        plot (indexed starting with 0) -- not valid with -m' )
    parser.add_argument( '-v', '--verbose', action='store_true', help='be verbose', dest='
        verbose', default=False )
    parser.add_argument( '--mat', action='store_true', help='store *.mat datafile', dest='
        mat', default=False )
    parser.add_argument( '--snr', action='store_true', help='calculate SNR instead of just
        power', dest='snr', default=False )
    parser.add_argument( '-r', '--radar', action='store_true', help='assume fixed-frequency
        radar mode', dest='fixed', default=False )
    args = parser.parse_args()

    args.filename = ''.join( args.filename )
    params = { 'chan': args.channel, 'filename': args.filename, 'multi': args.multi, 'mat':
        args.mat, 'snr': args.snr, 'fixed':args.fixed }

    # Read the specified HDF5 and spit out the I/Q data and parameters
    iq,params = hdf5_read( params, args.verbose )

    # Call Octave script to shift, downconvert, decode, and plot data
    octave.addpath('/usr/local/bin/')
    print 'Running Octave plotter...'
    if params['multi']:
        print 'Running multi channel plotter...'
        octave.iono_plotter_multi( iq, params )
    else:
        octave.iono_plotter_single( iq, params )
    print 'Done plotting.'

    # Write the image plot back to the HDF5 file
    hdf5_write( params, args.verbose )

if __name__ == '__main__':
    main()
```

Listing A.9: `hdf5_read.py` – script for loading I/Q data from HDF5 data files for PISCO.

```
#!/usr/bin/env python

import numpy as np
import matplotlib as mp
import h5py
import re
import scipy.io as sio

def hdf5_read( params, verbose ):

    if( verbose ):
        print 'Opening file: ', params['filename']

    # FIXME put a try/catch here
    fid = h5py.File( params['filename'], mode='r', memb_size=1<<30 )
    keys = [ key for key in fid.keys() if re.match( 'T\d{8,8}', key ) ]
    ntabs = len( keys )
    toff = 0

    for x in np.arange( ntabs ):
        # Generate table name
        tname = 'T' + str(x+toff).zfill(8)

        if( verbose ):
            print 'Reading Table', tname

        if( x == 0 ):
            # Read dataset parameters from HDF5 file
            nchans = fid.attrs[ 'CHANNELS' ]
            ipp = fid.attrs[ 'IPP_s' ]
            bw = fid.attrs[ 'SAMP_BW_Hz' ]
            tsweep = fid.attrs[ 'SWEEP_TIME_s' ]
            baud = fid.attrs[ 'BAUD_s' ]
            code = fid.attrs[ 'CODE' ]

            # Put parameters into a dictionary structure
            params['nchans'] = int(nchans)
            params['ipp'] = float(ipp)
            params['bw'] = float(bw)
            params['tsweep'] = float(tsweep)
            params['baud'] = float(baud)
            params['code'] = code

        # Read tabular data
        tdata = fid[ tname ];
        tx = tdata.shape[0]
        if params['multi']:
            ty = tdata.shape[1]
        else:
            ty = tdata.shape[1] #/params['nchans']


        if( x == 0 ):
            # Instantiate a big numpy array to hold all the data
            if params['multi']:
                iq = np.zeros( ( ntabs*tx, ty ), dtype=np.complex64 )
            else:
                iq = np.zeros( ( ntabs*tx, ty/params['nchans'] ), dtype=np.complex64 )

        # Read the I/Q data from the HDF5 file
        if params['multi']:
            i = tdata['real']*1.0
            q = tdata['imag']*1.0
        else:
            i = tdata['real'][:,params['chan']::params['nchans']]*1.0
            q = tdata['imag'][:,params['chan']::params['nchans']]*1.0

        # Throw the I/Q data into the big array
        iq[ x*tx:((x+1)*tx), : ] = i[:,:] + 1J*q[:,:]

    fid.close()

    print iq.shape

    return iq, params
```

Listing A.10: `iono_plotter_multi.m` for processing and plotting data for PISCO.

```
%% USRP Ionogram Plotter Octave Function
%  Alex Hackett
%  Penn State University

function x = iono_plotter_multi( iq, params )

    pkg load image

    % Use wxt backend for gnuplot
    setenv( 'GNUTERM', 'wxt' )

    % Load data and set up workspace
    %load( '/home/alex/two_chan.dat.mat' )

    % Radar parameters
    bw = params.bw;
    ipp_t = params.ipp;
    baud = params.baud;
    code = params.code;
    nchans = params.nchans;
    chan = params.chan;

    % Transpose the raw data to get IPP on x-axis and range on y-axis
    iq = double(iq)';

    i_0 = reshape( real( iq(1:nchans:end) ), [ size(iq,1)/nchans, size(iq,2) ] );
    q_0 = reshape( imag( iq(1:nchans:end) ), [ size(iq,1)/nchans, size(iq,2) ] );
    i_1 = reshape( real( iq(2:nchans:end) ), [ size(iq,1)/nchans, size(iq,2) ] );
    q_1 = reshape( imag( iq(2:nchans:end) ), [ size(iq,1)/nchans, size(iq,2) ] );

    %i = double( real(iq) )';
    %q = double( imag(iq) )';
    clear iq

    % Deinterlace channels, rotate by +/-90 degrees and sum them
    %i = reshape( real( iq(1:2:end)*exp(j*pi/2) + iq(2:2:end)*exp(-j*pi/2) ), [ size(iq,1)
        /2, size(iq,2) ] );
    %q = reshape( imag( iq(1:2:end)*exp(j*pi/2) + iq(2:2:end)*exp(-j*pi/2) ), [ size(iq,1)
        /2, size(iq,2) ] );
    %clear iq

    % Generate a power map
    pmap = 10*log10( i_0.^2 + q_0.^2 );

    % Zero out low amplitude data and scale to [0,1] to help edge detection
    pmap( pmap<30 ) = 0;
    pmap = pmap / max( pmap(:) );

    % Do edge detection on the image
    e = edge( pmap );
    clear pmap

    % Use the Hough transform to look for straight lines that are close to horizontal
    theta = pi/180*linspace( 170, 180, 100 );
    [h,r] = houghtf( e, 'line', theta );
    clear e

    %keyboard

    % Find the peak in the parameter space to give us the best line
    [rh,ch] = ind2sub( size(h), find( h==max(max(h)) ) );
    phi = pi/2 - theta(ch);
    rho = r(rh);
    clear h

    % Calculate slope and y-intercept from
    %  rho = x*cos(phi) + y*sin(phi)  ==>  y = -cot(phi)*x + rho*csc(phi)
    %                                 ==>  y = m*x + b
    m = -cot( phi );
    b = rho * csc( phi );

    %keyboard

    % Shift the data to bring the groundwave to 0 km (technically not quite correct)
    is_0 = zeros( size(i_0) );
    qs_0 = zeros( size(q_0) );
    is_1 = zeros( size(i_1) );
    qs_1 = zeros( size(q_1) );
    for ipp = 1:size(i_0,2)
        shiftAmt = -round( b + m*ipp );
        is_0(:,ipp) = circshift( i_0(:,ipp), shiftAmt );
        qs_0(:,ipp) = circshift( q_0(:,ipp), shiftAmt );
        is_1(:,ipp) = circshift( i_1(:,ipp), shiftAmt );
        qs_1(:,ipp) = circshift( q_1(:,ipp), shiftAmt );
    end

    clear i_0 i_1 q_0 q_1

    % Look for frequency offset from baseband in groundwave
    fftRange = 5:255;                                 % Range to get accurate freq offset
    %fpeaks = zeros( 1, size(iq,2) );                  % Vector to store freq offsets
    time = linspace( 0, ipp_t, size(is_0,1) );          % Time vector, length of 1 IPP
    frange = linspace( -bw/2, bw/2, length(fftRange) ); % Proper freq vector for FFT

    i = zeros( size(is_0) );
    q = zeros( size(qs_0) );
    % Loop through each IPP of the dataset and bring the signal down to baseband
    for ipp = 1:size(is_0,2)
        sig_0 = is_0(:,ipp) + j*qs_0(:,ipp);                                 % Get current
            IPP
        sig_1 = is_1(:,ipp) + j*qs_1(:,ipp);                                 % Get current
            IPP
        spec = abs( fftshift( fft( sig_0(fftRange) ) ) ); % Find spectrum
        fpeak = frange( find( spec==max(spec) ) );      % Find frequency offset
        %fpeaks(:,ipp) = fpeak;                          % Store frequency offset in list
        x = exp( j*2*pi*fpeak*time );                   % Complex exponential "LO"
        i(:,ipp) = real( (x' .* sig_0)*exp(j*pi/2) + (x' .* sig_1) );
                                     % Mix signals to bring down to baseband
        q(:,ipp) = imag( (x' .* sig_0)*exp(j*pi/2) + (x' .* sig_1) );
                                     % Mix signals to bring down to baseband
    end

    clear sig spec x time is qs

    % Phase decoding filter taps
    b = round(baud*bw);
    b = 20;
    fir = fliplr([  ones(1,b*5), ...                  % FIXME hardcoded for Barker 13
             -ones(1,b*2), ...
```

```
              ones(1,b*2), ...
             -ones(1,b*1), ...
              ones(1,b*1), ...
             -ones(1,b*1), ...
              ones(1,b*1) ]);

    % Use FIR filter to decode the signal
    iq = filter( fir, 1, i+j*q, [], 1 );
    clear i q

    % Shift data to set the ACF of the groundwave at 0 km
    a = -length(code)*round(baud*bw);
    iq = circshift( iq, a );

    % Generate baseband decoded power map
    pmap = real(iq).^2 + imag(iq).^2;
    clear iq

    % Do some basic denoising
    %clip_lower = 1e-3;
    %snr = clip_lower*ones( size(pmap) );
    %for ipp = 1:size(pmap,2)
    %    z = sort( pmap(:,ipp) );
    %    nl = mean( z(1:length(z)/2) );
    %    snr(:,ipp) = ( pmap(:,ipp) - nl ) / nl;
    %end
    %snr(snr<1e-3) = 1e-3;
    %snr = 10*log10( snr );
    %clear z pmap

    snr = 10*log10(pmap)
    clear pmap

    % Resize array down to a reasonable size for plotting
    w = 1000; h = 1000;
    snr = imresize( snr, [h, w] );

    % Finally, plot everything nicely
    out_size = [ 1024, 768 ];                          % FIXME make this adjustable at top level
    px = size(snr,1);
    py = size(snr,2);
    f = figure(1, 'Position', [0, 0, out_size], 'Visible', 'on');
    image( 1:py, linspace( 0, 1.5e5*ipp_t, px ), snr, 'CDataMapping', 'scaled' )
    xlim([1, py])
    ylim([0, 1.5e5*ipp_t])
    set(gca,'XTick', linspace( 1, py, 11 ) )
    xlabels = [ ' 1.0'; ' 1.3'; ' 1.8'; ' 2.5'; ' 3.3'; ' 4.5'; ' 6.0'; ' 8.1'; '11.0'; ' 14.8'; '20.0' ];
    set(gca,'XTickLabel', xlabels, 'FontSize', 8 );
    caxis( [40 80] )
    xlabel( 'Frequency [MHz]', 'FontSize', 10 )
    ylabel( 'Range [km]', 'FontSize', 10 )
    title( 'USRP Ionosonde Receiver Ionogram', 'FontSize', 16 )
    h = colorbar;
    ylabel ( h, 'S+N [dB]', 'FontSize', 10 )
    set( h, 'FontSize', 8 )
    axis xy;

    print( f, '/data/Ionogram.png', '-dpng', '-S1024,768' ) % FIXME make output size based on out_size
    x = 0;                  % Dummy return output

endfunction
```

Listing A.11: `hdf5_write.py` for writing ionogram image back to HDF5 data file for PISCO.

```python
#!/usr/bin/env python

import numpy as np
import matplotlib as mp
import scipy as sci
import h5py
import Image

def hdf5_write( params, verbose ):

    if( verbose ):
        print 'Writing to file: ', params['filename']

    #im = sci.misc.imread( 'rti_latest.png' )
    im = Image.open( '/data/Ionogram.png' )

    fid = h5py.File( params['filename'], mode='a' )
    keys = fid.keys()

    if 'Ionogram' in keys:
        if( verbose ):
            print 'Ionogram already added, skipping...'
    else:
        dset = fid.create_dataset( 'Ionogram', data=im )
        dset.attrs.create( 'CLASS', 'IMAGE' )
        dset.attrs.create( 'IMAGE_SUBCLASS', 'IMAGE_TRUECOLOR' )
        dset.attrs.create( 'IMAGE_VERSION', '1.2' )

    fid.close()
```

Listing A.12: `crontab` for scheduling on PISCO.

```
# uncomment me for normal operation
14,29,44,59     * * * * /usr/local/bin/ionorun
```

Listing A.13: `ionosched.cpp` – program to use high-precision scheduler on CADI for use with PISCO.

```cpp
#include <unistd.h>
#include <ionoscheduler/IonoScheduler.h>

void main( int argc, char* argv[] ) {

    int pid = fork();

    if( pid == 0 ) {
        // kill existing cadirun if it's still running
        execl( "/usr/bin/killall", "/usr/bin/killall", "-q", "cadirun", NULL );
    }
    else {
        IonoScheduler myScheduler(60,true);
        //IonoScheduler myScheduler(30,true);
        execl( "/usr/sbin/cadirun", "/usr/sbin/cadirun", argv[1], NULL );
    }

}
```

Listing A.14: `Makefile` for building high-precision scheduler for CADI.

```
all:
	@if g++ src/ionosched.cpp -o bin/ionosched -lpthread -lrt ; then echo ">>> Ionosonde
	    Scheduler compiled successfully." ; else echo "!!! Ionosonde Scheduler
	    compilation failed." ; fi
```

Listing A.15: `crontab` for scheduling on CADI.

```
# uncomment us for normal operation
14,29,44,59 * * * * /usr/sbin/ionosched igram300.def
4,19,34,49 * * * * /usr/bin/perl /home/cadi/cadiserver.pl
```

A.2 PARIS

Listing A.16: `rti_big.py` – Python script for reading and plotting data from HDF5 files taken with GnuRadar.

```python
#!/usr/bin/env python

import numpy as np
import matplotlib as mp
mp.interactive(1)
mp.use('Agg')
import matplotlib.pyplot as pp
import h5py
import sys
import datetime
from datetime import datetime as dt
import dateutil.parser as dup
from scipy.ndimage.filters import convolve1d
from matplotlib import rcParams
rcParams['xtick.direction'] = 'out'
rcParams['ytick.direction'] = 'out'
import re
import scipy.io as sio
import argparse

C = 3e5          # Speed of light constant (km/s)

def reader( d_params, p_params, verbose=False ):

  # Read the first data file to get the radar parameters
  if( verbose):
      print 'Getting radar parameters...'
  fid = h5py.File( d_params['base_name']+str(1).zfill(8)+'.h5', mode='r', memb_size=1<<30 )
  nchans = fid.attrs[ 'CHANNELS' ]
  ipp = fid.attrs[ 'IPP' ]
  bw = fid.attrs[ 'BANDWIDTH' ]
  t_start = fid.attrs[ 'START_TIME' ]

  # Put parameters into a dictionary structure
  d_params['nchans'] = int(nchans)
  d_params['ipp'] = int(round(float(ipp)*1e6,-1))/1e6
  d_params['bw'] = float(bw)
  d_params['t_start'] = t_start

  # Get the number of tables per file
  if( verbose ):
      print 'Finding number of tables per file...'
  keys = [ key for key in fid.keys() if re.match( 'T\d{8,8}', key ) ]
  tpf = len( keys )       # Number of tables per file

  fid.close()

  ## Find the numbers of the tables we want to plot
  # FIXME this assumes you start plotting the morning after you start collecting data
  if( verbose ):
      print 'Calculating required table numbers...'
  T_start = int((dup.parse(p_params['t_start'])-dup.parse(d_params['t_start'])+datetime.
      timedelta(days=1)).total_seconds())
  T_end = int((dup.parse(p_params['t_end'])-dup.parse(d_params['t_start'])+datetime.
      timedelta(days=1)).total_seconds())
  T_num = T_end - T_start + 1
  T_offset = 0

  # Find the numbers of the files we want to plot from
  if( verbose ):
      print 'Calculating file numbers needed...'
  F_start = T_start / tpf + 1
  F_end = T_end / tpf + 1
  F_num = F_end - F_start

  if( verbose ):
      print 'Reading data tables', T_start, 'through', T_end, 'from files', F_start, '
          through', F_end

  Tx = 0
  Ty = 0

  if( verbose ):
      print 'Caclulating decoding filter coefficients...'
  fir = [i for s in [[2*(d_params['code'][n]=='1')-1]*int(bw*d_params['baud']) for n in
      range(len(d_params['code'])-1,-1,-1)] for i in s]

  r_off = 0

  # Loop through each table
  y = F_start
  x = T_start
  fid = -1234
  while ( x >= T_start and x <= T_end ):

    # This is the case where we don't have an open file yet
    if( fid == -1234):
      # Open data file
      fname = d_params['base_name']+str(y).zfill(8)+'.h5'
      print 'Opening', fname, '...'
      fid = h5py.File( fname, mode='r', memb_size=1<<30 )

      # Get list of table entries in data files
      keys = [ key for key in fid.keys() if re.match( 'T\d{8,8}', key ) ]

    # This is the data table we want to grab
    T_name = 'T'+str(x).zfill(8)
    if( verbose ):
       print 'Loading', T_name

    # Check to see if we need to jump to the next file
    while T_name not in keys:
      # Close the old file
      fid.close()

      # Open the next file FIXME make sure this doesn't go beyond F_end
      y += 1
      fname = d_params['base_name']+str(y).zfill(8)+'.h5'
      print 'Opening', fname, '...'
      fid = h5py.File( fname, mode='r', memb_size=1<<30 )

      # Regenerate the keys list for the loop to check
      keys = [ key for key in fid.keys() if re.match( 'T\d{8,8}', key ) ]

    T_data = fid[T_name]

    if( x == T_start ):
      n = x - T_start
      Tx = T_data.shape[0]
      Ty = T_data.shape[1]/d_params['nchans'] - p_params['hide_dtag']

      # Coherent integration of data
      if( p_params['int_mode'] == 'c' ):
        # Instantiate a numpy array to hold all the data before integration
        iq = np.zeros( ( np.ceil(p_params['t_int'])*Tx, Ty ), dtype=np.complex64 )

         # Instantiate a big numpy array to hold all the integrated data
         iq_big = np.zeros( (T_num/p_params['t_int'], Ty), dtype=np.complex64 )

       elif( p_params['int_mode'] == 'n' ):
         iq = np.zeros( ( T_num*Tx, Ty ), dtype=np.complex64 )
       else:
         iq = np.zeros( ( np.ceil(p_params['t_int'])*Tx, Ty ), dtype=np.complex64 )
         if( p_params['int_mode'] == 'i' ):
              pmap = np.zeros( (T_num/p_params['t_int'], Ty) )

    # Read the I/Q data from the HDF5 file
    i = T_data['real'][:,p_params['hide_dtag']+p_params['channel']::d_params['nchans']]*1.0
    q = T_data['imag'][:,p_params['hide_dtag']+p_params['channel']::d_params['nchans']]*1.0

    # Apply the decoding filter
    i_d = convolve1d( i, fir, axis=1 )
    q_d = convolve1d( q, fir, axis=1 )

    # This loop is to support integration times less than 1 s
    m = 0
    while( m*p_params['t_int']*Tx < Tx ):
      # Throw the I/Q data into the big array
      #print 'm:', m
      #print 'iq indexes:', (n+m)*p_params['t_int']*Tx, 'to', ((n+m+1)*p_params['t_int']*Tx)
      #print 'i_d indexes:', m*p_params['t_int']*Tx, 'to', (m+1)*p_params['t_int']*Tx
      #print 'iq shape:', iq.shape
      iq[ (n+m)*p_params['t_int']*Tx:((n+m+1)*p_params['t_int']*Tx),: ] =
                                   \
                                     i_d[m*p_params['t_int']*Tx:(m+1)*p_params['t_int']*Tx,:] +
                                            \
                                     1J*q_d[m*p_params['t_int']*Tx:(m+1)*p_params['t_int']*Tx,:]
      m += 1

    # Have we reached enough tables to integrate?
    if( (n+1) % p_params['t_int'] == 0 and p_params['int_mode'] != 'n'):

      if( p_params['int_mode'] == 'c' ):
        if( verbose ):
          print 'Integrating coherently...'

        # Integrate in the time direction
        for c in xrange( iq.shape[1] ):
          for r in xrange( int(iq.shape[0]*d_params['ipp']/p_params['t_int']) ):
            iq_big[r+r_off,c] = np.sum( iq[r*p_params['t_int']/d_params['ipp']:(r+1)*
                p_params['t_int']/d_params['ipp']-1,c] )

        # Reset the IQ array
        iq = np.zeros( ( p_params['t_int']*Tx, Ty ), dtype=np.complex64 )

      elif( p_params['int_mode'] == 'i' ):
        if( verbose ):
          print 'Integrating incoherently...'

        p = iq.real**2 + iq.imag**2
        for c in xrange( iq.shape[1] ):
          for r in xrange( int(iq.shape[0]*d_params['ipp']/p_params['t_int']) ):
            pmap[r+r_off,c] = np.sum( p[r*p_params['t_int']/d_params['ipp']:(r+1)*p_params['
                t_int']/d_params['ipp']-1,c] )

      n = 0
      r_off += 1
    else:
      n += 1

    x += 1

  fid.close()

  if( p_params['int_mode'] == 'c' ):
    pmap = iq_big.real**2 + iq_big.imag**2
  elif( p_params['int_mode'] == 'n' ):
    pmap = iq.real**2 + iq.imag**2

  return pmap, d_params


def rti_plotter( pmap, d_params, p_params, index=1, verbose=False ):

  ir_min = int(d_params['bw']/(C/2)*p_params['rmin'])
  ir_max = int(d_params['bw']/(C/2)*p_params['rmax'])

  pmap = pmap[:,ir_min:ir_max]

  nl = np.sort( pmap.flatten() )[round(pmap.shape[0]*pmap.shape[1]/2)]
  s = np.transpose( np.clip( pmap-nl, 1e-3, (pmap-nl).max() ) )
  snr = 10*np.log10( s/nl )

  fig = pp.figure( num=1 , figsize=(12,9) )
  if( p_params['int_mode'] == 'c' or p_params['int_mode'] == 'i'):
    lims = ( 0, snr.shape[1]*p_params['t_int'], p_params['rmin'], p_params['rmax'] )
  elif( p_params['int_mode'] == 'n' ):
    lims = ( 0, snr.shape[1], p_params['rmin'], p_params['rmax'] )
  im = pp.imshow( snr,
                  aspect='auto',
                  origin='lower',
                  extent=lims,
                  vmin=p_params['vmin'],
                  vmax=p_params['vmax'],
                  interpolation='nearest' )
  im.set_cmap( 'gnuplot' )

  ax = pp.gca()

  n_cticks = 11
  n_xticks = 10
  n_yticks = 12

  if( p_params['int_mode'] == 'n' ):
    int_string = 'No integration'
  else:
    int_string = str(p_params['t_int']) + '-s '
    if( p_params['int_mode'] == 'c' ):
      int_string += 'Coherent '
    elif( p_params['int_mode'] == 'i' ):
      int_string += 'Incoherent '
    else:
      in_string += 'Unknown '
    int_string += 'integration '

  rp_string = str(d_params['ipp']*1e3) + '-ms IPP, ' + \
              str(d_params['baud']*1e6) + '-us baud, ' + \
              str(len(d_params['code'])) + '-bit code, ' + \
              str(d_params['bw']/1e6) + '-MHz BW, ' + \
              int_string

  pp.title( 'PSU All-Sky Radar: 15 June 2013' )
  pp.xlabel( rp_string )
  pp.ylabel( 'Range [km]' )

  snr_sec = int((dup.parse(p_params['t_end'])-dup.parse(p_params['t_start'])+datetime.
      timedelta(seconds=0)).total_seconds())

  xticks = []
  for i in xrange( n_xticks+1 ):
    xticks.append( str((dup.parse(p_params['t_start']) + datetime.timedelta(seconds=(i*
        snr_sec/n_xticks))).time()) )

  ax.xaxis.set_major_locator( mp.ticker.MaxNLocator( n_xticks+1 ) )
  ax.set_xticklabels( xticks, rotation=20 )


  yticks = ( p_params['rmin'] + np.arange(n_yticks)*(p_params['rmax']-p_params['rmin'])
      /(1.0*n_yticks-1) )
  ax.set_yticks( yticks )

  cticks = ( p_params['vmin'] + np.arange(n_cticks)*(p_params['vmax']-p_params['vmin'])
      /(1.0*n_cticks-1) )
  cb = pp.colorbar( im, ticks=cticks )

  cb.ax.set_ylabel( 'SNR [dB]' )


  print 'Plotting...'
  if( not p_params['batch'] ):
    pp.savefig( 'rti_big.png' )
  else:
    outfile = d_params['base_name']+str(xticks[0]).zfill(8)+'.png'
    print 'Saving to', outfile, '...'
    pp.savefig( outfile )

  pp.close('all')


def main():

  # CLP
  parser = argparse.ArgumentParser( description='Process and plot USRP-based Ionosonde
      receiver data' )
  parser.add_argument( '-f', '--base_filename', type=str, default='/data/data0/rock_springs_
      ',
                      help='the HDF5-family base filename to process and plot' )
  parser.add_argument( '-v', '--verbose', action='store_true', help='be verbose', dest='
      verbose', default=False )
  parser.add_argument( '-b', '--batch', action='store_true', help='batch process data (multi
       RTI output)', dest='batch', default=False )
  parser.add_argument( '-s', '--seconds', type=int, default=5*60, help='number of seconds to
       batch process -- only valid with -b' )
  args = parser.parse_args()

  #base_name = '/data/data0/rock_springs_' # Base filename for HDF5 family
  t_start = '02:00:00'                    # Time to start plotting
  t_end = '07:59:59'                      # Time to stop plotting
  t_int = 1                               # Integration time (seconds) -- only integers for
       now
  channel = 0
  code = '110110100100010001000111 1000'
  vmin = -10
  vmax = 10
  baud = 5e-6
  rmin = 40                                # Minimum range to plot in km
  rmax = 150                               # Maximum range to plot in km
  hide_dtag = 1                                # 1: hide data tag in plot
                                               # 0: show data tag in plot
  int_mode = 'i'                               # c: coherent
                                               # i: incoherent
                                               # n: none

  d_params = { 'base_name': args.base_filename,   # Data parameters
               'code': code,
               'baud': baud }
  p_params = { 't_start': t_start,                # Plotting parameters
               't_end': t_end,
               't_int': t_int,
               'channel': channel,
               'vmin': vmin,
               'vmax': vmax,
               'rmin': rmin,
               'rmax': rmax,
               'hide_dtag': hide_dtag,
               'int_mode': int_mode,
               'batch': args.batch }

  if( not args.batch ):
    print '>>> Entering reader...'
    pmap, d_params = reader( d_params, p_params, args.verbose )

    print '>>> Entering plotter...'
    rti_plotter( pmap, d_params, p_params, args.verbose )
  else:
    n = 0
    done = False
    while( not done ):
      print '>>> Entering reader...'
      p_params['t_end'] = str( ( dup.parse(p_params['t_start']) + datetime.timedelta(seconds
          =args.seconds) ).time() )
      pmap, d_params = reader( d_params, p_params, args.verbose )

      print '>>> Entering plotter...'
      rti_plotter( pmap, d_params, p_params, n, args.verbose )

      if( dup.parse(p_params['t_end']) >= dup.parse(t_end) ):
          done = True

      #p_params['t_start'] = str( ( dup.parse(p_params['t_end']) + datetime.timedelta(
          seconds=1) ).time() )
      p_params['t_start'] = p_params['t_end']
      n += 1

if __name__ == '__main__':
  main()
```

Listing A.17: `GnuRadarDevice.h` – class for handling data transfer from multiple USRP1 devices with GnuRadar. Note, this multi-device mode is not fully functional!

```
// Copyright (c) 2010 Ryan Seal <rlseal -at- gmail.com>
//
// This file is part of GnuRadar Software.
//
// GnuRadar is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// GnuRadar is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with GnuRadar.  If not, see <http://www.gnu.org/licenses/>.
#ifndef GNURADARDEVICE_H
#define GNURADARDEVICE_H

#include<gnuradar/GnuRadarSettings.h>
#include <gnuradar/Device.h>
#include <gnuradar/StreamBuffer.hpp>
#include <gnuradar/DeviceDataInterleave.hpp>
#include <boost/lexical_cast.hpp>

#include <boost/cstdint.hpp>
#include <boost/shared_ptr.hpp>
#include <usrp/usrp/standard.h>

#include <iostream>
#include <vector>
#include <cstring>
#include <fstream>
#include <stdexcept>
#include <math.h>
#include <sstream>

namespace gnuradar{

/// Device class providing access to the USRP data stream.
class GnuRadarDevice: public Device {

    // define width of I/Q components
    typedef int16_t iq_t;

    // define synchro buffer
    typedef StreamBuffer< iq_t > SynchronizationBuffer;
    typedef boost::shared_ptr< SynchronizationBuffer >

    SynchronizationBufferPtr;
    std::vector<SynchronizationBufferPtr> synchroBuffer_;

    // define constants
    const int ALIGNMENT_SIZE_BYTES;
    const int ALIGNMENT_SIZE;
    const int FX2_FLUSH_FIFO_SIZE_BYTES;

    // this is a gnuradio pointer of some sort.
    // older versions did not use this.
    std::vector<usrp_standard_rx_sptr> usrp_;

    // configuration settings class
    GnuRadarSettingsPtr grSettings_;

    // define flags
    bool overFlow_;
    bool isFirstDataRequest_;

    //StreamBuffer<int16_t> stBuf_;
    std::vector<int> sequence_;

    // Number of USRP devices
    int nBoards_;

    //number of channels each device has
    std::vector<int> chan;
```

```

public:

    /// Constructor.
    GnuRadarDevice ( GnuRadarSettingsPtr grSettings ) :
            ALIGNMENT_SIZE ( 256 ),
            ALIGNMENT_SIZE_BYTES ( ALIGNMENT_SIZE*sizeof ( iq_t ) ),
            FX2_FLUSH_FIFO_SIZE_BYTES ( 2048 ),
            grSettings_ ( grSettings ),
            overFlow_ ( false ),
            isFirstDataRequest_ ( true ),
            sequence_ ( grSettings->Channels(), 16384 ) {


        // FIXME hard-coded 4 channels per board
        nBoards_ = int( ceil( float(grSettings_->Channels()) / 4.0 ) );

        int chans = 0;
        for( int i = 0; i < nBoards_; ++i) {
            if( nBoards_ - i == 1 ){
                chans = grSettings_->Channels() % 4;
                if(chans == 0) chans = 4; //4%4=0, so set channels to 4
            }
            else
                chans = 4;

            usrp_.push_back(
                    usrp_standard_rx::make (
                        i,
                        grSettings_->decimationRate,
                        chans,
                        grSettings_->mux,
                        grSettings_->mode,
                        grSettings_->fUsbBlockSize,
                        grSettings_->fUsbNblocks,
                        grSettings_->fpgaFileName,
                        grSettings_->firmwareFileName
                    )
            );

        }

        //check to see if devices are connected
        std::cout<<"GnuRadarDevice: checking to see if devices are connected"<<std::endl;
        for( int i = 0; i < nBoards_; ++i ) {
            if ( usrp_[i].get() == 0 ) {
                std::stringstream msg;
                msg << "USRP #" << i << " missing -- check your connections";
                throw std::runtime_error( msg.str() );
                //exit ( 0 );
            }
        }
        std::cout<<"GnuRadarDevice: all devices must be connected"<<std::endl;

        // check to see if we picked up the master and slaves correctly (if not, sort)
        // --> devices have binary "addresses" set by jumpers from io_rx_a[0-14] to GND
        // --> master has no jumpers
        // --> these pins are pulled high in the fpga by default
        // ----> the address counter is actually counting backwards from 0x7FFF
        // --> FIXME no error checking so we have to trust the user right now
        uint16_t address = 0xFFFF;
        uint16_t mask = 0x8000;         // bit 15 here is the trigger signal so we ignore it
             by using a mask

        for( uint16_t i = 0; i < usrp_.size(); ++i ) {
            address = ~( usrp_[i]->read_io( 0 ) | 0x8000 );

            while( address != i ) {
                std::swap( usrp_[address], usrp_[i] );
                address = ~( usrp_[i]->read_io( 0 ) );
            }
        }

        // setup frequency and phase for each ddc, set all gains to 0 dB by default
        for ( int i = 0; i < nBoards_; ++i ) {
            if( nBoards_ - i == 1 )
                chans = grSettings_->Channels() % 4;
            else
                chans = 4;
            for( int j = 0; j < chans; ++j) {
                usrp_[i]->set_rx_freq ( j, grSettings_->Tune ( i*4+j-1 ) );
                usrp_[i]->set_ddc_phase ( j, 0 );
                usrp_[i]->set_pga ( j, 0 );
            }
        }
    }

    /// This method is called from the Producer thread and transfers
    /// data from the hardware device to a specified buffer given
    /// by the address and bytes parameters.
    ///
    ///\param address shared memory write address.
    ///\param bytes number of bytes to write.
    virtual void RequestData ( void* address, const int bytes ) {

        std::vector<int> bytesRead (usrp_.size(), 0);
        bool overrun;
        int readRequestSizeSamples = bytes / sizeof ( iq_t );
        int bytesPerChannel = readRequestSizeSamples / grSettings_->Channels();

        //start data collection and flush fx2 buffer
        if ( isFirstDataRequest_ ) {


                int bufferSamples = 0;
                int TotNumChan = grSettings_->Channels();
                     //select how many channels to designate to each device since they
                         cannot handle more than
                     //4 channels. The current method below is not the best approach as it
                         hardcodes 4 channels
                     //for each device as long as it is not the last device.
                  for( int i = 0; i < nBoards_; ++i ) {
                      if( nBoards_ - i == 1 ) { //last device
                          bufferSamples = (bytesPerChannel * (grSettings_->Channels() % 4));
                              if((grSettings_->Channels() % 4) == 0) bufferSamples =
                                  bytesPerChannel * 4; //4%4=0
                          chan.push_back (TotNumChan - ((nBoards_-1)*4));
                          std::cout<<"GnuRadarDevice: bufferSample: "<<bufferSamples<<std::
                              endl;
                      }
                      else {
                          bufferSamples = bytesPerChannel * 4;
                          chan.push_back (4);
                          std::cout<<"GnuRadarDevice: bufferSample: "<<bufferSamples<<std::
                              endl;
                      }
                std::vector<int> subSequence(
                                sequence_.begin() + i*(chan[i]),
                                sequence_.begin() + i*(chan[i]) + chan[i]
                                );

                // Initialize stream buffers
                synchroBuffer_.push_back(
                    SynchronizationBufferPtr (
                            new SynchronizationBuffer (
                                bufferSamples,
                                ALIGNMENT_SIZE,
                                subSequence
                                )
                    )
                );

                //create temporary buffer to sync data
                iq_t buf[FX2_FLUSH_FIFO_SIZE_BYTES/sizeof ( iq_t ) ];

                // Read some data to flush the FX2 buffers in the USRP.
                // This data is discarded.
                usrp_[i]->start();
                usrp_[i]->read ( buf, FX2_FLUSH_FIFO_SIZE_BYTES, &overFlow_ );

                // write aligned data into the synchro buffer
                usrp_[i]->read (
                    synchroBuffer_[i]->WritePtr(),
                    synchroBuffer_[i]->WriteSizeBytes(),
                    &overrun
                );

                // synchronize the data stream
                synchroBuffer_[i]->Sync();

                // write a another buffer after synchronizing.
                // This is a requirement of the StreamBuffer class.
                usrp_[i]->read (
                    synchroBuffer_[i]->WritePtr(),
                    synchroBuffer_[i]->WriteSizeBytes(),
                    &overrun
                );
            }

            /****************************************
             * FIXME Data Interleaver goes here
             ****************************************/

            for(int z=0;z<2;z++){
            std::cout<<"GnuRadarDevice: RequestData(): synchroBuffer["<<z<<"] size: "
                     <<synchroBuffer_[z]->ReadSize()<<std::endl;
            }

            //TODO : make Data1/Data2 as void* and then rewrite interleave to accept void*
            void *CompleteDataPtr = NULL;
            void *Data1 = synchroBuffer_[0]->ReadPtr();
            void *Data2 = synchroBuffer_[1]->ReadPtr();

            std::vector<int> BufferSize;
                             BufferSize.push_back(synchroBuffer_[0]->ReadSize());
                             BufferSize.push_back(synchroBuffer_[1]->ReadSize());
            std::cout<<"GnuRadarDevice: BufferSize: "<<BufferSize[0]<<" & "<<BufferSize[1]<<
                std::endl;

            CompleteDataPtr = DeviceDataInterleave(Data1, Data2, BufferSize, chan);

            // copy 1 second of data from synchro buffer
            memcpy (
                address,
                CompleteDataPtr,
                bytesPerChannel * grSettings_->Channels()
            );

            // update read and write pointers
            for( int i = 0; i < synchroBuffer_.size(); ++i )
                synchroBuffer_[i]->Update();

            isFirstDataRequest_ = false;

        } else {

            for( int i = 0; i < usrp_.size(); ++i ) {
                    std::cout<<"device.h: for loop: i = "<<i<<std::endl;
                bool overFlow = false;
                //read data from USRP
                bytesRead[i] = usrp_[i]->read (
                                synchroBuffer_[i]->WritePtr(),
                                synchroBuffer_[i]->WriteSizeBytes(),
                                &overFlow
                            );
                    std::cout<<"device.h: bytesRead(): "<<bytesRead[i]<<std::endl;
                overFlow_ |= overFlow;
            }

            /**************************************
             * FIXME Data Interleaver goes here
             ***************************************/

            void *CompleteDataPtr = NULL;
            void *Data1 = synchroBuffer_[0]->ReadPtr();
            void *Data2 = synchroBuffer_[1]->ReadPtr();
            std::vector<int> BufferSize;
                             BufferSize.push_back(synchroBuffer_[0]->ReadSize());
                             BufferSize.push_back(synchroBuffer_[1]->ReadSize());

            CompleteDataPtr = DeviceDataInterleave(Data1, Data2, BufferSize, chan);

                int16_t* dataprint = reinterpret_cast<int16_t*>(CompleteDataPtr);
                int16_t* dataprint1 = reinterpret_cast<int16_t*>(Data1);
                int16_t* dataprint2 = reinterpret_cast<int16_t*>(Data2);
                //for(int d=0;d<30;d++){
                //      std::cout<<*(dataprint+d)<<"\t"<<*(dataprint1+d)<<"\t"<<*(dataprint2
                    +d)<<std::endl;
                //}

            //Transfer data to shared memory buffer
            memcpy (
                address,
                CompleteDataPtr,
                bytesPerChannel * grSettings_->Channels()
            );

            // update read and write pointers
            for( int i = 0; i < synchroBuffer_.size(); ++i )
                synchroBuffer_[0]->Update();

            if ( overFlow_ ) {
                //TODO: throw exception here
                std::cerr << "GnuRadarDevice: Data overflow detected !!!"
                          << std::endl;
            }
        }
    }

    /// Stops data collection.
    virtual void Stop() {
        for(int i=0; i<usrp_.size(); i++)
            usrp_[i]->stop();
    }
};
};
#endif
```

Listing A.18: `GnuRadarSettings.h` – class modified to support >4 channels (multiple devices) with GnuRadar. Note, this multi-device mode is not fully functional!

```
// Copyright (c) 2010 Ryan Seal <rlseal -at- gmail.com>
//
// This file is part of GnuRadar Software.
//
// GnuRadar is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// GnuRadar is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with GnuRadar.  If not, see <http://www.gnu.org/licenses/>.
#ifndef GNURADAR_SETTINGS_H
#define GNURADAR_SETTINGS_H

#include <iostream>
#include <vector>

namespace gnuradar{

   struct GnuRadarSettings {
      private:
      int numChannels;

      bool ValidChannel ( int channel ) {
         return ! ( channel < 0 || channel >= numChannels );
      }

      public:
      GnuRadarSettings() :
         whichBoard ( 0 ), decimationRate ( 8 ), numChannels ( 1 ), mux ( -1 ),
         mode ( 0 ), fUsbBlockSize ( 0 ), fUsbNblocks ( 0 ), fpgaFileName ( "" ),
         firmwareFileName ( "" ), tuningFrequency ( 4, 0.0 ), ddcPhase ( 4, 0.0 ),
         clockRate ( 64e6 ) {}

      int whichBoard;
      int decimationRate;
      int mux;
      int mode;
      int fUsbBlockSize;
      int fUsbNblocks;
      std::string fpgaFileName;
      std::string firmwareFileName;
      std::vector<double> tuningFrequency;
      int fpgaMode;
      std::vector<double> ddcPhase;
      int format;
      double clockRate;

      void Channels( int channels ) {
        numChannels = channels;
        if( tuningFrequency.size() < numChannels )
            tuningFrequency.resize ( numChannels );
        if( ddcPhase.size() < numChannels )
            ddcPhase.resize ( numChannels );
      }

      void Tune ( int channel, double frequency ) {
         if ( ValidChannel ( channel ) ) tuningFrequency[channel] = frequency;
         else std::cout << "GnuRadarSettings: Tune Error - invalid channel number " << std::
             endl;
      }

      void Phase ( int channel, double phase ) {
         if ( ValidChannel ( channel ) ) ddcPhase[channel] = phase;
         else std::cout << "GnuRadarSettings: Phase Error - invalid channel number " << std
             ::endl;
      }

      const int& Channels () {
         return numChannels;
      }

      const double& Tune ( int channel ) {
         return ValidChannel ( channel ) ? tuningFrequency[channel] : 0;
      }

      const double& Phase ( int channel ) {
         return ValidChannel ( channel ) ? ddcPhase[channel] : 0;
      }

   };
   typedef boost::shared_ptr<GnuRadarSettings> GnuRadarSettingsPtr;
};

#endif
```

Listing A.19: `Start.hpp` – command for beginning data collection with GnuRadar, modified for multi-device operation. Note, this multi-device mode is not fully functional!

```
// Copyright (c) 2010 Ryan Seal <rlseal -at- gmail.com>
//
// This file is part of GnuRadar Software.
//
// GnuRadar is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// GnuRadar is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with GnuRadar.  If not, see <http://www.gnu.org/licenses/>.
#ifndef START_HPP
#define START_HPP

#include <vector>

#include<boost/shared_ptr.hpp>
#include<boost/scoped_ptr.hpp>
```

```
#include<boost/filesystem.hpp>
#include<boost/asio.hpp>
#include<boost/lexical_cast.hpp>

#include<hdf5r/HDF5.hpp>

#include<gnuradar/GnuRadarCommand.hpp>
#include<gnuradar/ProducerConsumerModel.h>
#include<gnuradar/Device.h>
#include<gnuradar/GnuRadarDevice.h>
#include<gnuradar/SynchronizedBufferManager.hpp>
#include<gnuradar/yaml/SharedBufferHeader.hpp>
#include<gnuradar/SharedMemory.h>
#include<gnuradar/Constants.hpp>
#include<gnuradar/Units.h>
#include<gnuradar/network/StatusServer.hpp>
#include<gnuradar/commands/Response.pb.h>
#include<gnuradar/commands/Control.pb.h>
#include<gnuradar/utils/GrHelper.hpp>



namespace gnuradar {
   namespace command {

      class Start : public GnuRadarCommand {

         typedef boost::asio::io_service IoService;
         typedef boost::shared_ptr<SharedMemory> SharedBufferPtr;
         typedef std::vector<SharedBufferPtr> SharedArray;
         typedef boost::shared_ptr<HDF5> Hdf5Ptr;
         typedef boost::shared_ptr<SynchronizedBufferManager> SynchronizedBufferManagerPtr;
         typedef boost::shared_ptr< ProducerConsumerModel > PCModelPtr;
         typedef boost::shared_ptr< ProducerThread > ProducerThreadPtr;
         typedef boost::shared_ptr< ConsumerThread > ConsumerThreadPtr;
         typedef boost::shared_ptr< GnuRadarDevice > GnuRadarDevicePtr;
         typedef boost::shared_ptr< GnuRadarSettings > GnuRadarSettingsPtr;
         typedef boost::shared_ptr< Device > DevicePtr;
         typedef boost::shared_ptr< ::yml::SharedBufferHeader > SharedBufferHeaderPtr;
         typedef boost::shared_ptr< network::StatusServer > StatusServerPtr;


         // setup shared pointers to extend life beyond this call
         PCModelPtr pcModel_;
         ProducerThreadPtr producer_;
         ConsumerThreadPtr consumer_;
         SynchronizedBufferManagerPtr bufferManager_;
         Hdf5Ptr hdf5_;
         SharedArray array_;
         SharedBufferHeaderPtr header_;
         StatusServerPtr statusServer_;

         /////////////////////////////
         /////////////////////////////
         void CreateSharedBuffers( const int bytesPerBuffer ) {

            // setup shared memory buffers
            for ( int i = 0; i < constants::NUM_BUFFERS; ++i ) {

               // create unique buffer file names
               std::string bufferName = constants::BUFFER_BASE_NAME +
                  boost::lexical_cast<string> ( i ) + ".buf";

               // create shared buffers
               SharedBufferPtr bufPtr (
                     new SharedMemory (
                        bufferName,
                        bytesPerBuffer,
                        SHM::CreateShared,
                        0666 )
                     );

               // store buffer in a vector
               array_.push_back ( bufPtr );
            }
         }

         ///////////////////////////////
         // pull settings from the configuration file
         ///////////////////////////////
         GnuRadarSettingsPtr GetSettings( gnuradar::File* file ) {

            GnuRadarSettingsPtr settings( new GnuRadarSettings() );

            settings->Channels( file->numchannels() );
            settings->decimationRate = file->decimation();
            settings->fpgaFileName   = file->fpgaimage();

            //Program GNURadio
            for ( int i = 0; i < file->numchannels(); ++i ) {
               settings->Tune ( i, file->channel(i).frequency() );
               settings->Phase( i, file->channel(i).phase() );
            }

            //change these as needed
            settings->fUsbBlockSize  = 0;
            settings->fUsbNblocks    = 0;
            settings->mux            = 0xf3f2f1f0;

            return settings;
         }

         ////////////////////////////////////////////
         ////////////////////////////////////////////
         Hdf5Ptr SetupHDF5( gnuradar::File* file ) throw( H5::Exception )
         {

            Hdf5Ptr h5File_( new HDF5 ( file->basefilename() , hdf5::WRITE ) );

            h5File_->Description ( "GnuRadar Software" + file->version() );
            h5File_->WriteStrAttrib ( "START_TIME", currentTime.GetTime() );
            h5File_->WriteStrAttrib ( "INSTRUMENT", file->receiver() );
            h5File_->WriteAttrib<int> ( "CHANNELS", file->numchannels(),
                  H5::PredType::NATIVE_INT, H5::DataSpace() );
            h5File_->WriteAttrib<double> ( "SAMPLE_RATE", file->samplerate(),
                  H5::PredType::NATIVE_DOUBLE, H5::DataSpace() );
            h5File_->WriteAttrib<double> ( "BANDWIDTH", file->bandwidth(),
                  H5::PredType::NATIVE_DOUBLE, H5::DataSpace() );
            h5File_->WriteAttrib<int> ( "DECIMATION", file->decimation(),
                  H5::PredType::NATIVE_INT, H5::DataSpace() );
            h5File_->WriteAttrib<double> ( "OUTPUT_RATE", file->outputrate(),
                  H5::PredType::NATIVE_DOUBLE, H5::DataSpace() );
            h5File_->WriteAttrib<double> ( "IPP", file->ipp(),
                  H5::PredType::NATIVE_DOUBLE, H5::DataSpace() );
            h5File_->WriteAttrib<double> ( "RF", file->txcarrier() ,
                  H5::PredType::NATIVE_DOUBLE, H5::DataSpace() );

            for ( int i = 0; i < file->numchannels(); ++i ) {

               h5File_->WriteAttrib<double> (
                     "DDC" + lexical_cast<string> ( i ),
                     file->channel(i).frequency(),
                     H5::PredType::NATIVE_DOUBLE,
                     H5::DataSpace()
                     );

               h5File_->WriteAttrib<double> (
                     "PHASE" + lexical_cast<string> ( i ),
                     file->channel(i).phase(),
                     H5::PredType::NATIVE_DOUBLE,
                     H5::DataSpace()
                     );
            }

            h5File_->WriteAttrib<int> (
                  "SAMPLE_WINDOWS", file->window_size(),
                  H5::PredType::NATIVE_INT, H5::DataSpace()
                  );

            for ( int i = 0; i < file->window_size(); ++i ) {

               // TODO: Window Renaming scheme - 10/19/2010
               // Standardize window naming and add the user-defined
               // window name as a separate attribute.
               string idx = boost::lexical_cast<string> ( i );

               h5File_->WriteAttrib<int> (
                     "RxWin"+ idx + "_START",
                     file->window(i).start(),
                     H5::PredType::NATIVE_INT, H5::DataSpace()
                     );

               h5File_->WriteAttrib<int> (
                     "RxWin" + idx + "_STOP",
                     file->window(i).stop(),
                     H5::PredType::NATIVE_INT, H5::DataSpace()
                     );

               // update gnuradar shared buffer header
               header_->AddWindow( file->window(i).name(), file->window(i).start(), file->
                   window(i).stop() );
            }

            return h5File_;
         }

         public:

         ///////////////////////////////////////
         ///////////////////////////////////////
         Start( zmq::context_t& ctx, PCModelPtr pcModel): GnuRadarCommand( "start" ),
             pcModel_( pcModel )
         {
            std::string ipaddr = gr_helper::GetIpAddress("status");
            statusServer_ = StatusServerPtr( new network::StatusServer( ctx, ipaddr, pcModel
                 ) );
         }

         ///////////////////////////////////////
         ///////////////////////////////////////
         virtual const gnuradar::ResponseMessage Execute( gnuradar::ControlMessage& msg ){

            std::cout << "GNURADAR: RUN CALLED" << std::endl;

            gnuradar::ResponseMessage response_msg;
```

```

            try{
               // reset any existing configuration
               producer_.reset();
               consumer_.reset();
               bufferManager_.reset();
               hdf5_.reset();
               array_.clear();

               gnuradar::File* file = msg.mutable_file();

               // standardizes units of input file.
               gr_helper::FormatFileFromMessage( file );

               gnuradar::RadarParameters* rp = file->mutable_radarparameters();

               // setup shared buffer header to assist in real-time processing
               std::cout<<"Start.hpp: rp.bytesperbuffer: "<<rp->bytesperbuffer()<<std::endl;
               std::cout<<"Start.hpp: rp.samplesperbuffer "<<rp->samplesperbuffer()<<std::
                   endl;
               header_ = SharedBufferHeaderPtr
                  (
                   new ::yml::SharedBufferHeader
                   (
                    constants::NUM_BUFFERS,
                    rp->bytesperbuffer(),
                    file->samplerate(),
                    file->numchannels(),
                    rp->prisperbuffer(),
                    rp->samplesperbuffer()
                   )
                  );

               // read and parse configuration file->
               GnuRadarSettingsPtr settings = GetSettings( file );

               // create a device to communicate with hardware
               GnuRadarDevicePtr gnuRadarDevice( new GnuRadarDevice( settings ) );

               // make sure we don't have an existing data set
               if( gr_helper::HdfFileExists( file->basefilename() ))
               {
                  throw std::runtime_error( "HDF5 File set " + fileName +
                        " exists and cannot be overwritten. Change your "
                        "base file set name and try again");
               }

               // setup HDF5 attributes and file->set.
               hdf5_ = SetupHDF5( file );

               // setup shared memory buffers
               CreateSharedBuffers( rp->bytesperbuffer() );

               // setup the buffer manager
               bufferManager_ = SynchronizedBufferManagerPtr(
                     new SynchronizedBufferManager(
                        array_, constants::NUM_BUFFERS, rp->bytesperbuffer()) );

               // setup table dimensions column = samples per ipp , row = IPP number
               std::vector<hsize_t> dims;
               dims.push_back( rp->prisperbuffer() );
               dims.push_back ( static_cast<int> ( rp->samplesperpri() * file->numchannels()
                    ) );

               // setup producer thread
```

```
                producer_ = gnuradar::ProducerThreadPtr (
                      new ProducerThread ( bufferManager_, gnuRadarDevice ) );

                // flush header information
                header_->Write(0,0,0);

                // setup consumer thread
                consumer_ = gnuradar::ConsumerThreadPtr(
                      new ConsumerThread ( bufferManager_ , header_, hdf5_, dims ) );

                // new model
                pcModel_->Initialize( bufferManager_, producer_, consumer_);

                // start producer thread
                pcModel_->Start();

                response_msg.set_value(gnuradar::ResponseMessage::OK);
                response_msg.set_message("Data collection successfully started.");

                // Start status thread to broadcast status packets to any subscribers.
                if( statusServer_->IsActive() == false )
                {
                   statusServer_->Start();
                }

             }
             catch( std::runtime_error& e ){

                response_msg.set_value(gnuradar::ResponseMessage::ERROR);
                response_msg.set_message(e.what());

             }
             catch( H5::Exception& e ){
                response_msg.set_value(gnuradar::ResponseMessage::ERROR);
                response_msg.set_message(e.getDetailMsg());
             }

             return response_msg;
          }
       };
   };
};

#endif
```

Listing A.20: `Verify.hpp` – command for beginning data collection with GnuRadar, modified for multi-device operation. Note, this multi-device mode is not fully functional!

```
// Copyright (c) 2010 Ryan Seal <rlseal -at- gmail.com>
//
// This file is part of GnuRadar Software.
//
// GnuRadar is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// GnuRadar is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with GnuRadar.  If not, see <http://www.gnu.org/licenses/>.
```

```
#ifndef VERIFY_HPP
#define VERIFY_HPP

#include <vector>
#include <gnuradar/utils/GrHelper.hpp>
#include <gnuradar/GnuRadarCommand.hpp>
#include <gnuradar/commands/Response.pb.h>
#include <gnuradar/SystemValidation.hpp>
#include <boost/shared_ptr.hpp>
#include <gnuradar/GnuRadarSettings.h>

namespace gnuradar {
    namespace command {

        class Verify : public GnuRadarCommand {

            typedef std::vector<gnuradar::iq_t> Buffer;
            typedef Buffer::iterator BufferIterator;

            public:

            Verify(): GnuRadarCommand( "verify" ){ }

            virtual const gnuradar::ResponseMessage Execute( gnuradar::ControlMessage& msg )
            {

                Buffer buffer;

                gnuradar::ResponseMessage response_msg;

                gnuradar::File* file = msg.mutable_file();

                // standardizes units of input file.
                gr_helper::FormatFileFromMessage( file );

                gnuradar::RadarParameters* rp = file->mutable_radarparameters();

                // buffer one second's worth of data
                std::cout << "rp->Bps: " << rp->bytespersecond() << std::endl;
                buffer.resize ( rp->bytespersecond() / sizeof ( gnuradar::iq_t ) );
                void* bufferPtr = &buffer[0];

                try
                {

                    // set require gnuradar settings.
                    GnuRadarSettings settings;
                    settings.Channels( file->numchannels() );
                    settings.decimationRate = file->decimation();
                    settings.fpgaFileName   = file->fpgaimage();
                    settings.fUsbBlockSize  = 0;
                    settings.fUsbNblocks    = 0;
                    settings.mux            = 0xf3f2f1f0;

                    for ( int i = 0; i < settings.Channels(); ++i ) {

                        gnuradar::Channel* channel = file->mutable_channel(i);
                        settings.Tune ( i, channel->frequency() );
                        settings.Phase ( i, channel->phase() );
                    }

                    // create a vector of USRP objects.
                    std::vector<usrp_standard_rx_sptr> usrp;
                    GnuRadarSettingsPtr grsettings_;
                    int nBoards = int(ceil(float(settings.Channels())/USRP_MAX_CHANNELS));
                    int nChan;
                    int TotalNumChannels = settings.Channels();
                    for( int i = 0; i < nBoards; ++i ) {
                        if(nBoards - i == 1) nChan = TotalNumChannels-((nBoards-1)*USRP_MAX_CHANNELS);
                        else nChan = USRP_MAX_CHANNELS;

            //create each USRP device object
                        usrp.push_back(
                             usrp_standard_rx::make (
                                i,
                                settings.decimationRate,
                                nChan,
                                settings.mux,
                                settings.mode,
                                settings.fUsbBlockSize,
                                settings.fUsbNblocks,
                                settings.fpgaFileName,
                                settings.firmwareFileName
                                )
                    );

                    // quickly see which device is master
                    if( i == 0 ) usrp[i]->_set_led(1,1);

                    //check to see if device is connected
                    if ( usrp[i].get() == 0 ) {
                        throw std::runtime_error (
                                "GnuRadarVerify: One or more USRP devices could not be detected."
                                );
                    }

                    // setup frequency and phase for each ddc
                    //set all gain to 0dB by default
                    for ( int j = 0; j < nChan; ++j ) {
                        usrp[i]->set_rx_freq ( j, settings.Tune ( i*4+j ) );
                        usrp[i]->set_ddc_phase ( j, settings.Phase( i*4+j) );
                        usrp[i]->set_pga ( j, 0 );
                    }

                    // initialize data collection and flush FX2 buffer.
                    usrp[i]->start();
                    bool over_flow;
                    usrp[i]->read ( bufferPtr, gnuradar::FX2_FLUSH_FIFO_SIZE_BYTES, & over_flow );

                    // resize buffer aligned on required byte boundary - 512 bytes
                    int byteRequest = rp->bytespersecond();
                    int alignedBytes = byteRequest % gnuradar::BUFFER_ALIGNMENT_SIZE_BYTES;
                    int alignedByteRequest = byteRequest - alignedBytes;
                    buffer.resize ( alignedByteRequest / sizeof ( gnuradar::iq_t ) );

                    //read data from USRP
                    int bytesRead = usrp[i]->read ( bufferPtr, alignedByteRequest, & over_flow);

                    usrp[i]->stop();

                    if ( bytesRead != alignedByteRequest ) {
                        throw std::runtime_error (
                                "GnuRadarVerify: Number of bytes read is not equal to the "
                                "number of requested bytes.\n Expected " +
                                lexical_cast<string> ( alignedByteRequest ) + " Found " +
                                lexical_cast<string> ( bytesRead )  + "\n"
```

```
                                    );
                        }

                        int stride = nChan * sizeof( gnuradar::iq_t );
                        std::cout<<"Verify.hpp: setting stride to " << stride <<std::endl;

                        std::cout << "buffer.size(): " << buffer.size() << std::endl;

                        Buffer channelBuffer ( buffer.size() / stride );
                        BufferIterator bufferIter = buffer.begin();
                        BufferIterator channelBufferIter = channelBuffer.begin();

                        while ( bufferIter != buffer.end() ) {
                            *channelBufferIter = *bufferIter;
                            bufferIter += stride;
                            if( bufferIter != buffer.end() )
                                ++channelBufferIter;
                        }
                        std::cout << "chBufIter pos: " << channelBufferIter - channelBuffer.
                            begin() << std::endl;
                        // validate collected window sizes with those in configuration file.
                        SystemValidation validator;
                        bool valid = validator.Validate ( channelBuffer, file );

                        if( !valid )
                        {
                            throw std::runtime_error( validator.GetResults() );
                        }


                        // create a response packet and return to requester
                        response_msg.set_value(gnuradar::ResponseMessage::OK);
                        response_msg.set_message("Configuration Verified.");


                    }//end for loop for nBoards

                    }//end try
                    catch( std::runtime_error& e ){

                        response_msg.set_value(gnuradar::ResponseMessage::ERROR);
                        response_msg.set_message(e.what());

                    }

                    return response_msg;
                }
            };
    };
};

#endif
```

Listing A.21: `GrHelper.hpp` – class for multi-device data collection with GnuRadar. Note, this multi-device mode is not fully functional!

```
#ifndef GR_HELPER_HPP
#define GR_HELPER_HPP

#include<fstream>
#include<boost/filesystem.hpp>
#include<yaml-cpp/yaml.h>
#include<gnuradar/Constants.hpp>
#include<gnuradar/commands/Control.pb.h>
```

```
#include<hdf5r/HDF5.hpp>

namespace gr_helper{

   ///////////////////////////////////////
   ///////////////////////////////////////
   std::string GetIpAddress(const std::string& networkType )
   {
      std::string ip_addr;

      try{
         std::ifstream fin( gnuradar::constants::SERVER_CONFIGURATION_FILE.c_str() );
         YAML::Parser parser(fin);
         YAML::Node doc;
         parser.GetNextDocument(doc);
         doc[networkType]  >> ip_addr;
      }
      catch( YAML::ParserException& e )
      {
         std::cerr << e.what();
      }

      return ip_addr;
   };


   ///////////////////////////////////////
   ///////////////////////////////////////
   bool HdfFileExists ( const std::string& fileSet )
   {
      std::string fileName = fileSet + "." + hdf5::FILE_EXT;
      boost::filesystem::path file ( fileName );
      return boost::filesystem::exists ( file );
   }

   int Round( double x)
   {
      return static_cast<int>(floor( x + 0.5 ));
   }

   void FormatFileFromMessage( gnuradar::File* file )
   {
      const int BYTES_PER_SAMPLE=4;
      const double SECONDS_PER_BUFFER=1.0;
      int sum=0;

      Units units;

      // convert units
      file->set_samplerate( file->samplerate() * 1e6);
      file->set_outputrate( file->samplerate() / file->decimation());
      file->set_ipp( file->ipp() * units(file->ippunits()).multiplier);
      file->set_bandwidth( file->bandwidth() * units(file->bandwidthunits()).multiplier);
      file->set_txcarrier( file->txcarrier() * 1e6);

      for ( int i = 0; i < file->channel_size(); ++i ) {
         gnuradar::Channel* channel = file->mutable_channel(i);
         channel->set_frequency( channel->frequency() *
               units(channel->frequencyunits()).multiplier);
         channel->set_phase( channel->phase() *
               units(channel->phaseunits()).multiplier);
      }

      for ( int i = 0; i < file->window_size(); ++i ) {
         gnuradar::Window* window = file->mutable_window(i);
         UnitType u = units(window->units());
         double multiplier = u.units == "samples" ? 1e0 : u.multiplier*file->outputrate();
         window->set_start( window->start() * multiplier);
         window->set_stop( window->stop() * multiplier);
         sum += ceil(window->stop()-window->start());
      }

      gnuradar::RadarParameters* rp = file->mutable_radarparameters();
      rp->set_samplesperpri( sum );
      rp->set_pri( file->ipp() );
      rp->set_prf( 1.0/rp->pri() );
      rp->set_bytespersample( BYTES_PER_SAMPLE );
      rp->set_secondsperbuffer( SECONDS_PER_BUFFER );
      //edited secondsperbuffer so that a 2 second buffer doesnt throw an error
      rp->set_samplesperbuffer( Round(rp->prf()*rp->samplesperpri()*rp->secondsperbuffer())
          );
      rp->set_prisperbuffer( Round(rp->samplesperbuffer()/rp->samplesperpri()) );
      rp->set_bytesperbuffer( Round(rp->samplesperbuffer()*rp->bytespersample()*file->
          numchannels()) );
      rp->set_bytespersecond( Round(rp->bytesperbuffer()/rp->secondsperbuffer()) );

   }
};

#endif
```

Listing A.22: `DeviceDataInterleave.hpp` – function for interleaving data streams between multiple USRP devices for GnuRadar. Note, this multi-device mode is not fully functional!

```
/***************************************************
 * Author: Ben Young (EEREU Summer 2013)
 * Date Created: 6/5/13
 * FileName: DeviceDataInterleave.cpp
 *
 * Summary: This file interleaves the elements of multiple separate
 *          vectors containing the same number of elements in the
 *          following pattern:
 *
 *          (Vector 1) Device 1: I1 Q1 I2 Q2 ... I1 Q1 ...
 *          (Vector 2) Device 2: I5 Q5 I6 Q6 ... I5 Q5 ...
 *
 *          Output Vector: I1 Q1 I2 Q2 ... I5 Q5 I6 Q6 ... I1 Q1 ...
 *
 *          Where the numbers following I and Q are channels.
 *          Each device can support four(4) channels max and one(1)
 *          channel min. A device cycles sequentially through its
 *          channels (I1 Q1 I2 Q2 ...) until it reaches its highest
 *          operating channel and then starts over.
 *
 * Note:     Currently this file only supports the interleavnig of a
 *           fixed number of devices (two(2), no more and no less).
 *
 *           If the file needs to be upgraded, the addition of
 *           NumChanDx, Datax, and Tposx variables is necessary,
 *           as well as some slight modifications to the state
 *           machine that interleaves the data.
 *
 *           The object of this file was to interleave 8 channels
 *           worth of data, where each channel contains 1000
 *           samples. Furthermore, the function must open, operate,
 *           and close, having returned the proper vector in under
 *           one(1) second.
 *
 *           The Boost cpu timer was inserted to make sure that this
 *          function can run at least 8 channels in less than one(1)
 *          second. To use the timer, uncomment the include and the
 *          boost::timer::auto_cpu_timer line and run. The output
 *          is automatic. To compile the code with the timer, the
 *          proper boost library must be linked in the command line.
 *          use: g++ -lboost_timer <program name> -o <output name>
 *
 ******************************************************/

#include <stdio.h>
#include <vector>
#include <stdint.h>
#include <stdlib.h>
#include <iostream>
#include <gnuradar/StreamBuffer.hpp>
#include <boost/shared_ptr.hpp>

//#include <boost/timer/timer.hpp>

#define DEVICE1    1
#define DEVICE2    2

typedef int16_t iq_t;

void* DeviceDataInterleave(void* Data1,
                                   void* Data2,
                                   std::vector<int> & BufferSize,
                                   std::vector<int> & chans){

    //code to benchmark the time it takes to complete this func.
        //FIXME: if this function is to be used then the boost_timer library is going to
            need to be linked.
        //In g++ this can be done with -lboost_timer but I am not sure how to do that in WAF
    //boost::timer::auto_cpu_timer t;

        int16_t* DataTemp1 = reinterpret_cast<int16_t*>(Data1);
        int16_t* DataTemp2 = reinterpret_cast<int16_t*>(Data2);

    int i; //counter
    int Tpos1 = 0, Tpos2 = 0, TposOut = 0; //iterator positions

    //state machine to handle merging the data vectors
    //together properly for a max of 8 channels. If
    //ever more devices are needed then the state machine
    //can be upgraded by simply creating more states and
    //then modifing the current states to accept the new ones.

    int DataStream = DEVICE1; //used to denote which state is the current state
    int VECT_SIZE = 0; //size of all the data combined
    int NumSampleBitsD1 = (chans[0])*2;
    int NumSampleBitsD2 = (chans[1])*2;

    for(int i=0;i<BufferSize.size();i++)
        VECT_SIZE += BufferSize[i];

        std::cout<<"DeviceDataInterleave: VECT_SIZE = "<<VECT_SIZE<<std::endl;

    //create output data
    int16_t DataOut[VECT_SIZE]; //initialinze output array to 0

    std::cout<<"DeviceDataInterleave: DataOut: size: "<<sizeof(DataOut)<<std::endl;

    do{
    switch(DataStream){
    case DEVICE1:
        for(i=0;i<(NumSampleBitsD1);i++)
            DataOut[i+TposOut] = (DataTemp1)[Tpos1+i];

        //increase position in vector so as to not
        //overwrite data just entered. Move 2*NumChanDx
        //elements because there are 2 samples per channel
        TposOut+=NumSampleBitsD1;
        Tpos1+=NumSampleBitsD1;
        DataStream = DEVICE2; //go to next state
        break;

    case DEVICE2:
        for(i=0;i<(NumSampleBitsD2);i++)
            DataOut[i+TposOut] = (DataTemp2)[Tpos2+i];

        TposOut+=NumSampleBitsD2;
        Tpos2+=NumSampleBitsD2;
        DataStream = DEVICE1;
        break;

    }
    }while(TposOut<VECT_SIZE);

        std::cout<<"Sample Data Stream From Device 1:"<<std::endl;
        for(int16_t i=0;i<10;i++)
                std::cout<<*(DataTemp1+i)<< "   ";
        std::cout<<"\n"<<std::endl;
        std::cout<<"Sample Data Stream From Device 2:"<<std::endl;
        for(int16_t i=0;i<10;i++)
                std::cout<<*(DataTemp2+i)<< "   ";
        std::cout<<"\n"<<std::endl;
        std::cout<<"Sample Interleaved Data Stream:"<<std::endl;
        for(int16_t i=0;i<10;i++)
                std::cout<<*(DataOut+i)<< "     ";
        std::cout<<"\n"<<std::endl;

        return (reinterpret_cast<void*>(DataOut));

}//end DeviceDataInterleave()
```

A.3 CIRI@PSU

Listing A.23: `crontab` for CIRI@PSU.

```
* * * * *       /home/radar/bin/rti_updater.sh
```

Listing A.24: `rti_updater.sh` – script for automatically updating link to latest RTI image and data run folder (used for basic website) for CIRI@PSU.

```
#!/bin/bash

IMFORMAT=".png"
DATADIR="/media/dataswap"
LINKDIR="/home/radar/public_html"
#EXCLUDE=$DATADIR/!(rtis|trash|processed|rawdata|events)

# 1. Enable extglob in the shell (this allows us to use ! as negation)
# 2. List the contents of 'output' folders in media dataswap (that aren't used by UIUC code)
# --> These are the folders that contain the RTI images and associated text files
# --> This list is time sorted with most recent file listed at the top
# 3. Pipe this to grep and get just the files that are images
# 4. Pipe this to head to get just the most recently modified image filename
# 5. Use 'find' to get the full path of this image file (this is kludgy and I don't like it)
shopt -s extglob
RTINAME=`ls -t $DATADIR/!(rti|trash|processed|rawdata|events)/output | grep $IMFORMAT | head
    -1`
RTI=`find $DATADIR/!(rti|trash|processed|rawdata|events)/output -name $RTINAME`
RTIDIR=`dirname $RTI`

echo ">>> Linking" $RTI "to "$LINKDIR"/rti_latest.png ..."
unlink $LINKDIR/rti_latest.png
ln -s $RTI $LINKDIR/rti_latest.png

echo ">>> Linking" $RTIDIR "to "$LINKDIR"/rtis ..."
unlink $LINKDIR/rti
ln -s $RTIDIR/ $LINKDIR/rti
```

Listing A.25: `index.html` – webpage for displaying latest RTI image for CIRI@PSU.

```
<html>
        <script type="text/javascript">
                var GB_ROOT_DIR = "http://aspirl.eradio.ee/~radar/greybox/";
        </script>
        <script type="text/javascript" src="greybox/AJS.js"></script>
        <script type="text/javascript" src="greybox/AJS_fx.js"></script>
        <script type="text/javascript" src="greybox/gb_scripts.js"></script>
        <link href="greybox/gb_styles.css" rel="stylesheet" type="text/css" />

<head>
        <title>CIRI@PSU Realtime RTI</title>
</head>

<body bgcolor="black" text="white">
    <h1><font color="yellow">CIRI@PSU Realtime RTI</font></h1>
    <b><font color="lightblue">Applied Signal Processing and Instrumentation Research
        Laboratory at Penn State</font></b>
    <hr>
    <p>The following range-time-intensity (RTI) plot shows a power map of the most recent
        data taken by the Cognitive Interferometry Radar Imager at The Pennsylvania State
        University (CIRI@PSU).  An archive of all of the RTIs from the most recent data run
        can be found <a href="./rti/">here</a>.  The current radar parameters are as follows
        :
        <ul>
            <li>49.8 MHz carrier
            <li>4 ms IPP
            <li>5 us baud
            <li>28 baud BPSK coding
            <li>30 kW peak power
        </ul>
    <p> CIRI Realtime <a href="http://aspirl.eradio.ee:8000" rel="gb_page_fs[]" >Temperature
         Monitor </a></p>
    <p>(refresh the page to update the plot below)</p>
    <p> <img src="./rti_latest.png"> </p>

</body>

</html>
```

A.4 General-purpose

Listing A.26: `sysNF.m` – MATLAB script to calculate cascaded noise figure and gain for RF front ends.

```
%% System Noise Figure Calculator
%  Alex Hackett
%  Fall 2011

% This script calculates system noise figure (and gain) using
% s_filt, s_att, and, s_amp objects.

close all; clear all; clc

%% Create system objects -- These are parameters to change!

% Theoretical values
% Filters
filt1 = s_filt( 1.8 );     % KR Electronics 2867
filt2 = s_filt( .2 );      % Minicircuits SIF-50+
filt3 = s_filt( .43 );     % Minicircuits SLP-70+
filt4 = s_filt( .01 );     % Minicircuits SIF-21.4+
filt5 = s_filt( .86 );     % Minicircuits SBP-21.4+
filt6 = s_filt( 4.3 );     % TTE KB8-49.8M-5M-50-720A

% Attenuators
sw1 = s_att( 1.1 );        % Minicircuits ZYSWA-2-50DR
att1 = s_att( 12 );        % Various Minicircuits attenuators
att2 = s_att( 18 );        % Various Minicircuits attenuators
att3 = s_att( 6 );         % Various Minicircuits attenuators
rflim1 = s_att( .04 );     % Minicircuits VLM-33-S+
mix1 = s_att( 6 );         % Minicircuits ZX05-1L-S+

% Amplifiers
amp1 = s_amp( 17, 2.9 );   % Minicircuits ZFL-1000LN+
amp2 = s_amp( 11, 3.5 );   % Advanced Receiver Research Broadband
amp3 = s_amp( 26, .5 );    % Advanced Receiver Research P49.92VDG
amp4 = s_amp( 22, .5 );    % Advanced Receiver Research P45VDG
amp5 = s_amp( 22, 5.7 );   % Minicircuits ZFL-500LN+

% ADC
adc1 = s_amp( 0, 34.9 );   % AD9862
adc2 = s_amp( 0, 21.0 );   % ADS64P44

% Bias Tee
tee1 = s_att( 0.5 );       % Minicircuits ZFBT-282-1.5A+

%% Define chain (input to output) as a cell array
%       -- These are parameters to change!

paris = {
           amp4;
           tee1;
           tee1;
           filt1;
           sw1;
           amp4;
           filt1;
           att3;
           amp4;
           att3;
           amp2;
```

```
            adc1
          };

ciri = {
            filt6;
            filt2;
            rflim1;
            amp5;
            filt3;
            mix1;
            filt4;
            filt5;
            amp5;
            adc2;
        };

% Change me to select the system
system = ciri;

%% Calculate system noise figure and gain
% Ftot = F1 + (F2 - 1)/G1 + (F3 - 1)/(G1*G2) + ...
% Gtot = G1*G2*G3*G4...
% NF = 10*log(Ftot) [dB]
% G = 10*log(Gtot)  [dB]

F_tot = 0;
G_tot = 1;

% Loop through and calculate cumulative noise factor and gain
for i = 1:length(system)

    if ( i == 1)
        % First F doesn't have -1 term
        F_tot = F_tot + system{i}.f;
    else
        % Calculate cumulative gain and noise factor
        G_tot = G_tot * system{i-1}.gain;
        F_tot = F_tot + ( system{i}.f - 1 ) / G_tot;

        if( i == length(system) )
            % Make sure to get the final gain
            G_tot = G_tot * system{i}.gain;
        end
    end
end

% Find the dB equivalents
NF = 10*log10(F_tot);
G = 10*log10(G_tot);

%% Display results
disp( '*** RESULTS ***' )
disp( [ 'Noise Figure : ', num2str( NF ), ' [dB] ' ] );
disp( [ 'Noise Factor : ', num2str( F_tot ) ] );
disp( [ '        Gain : ', num2str( G ), ' [dB] ' ] );
disp( [ '        Gain : ', num2str( G_tot ) ] );
```

Listing A.27: `s_amp.m` – simple amplifier model class for use with `sysNF.m`

```
%% S_AMP.M
%  Alex Hackett
%  Fall 2011

classdef s_amp
```

```
%  S_AMP  Simple amplifier model.
%     my_amp = S_AMP( gaindB, nf ), where gaindB is the gain
%     of the amplifier in dB and nf is the noise figure in dB,
%     creates an amplifier object to be used for system noise
%     figure calculation.

    properties
        gaindB = 0;      % Gain in dB
        gain = 0;        % Gain as ratio
        nf = 0;          % Noise figure in dB
        f = 0;           % Noise factor as ratio
    end
    methods
        % Default constructor
        function amp = s_amp( gaindB, nf )
            % Copy attdB and calculate other properties
            amp.gaindB = gaindB;
            amp.nf = nf;
            amp.gain = 10^( amp.gaindB / 10 );
            amp.f = 10^( amp.nf / 10 );
        end
    end
end
```

Listing A.28: `s_att.m` – simple attenuator model class for use with `sysNF.m`

```
%% S_ATT.M
%  Alex Hackett
%  Fall 2011

classdef s_att
%  S_ATT  Simple attenuator model.
%     my_att = S_ATT( attdB ), where attdB is the attenuation
%     (insertion loss) of the attenuator, creates an attenuator
%     object to be used for system noise figure calculation.

    properties
        attdB = 0;       % Attenuation (insertion loss) in dB
        att = 0;         % Attenuation (insertion loss) as ratio
        gaindB = 0;      % Gain in dB
        gain = 0;        % Gain as ratio
        nf = 0;          % Noise figure in dB
        f = 0;           % Noise factor as ratio
    end
    methods
        % Default constructor
        function my_att = s_att( attdB )
            % Copy attdB and calculate other properties
            my_att.attdB = attdB;
            my_att.gaindB = -attdB;
            my_att.att = 10^( attdB / 10 );
            my_att.gain = 10^( my_att.gaindB / 10 );
            my_att.nf = my_att.attdB;
            my_att.f = my_att.att;
        end
    end
end
```

Listing A.29: `s_filt.m` – simple filter model class for use with `sysNF.m`

```
%% S_FILT.M
%  Alex Hackett
%  Fall 2011
```

```

classdef s_filt
%  S_FILT  Simple filter model.
%      my_filt = S_FILT( attdB ), where attdB is the attenuation
%      (insertion loss) of the filter, creates a filter object
%      to be used for system noise figure calculation.

    properties
        attdB = 0;      % Attenuation (insertion loss) in dB
        att = 0;        % Attenuation (insertion loss) as ratio
        gaindB = 0;     % Gain in dB
        gain = 0;       % Gain as ratio
        nf = 0;         % Noise figure in dB
        f = 0;          % Noise factor as ratio
    end
    methods
        % Default constructor
        function filt = s_filt( attdB )
            % Copy attdB and calculate other properties
            filt.attdB = attdB;
            filt.gaindB = -attdB;
            filt.att = 10^( attdB / 10 );
            filt.gain = 10^( filt.gaindB / 10 );
            filt.nf = filt.attdB;
            filt.f = filt.att;
        end
    end
end
```

Appendix B

Preliminary Procedures

This appendix provides information on remote access, preliminary operations, and basic debugging of the PISCO, PARIS, and CIRI@PSU systems.

B.1 Remote Access

Although other methods exist, the simplest method of remotely connecting to the host machines for PISCO, PARIS, and CIRI@PSU is through the use of `ssh`. By adding the configuration in Listing B.1 to the local host's ~/`.ssh/config` files, the remote hosts can be accessed simply by:

```
$ ssh -Y <remote hostname>
```

and then entering the appropriate passwords.

Listing B.1: Full listing of ~/`.ssh/config` for remote access of PISCO (`coruscant` and `cadi`), PARIS (`kessel`), and CIRI@PSU (`zeltros`) machines. Note, `<USER>` should be replaced with a valid Arecibo Observatory network username.

```
Host coruscant
    User            radar
    Hostname        192.231.95.182
    ProxyCommand    ssh <USER>@remoto.naic.edu nc %h %p 2> /dev/null

Host cadi
    User            cadi
    Hostname        192.65.176.23
    ProxyCommand    ssh <USER>@remoto.naic.edu nc %h %p 2> /dev/null

Host kessel
    User        radar
    Hostname    aspirl.eradio.ee
    Port        33333

Host zeltros
    User        radar
    Hostname    aspirl.eradio.ee
    Port        55555
```

B.2 PISCO

This section describes the important directories and files, describes basic operation, and offers potential debugging solutions for PISCO.

B.2.1 Important Directories and Files

The following list explains the important directories and files on `coruscant` for running PISCO. Note, "∼" under Linux/Unix is equivalent to `/home/<user>/`, and in this case, specifically `/home/radar/`. The user is encouraged to be very familiar with system operations before making modifications.

- `~/pisco/` – Contains all the PISCO software.
- `~/pisco/Makefile` – Script to build PISCO software, using the command `make`.
- `~/pisco/bin/` – Contains executable programs.
- `~/pisco/bin/IonosondeRxRun` – Data collection program.
- `~/ionorun` – Script for automated operation of data collection, processing, and file renaming.
- `~/pisco/bin/plotter/` – Contains plotter scripts.
- `~/pisco/bin/plotter/main.py` – Main plotter program.
- `~/pisco/bin/plotter/hdf5_read.py` – Function to read tabular I/Q data from HDF5 files.
- `~/pisco/bin/plotter/iono_plotter_multi.m` – The bulk of the data processing, including groundwave detection, downconversion, decoding, and plotting.
- `~/pisco/bin/plotter/hdf5_write.py` – Writes the ionogram image back to the HDF5 data file.
- `~/pisco/config/` – Configurations directory. Contains a configuration file that lists the frequencies used by CADI, although this information isn't important anymore (relic from the host frequency adjustment configuration). All that's important is the number of frequencies listed in this file.
- `~/pisco/deps/` – Dependencies directory. Contains symbolic link to GnuRadar project.

- `~/pisco/fpga/` – Contains FPGA project files and bitstream image.
- `~/pisco/fpga/usrp1_iono_rx_300.rbf` – Symbolic link to the FPGA image modified for frequency sweeping operation.
- `~/pisco/fpga/usrp1/` – Directory containing all the project files for generating the FPGA image. Note, `coruscant` does not have the Altera Quartus II software to build the FPGA image – it must be installed on a local computer.
- `~/pisco/include/` – Contains header files for data collection program.
- `~/pisco/src/` – Contains code for data collection program.
- `/usr/local/bin/` – Contains symbolic links to the software necessary to run PISCO.
- `/usr/local/gnuradar/firmware/rev4/` – Directory in which GnuRadar looks for FPGA images and microcontroller firmware files.
- `crontab` – Schedule file for `cron` that allows automated operation of PISCO. Editable with the command `crontab -e`.
- `/data/` – Contains all HDF5 data files and generated ionogram images.
- `/data/rti_latest.png` – Symbolic link to the most recently generated ionogram (managed by `ionorun` script).
- `/data/data_latest.h5.bz2` – Symbolic link to the most recently collected compressed data file (managed by `ionorun` script).

The following list explains the important directories and files on `cadi` for running PISCO. Again, the user is encouraged to be very familiar with system operations before making modifications. WARNING: This system is relied upon for science operations, so consult with Arecibo Observatory staff before making ANY modifications, and avoid interrupting normal operations.

- `/root/ionoscheduler/` – Contains all the high-precision scheduling software for use of CADI with PISCO.
- `/root/ionoscheduler/Makefile` – Script to build scheduling software, using the command `make`.
- `/root/ionoscheduler/bin/` – Contains executable scheduler.

- `/root/ionoscheduler/bin/ionosched` – High-precision scheduler program that executes CADI so that operation is aligned with the top, 15, 30, and 45 minutes of every hour.
- `/root/ionoscheduler/deps/` – Contains dependencies for high-precision scheduler. Currently, only SThreads (`https://github.com/rseal/SThreads`) with some minor modifications for use with the very old version of `gcc` on CADI.
- `/root/ionoscheduler/include/` – Contains header files for high-precision scheduler.
- `/root/ionoscheduler/src/` – Contains code for high-precision scheduler.
- `crontab` – Schedule file for `cron` that operates CADI with high-precision scheduler for use with PISCO. Editable with the command `crontab -e` (as root user).

B.2.2 Operation

Nothing needs to be done to operate the system, as it is scheduled to automatically run with `cron`. To view the most recently generated ionogram, log into `coruscant` as described in Section B.1 and run the following command:

```
$ eog /data/rti_latest.png
```

Data files are date- and timestamped for convenience and can be transferred to another machine using the `rsync` or `scp` commands.

B.2.3 Debugging

The following list contains several potential solutions in the event of improper operation of PISCO.

- *Data files aren't being written to disk.* – Check to make sure the disk isn't full – at the current data rate of continued operation ($\sim$ 6.5 GB/day), the disk is expected to be full by the end of September 2013 (two months from the time of writing). If the disk is full, back up the data and ionograms on a separate disk and then remove them from

`/data/`. Alternatively, add a second "data" hard drive and mount under the mount point `/data/`.

- *Ionograms seem to be missing groundwave pulse, and thus aren't showing any ionospheric traces.* – Ensure that CADI is operational.

B.3 PARIS

This section describes the important directories and files, describes basic operation, and offers potential debugging solutions for PARIS.

B.3.1 Important Directories and Files

The following list explains the important directories and files on `kessel` for running PARIS. Note, "∼" under Linux/Unix is equivalent to `/home/<user>/`, and in this case, specifically `/home/radar/`. The user is encouraged to be very familiar with system operations before making modifications.

- `~/bin/rti_big.py` – Program to read and plot data from HDF5 files taken with GnuRadar.
- `~/sandbox/GnuRadar/` – Project directory for GnuRadar software package.
- `~/sandbox/uhd/` – Project directory for UHD software.
- `~/sandbox/gnuradio/` – Project directory for GNU Radio software.
- `~/gnuradar_configs/` – Directory containing GnuRadar configuration files.
- `/data/` – Directory containing mount point directories for "data" hard drives. At the time of writing, only `data0` is mounted.
- `/data/data0/` – Directory containing HDF5 data files written by GnuRadar.
- `/dev/shm/` – Shared memory directory (gives filesystem-type access to system RAM). Buffer files generated by GnuRadar's `ProducerThread` are stored here before the `Consumer` writes them to disk.
- `/usr/local/bin/` – Contains symbolic links to run GnuRadar software.
- `/usr/local/bin/gradar-configure` – GUI interface for generating GnuRadar configuration files.
- `/usr/local/bin/gradar-replay` – Program to emulate streaming data from USRP. Used for testing when a USRP and radar controller are not physically available.
- `/usr/local/bin/gradar-run` – Client GUI interface for sending command packets to server.

- `/usr/local/bin/gradar-run-server` – Server listener that receives and executes commands sent from `gradar-run`. This program handles all the USRP configuration, data collection, and data storage.
- `/usr/local/bin/gradar-verify` – Program to verify that the USRP trigger signal (generated by the radar controller) matches the GnuRadar configuration file. Note, this functionality is now built into `gradar-run` so there is no need to run this program separately.
- `/usr/local/bin/hdfview` – Third-party software for viewing HDF5 file contents. This program is useful in debugging of GnuRadar and the plotter.

B.3.2 Operation

The following steps describe the operation of PARIS:

1. Set up the radar controller
   (a) On the radar controller (not the host GPC), follow the on-screen instructions displayed after issuing the command `helpme`.
   (b) Define a new mode or modify an existing mode (*.hif file) using a text editor (e.g., `nano`) with the desired radar parameters. Generate the *.iif mode file.
   (c) Within the `bpg-shell` program, set the clock to standard input with the command `clock std`.
   (d) Add the mode defined using the `add mymode`, where `mymode` should be replaced by the previously generated mode, *excluding* the file extension.
   (e) Enable the outputs on the radar controller using the `start a` command.
   (f) Verify that the front-panel signal outputs are as expected using an oscilloscope.
2. Verify the pulsed RF signal is correct
   (a) Disconnect the cable attached to the "RF Input" port on the rear of the transmitter. Verify that the signal is indeed a gated RF signal, with peak-to-peak amplitude of ~6 V_{pp}.
   (b) Replace the cable on the back of the transmitter.
3. Start up the transmitter
   (a) Ensure that the transmitter's chassis is closed. There are sensors that will prevent operation if any panels have been removed. **NEVER** attempt to operate the transmitter with any panel removed.

   (b) Flip the "Radar TX" breaker switch on the wall to enable power to the transmitter.
   (c) Flip the transmitter's power switch at the bottom of its front panel.
   (d) Key in the code "F1E" to request state 1. Wait five minutes for the transmitter to warm-up. There is an internal timer that will not allow the transmitter to progress past state 1 until it has warmed up for five minutes.
   (e) Key in the code "F3E" to request state 3. At this point, the amplifiers are biased with high-voltage, but the input signal is not applied.
   (f) Key in the code "F4E" to request state 4. At this point, the input signal is applied to the amplifiers and the transmitter is transmitting. There should be an audible buzzing noise to indicate transmission (although for very low duty cycles or power levels, it may be difficult to hear over the sound of the fans and AC unit).

4. Start data collection
   (a) On the host computer, open a terminal and run the command `gradar-configure`. This will open up the configuration GUI. Load a previously created configuration or start a new configuration, with radar parameters as desired (matching those defined on the radar controller, of course). Ensure that the `usrp_trigger.rbf` FPGA bit image is selected. Save this file in `/home/radar/gnuradar_configs/` and quit the configuration GUI.
   (b) In the same terminal, run `gradar-run-server` to start the data collection server.
   (c) In a new terminal, run `gradar-run` to start the client GUI. Click "Load" and select the previously defined configuration file. Click "Verify" to ensure all settings match radar controller operation. Click "Run" to start the data collection
   (d) Optional: If desired, a basic realtime plotter (`gradar-plot`) can be run during data collection. In a new terminal, run `gradar-plot`. From the "File" menu, select "Connect" to begin plotting.

5. Stop data collection
   (a) When the experiment is done, stop data collection with the "Stop" button on the `gradar-run` GUI. If data collection is stopped any other way, data corruption WILL occur.
   (b) When the GUI indicates that data collection is complete, close the GUI.
   (c) Stop the server by issuing <CTRL>+C in the server's terminal.

6. Turn off the transmitter
   (a) Disable output on the transmitter by keying in the code "F3E" on the front panel.
   (b) Disable the transmitter's high-voltage power supply by keying in the code "F1E".

   (c) Flip the power switch on the transmitter's front panel.
   (d) Flip the "Radar TX" breaker switch on the wall to the OFF position.

7. Plot the collected data
   (a) Edit the parameters within the "EDITABLE PARAMETERS" section of the `main()` function of the Python script `~/bin/rti_big.py` as desired with a text editor. Choose a plotting start time, plotting end time, integration time and integration mode (if desired), channel index to plot (if multiple channels have been collected), and colorbar and range ranges. Close the file.
   (b) Run the script as follows, ensuring that the base filename specified after the `--base_filename` does not contain the HDF5 file index or ".h5" extension.

```
$ python ~/rti_big.py --base_filename /data/data0/mydataset_
```

   (c) Add the following flags to the command to modify the plotter's behavior:
      - `--verbose` – Output verbosely
      - `--batch` – Process the data in batch mode and output multiple images, each with a time length specified by the `--seconds` flag
      - `--seconds` – If running in batch mode, specify the number of seconds of data each output RTI image contains
   (d) For single-plot mode operation, the file will be outputted to an image called `rti_big.png`.
   (e) For batch mode plots, the plots will be named using the base filename of the data files along with the start time of each plot.

B.3.3 Debugging

The following list contains several potential solutions in the event of improper operation of PARIS.

- *Analog parameter error on transmitter (error code 4, 6, or 7)* – Check the heater/filament voltage (analog parameter 12 – key in "A12E") on the transmitter and ensure it is within the 9.8- to 10.7-V range, as this is the most common cause of error. If the voltage is out of range, turn off the transmitter and remove the power source. Open remove the left side panel, then adjust the potentiometer marked with white tape on the analog sensor PCB of the transmitter (when viewed from the front, top board on the left side of the transmitter – see Figure B.1). Clockwise rotation reduces the filament voltage, at a rate of around 0.5 V/turn (however, this appears highly nonlinear). Replace the side panel, start up the transmitter, and check the heater/filament voltage again.

- *High-voltage fuse error on transmitter (error code 5)* – Replace the high-voltage fuse wires found in rear of transmitter, according to the procedure described in the Tycho WPT-50 manual.

- *"Verify" step in* `gradar-run` *fails, with an error related to the number of samples, but the configuration matches what is defined on the radar controller.* – If the receive window defined in the GnuRadar configuration file (through `gradar-configure`) has been defined in units of "km" or "usec," sometimes a rounding error will trigger a failure in verify. The safest bet is to convert the receive window(s) to "samples" using the sampling rate, window ranges, and speed of light.

- `gradar-run` *opens up, but the "Verify" or "Run" stages don't work.* – Make sure that `gradar-run-server` is running in a separate terminal.

- `gradar-run` *opens up, but the "Run" stage produces an HDF5 access error.* – Either the data directory specified in the configuration file doesn't exist, or there is already a file with the base filename specified in the configuration file. Change the filename (or data directory) in the configuration file and reload the configuration file in `gradar-run`.

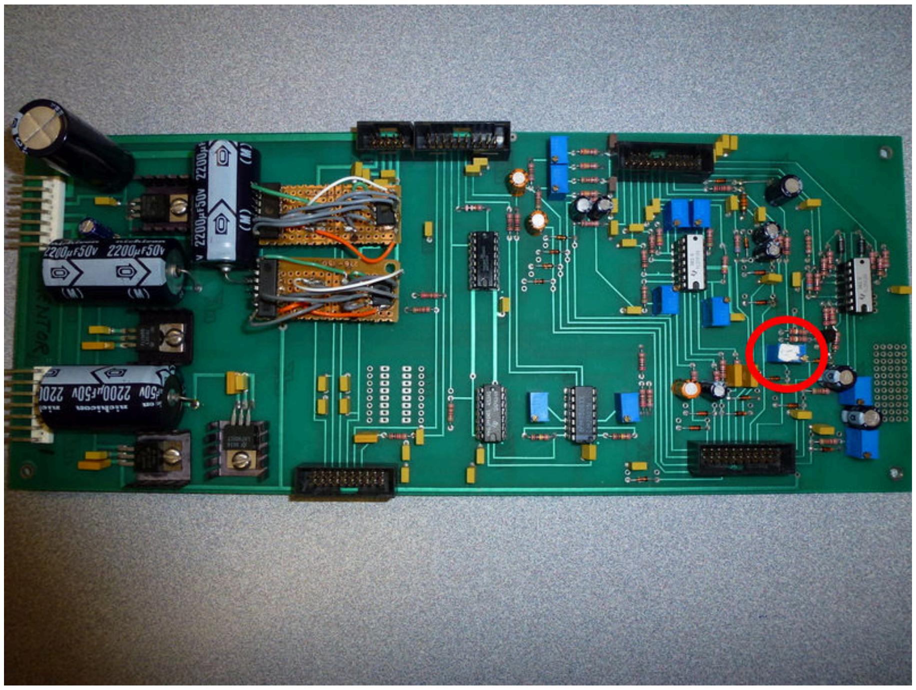


Figure B.1: Analog sensor board of the Tycho transmitter used with PARIS. Heater/filament voltage potentiometer circled in red.

B.4 CIRI@PSU

This section describes the important directories and files, describes basic operation, and offers potential debugging solutions for CIRI@PSU.

B.4.1 Important Directories and Files

The following list explains the important directories and files on `zeltros` for running CIRI@PSU. Note, "∼" under Linux/Unix is equivalent to `/home/<user>/`, and in this case, specifically `/home/radar/`. The user is encouraged to be very familiar with system operations before making modifications.

- `/bin/` – Contains several executable programs.
- `/bin/txcli.py` – Transmitter control interface program.
- `/bin/hv_enable` – Enables the output on the transmitter's high-voltage power supply.
- `/bin/hv_disable` – Disables the output on the transmitter's high-voltage power supply.
- `/bin/rti_updater.sh` – Script to automatically update `∼/public_html/rti` and `∼/public_html/rti_latest.png`
- `/Sauron_v1.7/` – Contains all Sauron-related code and basic documentation.
- `/Sauron_v1.7/Sauron.py` – Main Sauron executable program. Data collection parameters are configured within this file.
- `/public_html/index.html` – Basic webpage that shows the latest RTI image and has a link to the folder containing all RTI images from the most recent data run. Accessible from anywhere at `http://aspirl.eradio.ee/~radar/`.
- `/public_html/rti` – Symbolic link to the folder containing all RTI images from the most recent data run.
- `/public_html/rti_latest.png` – Symbolic link to the latest RTI image.[1]

[1] This technically isn't correct. The symbolic link is only updated once per minute. If Sauron is configured to display RTI plots shorter than one minute (default is 30 seconds), this symbolic link won't link to every image along the way. However, it's close enough for basic operation.

- `crontab` – Schedule file for `cron` that calls ∼`/bin/rti_updater.sh` script every minute. Editable with the command `crontab -e`.
- `/dev/shm/input/` – Directory containing raw data files captured by Sauron.
- `/dev/shm/processed/` – Directory containing *.mat files for each second of data for most recent 5 minutes, processed by the IRIS software.
- `/home/iris/IRIS/process4.py` – IRIS software data processor. Currently hard-coded for 4-ms IPP operation.
- `/home/iris/IRIS/imports.py` – Contains global directory variables for IRIS software. This file should be updated every time a new data run is made by Sauron.
- `/home/iris/public_html/realtime` – Python CGI for displaying the most recent five minutes of data processed by IRIS.
- `/home/iris/public_html/datascope` – Python CGI for displaying any five minute data set from the most recent data run.
- `/home/iris/public_html/index.html` – IRIS website that gives access to the `realtime` and `datascope` programs. Accessible from anywhere at `http://aspirl.eradio.ee/~iris/`.
- `/media/dataswap/` – Mount point for data hard drives.
- `/media/dataswap/<DATARUN>/` – Contains processed data (both Sauron and IRIS) from a particular data run (e.g., `June10_4ms_5us_28baud_gaussian/`) .
- `/media/dataswap/<DATARUN>/events/` – Contains *.mat files for each event detected by Sauron.
- `/media/dataswap/<DATARUN>/output/` – Contains RTI images and meteor detection text logs generated by Sauron.
- `/media/dataswap/<DATARUN>/processed/` – Contains *.mat files for each second of data generated by IRIS software.

B.4.2 Operation

The following steps describe the operation of CIRI@PSU:

1. Set up the radar controller
   (a) On the radar controller (not the host GPC), follow the on-screen instructions displayed after issuing the command `helpme`.

(b) Define a new mode or modify an existing mode (*.hif file) using a text editor (e.g., `nano`) with the desired radar parameters. Generate the *.iif mode file.

(c) Within the `bpg-shell` program, set the clock to standard input with the command `clock std`.

(d) Add the mode defined using the `add mymode`, where `mymode` should be replaced by the previously generated mode, *excluding* the file extension.

(e) Enable the outputs on the radar controller using the `start a` command.

(f) Verify that the front-panel signal outputs are as expected using an oscilloscope.

2. Start up the transmitter

(a) Turn off the enable switch on the GTS. Turn on the power switches for both the high-voltage power supply and the GTS.

(b) From the GPC, issue the following commands to turn on the HV power supply, disable transmitter output, set a pulse configuration, reset the transmitter, and enable the transmitter output (in software). Note, the pulse configuration specified below is for a 20-ms baud, Barker-7 coding, Gaussian shaped pulse, and maximum power output (see [45] for a full command description).

```
$ cd ~/bin/
$ ./hv_enable
$ python ./txcli.py -P /dev/iris_tx -c GTS_ENABLE 0
$ python ./txcli.py -P /dev/iris_tx -c GTS_SETPULSE 0x00 0x00 0x28
     0x01 0xFF 0x07 0xE4
$ python ./txcli.py -P /dev/iris_tx -c GTS_SAVPULSE
$ python ./txcli.py -P /dev/iris_tx -c RESET
$ python ./txcli.py -P /dev/iris_tx -c GTS_ENABLE 1
```

(c) Fully enable the transmitter output by flipping the enable switch on the GTS. If transmitting, the pulse count should be rising on the LCD displays of each PTM.

3. Start data collection

(a) Create a new data collection directory on the data hard drive. Within it, create three subdirectories, `output`, `events`, and `processed`. Change the permissions of `processed` to allow for group writing (i.e., `chmod -R g+w processed`).

(b) Create an `input` directory in `/dev/shm/`.

(c) Create a `processed` directory in `/dev/shm/`. Change the owner of this folder to the “iris” user (i.e., `chown -R iris:iris processed` as root).

(d) Change entries in `/home/iris/IRIS/imports.py` to match the new data run directory.

(e) Open up ~/`Sauron_v1.7/Sauron.py` in a text editor. Change the `OUT_DIR` variable to match the new data run directory. Change the `BAUD` and `CODE` parameters to match those defined in the transmitter pulse configuration. Close the file when finished.

(f) Run Sauron with the command `./Sauron.py`. Wait until data collection has started and the plotting window appears with streaming data.

(g) Open up another terminal and log in as the "iris" user. Start the IRIS processing software with the command ~/`IRIS/process4.py`.

(h) Keep both terminals open for the duration of the experiment.

4. Stop data collection

   (a) In the terminal running the IRIS software, issue the <CTRL>+C command to terminate the program.

   (b) Close all Figure windows associated with Sauron by clicking the "X" in the upper right corner. The Sauron software will close out the data files nicely and exit after a few seconds.

5. Turn off the transmitter

   (a) Flip the enable switch on the GTS to the off position.

   (b) From the GPC, issue the following commands to disable transmitter output and turn off the high-voltage power supply:

```
$ cd ~/bin/
$ python ./txcli.py -P /dev/iris_tx -c GTS_ENABLE 0
$ ./hv_disable
```

   (c) Flip the power switches for the GTS and high-voltage power supplies to the off position.

B.4.3 Debugging

The following list contains several potential solutions in the event of improper operation of CIRI@PSU.

- *The transmitter is not outputting even though the GTS enable switch is on and the* `GTS_ENABLE` *command has been sent.* – Check to make sure the TX Enable signal is connected and is logic high. The transmitter won't operate without all three conditions enabled.

- *Communications with either the high-voltage power supply or GTS failed.* – Repeat the command – sometimes there's some sort of hiccup that causes this. If that doesn't work, unplug both USB/RS-485 converters from the back of the host GPC, then wait fifteen seconds or so, and re-plug both. The delay is required to prevent Linux from reading the converters as new devices and allocating a higher device index for them.

- *Sauron is complaining about the disk being full but there's plenty of space left on the hard drive.* – Stop both Sauron and the IRIS processing software. Remove all files in `/dev/shm/input/` and `/dev/shm/processed/`. After several data runs, these partially completed raw and processed files can accumulate in `/dev/shm/` (where Sauron writes the raw data files to) and fill up the RAM. Upon restart of the computer, the directory will be cleared as well.

- *Some updates were applied to* `zeltros` *and now Sauron is not collecting data.* – This is most likely to due to updating a dependency of UHD or GNU Radio. Rebuild the UHD and GNU Radio packages (in that order) found in `/home/radar/` according to the instructions on the on their respective project websites (at the time of writing, these are `http://code.ettus.com/redmine/ettus/projects/uhd/wiki` and `http://gnuradio.org/redmine/projects/gnuradio/wiki`, respectively). If this still isn't working, ensure `boost` hasn't been upgraded to a "bad" version, according to GNU Radio.