Implementation of a Total Power Radiometer in Software Defined Radios

by

Matthew Erik Nelson

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering (Embedded Systems)

Program of Study Committee:

Phillip H Jones, Major Professor

John Basart

Mani Mina

Brian Hornbuckle

Iowa State University

Ames, Iowa

2014

DEDICATION

I would like to dedicate this thesis to my wife Jennifer who has stood by me through this long journey towards my Masters. I would also like to thank my parents without whose support I would not have been able to complete this work.

I would also like to thank my friends and family for their continued support and constant encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Phillip Jones for his guidance, patience and support throughout this research and the writing of this thesis. I would also like to thank Dr. John Basart who patiently spent many hours with me going over both the fundamentals and details with radiometers. I would like to thank Dr. Brian Hornbuckle for his continued support and for allowing me to utilize his radiometer, lab and students towards my research. Finally I would like to thank Dr. Mani Mina who not only was instrumental in my undergraduate career but continued his support into my graduate path as well. If it was not for Dr. Mina hiring me as the Chief Engineer of the Spacecraft Systems and Operations Lab, I would not be here today.

ABSTRACT

A software defined radio is defined as a communication system where components of a communication system that are typically done in hardware are now done in software. The result is a highly flexible communication system that can adapt to changes to the system based on requests by the user or due to conditions in the radio frequency channel. Software Defined Radios (SDR) have been used for a variety of applications, mostly in the area of communications. However, they have not been widely applied to remote sensing applications such as a radiometer. A SDR based radiometer offers a very flexible and robust system more so than a more traditional radiometer which has fixed hardware and usually little room for flexibility and adaptability based on the needs of the radiometer application or due to changes in the radiometers environment. The price of SDRs have also been steadily dropping making implementing them into radiometers a more attractive option compared to using traditional RF hardware. In this thesis we will look into the feasibility and theory of using an off the shelf SDR hardware platform for a radiometer application. In addition, we will look into how we can use the GNURadio software to create a total power radiometer within software and how this software can be used to make easy changes to the functionality of the radiometer. Finally we will look at the preliminary results obtained from laboratory tests of the SDR radiometer system.

CHAPTER 1. BACKGROUND

Software Defined Radios (SDR) have been used for a variety of applications, but their primary application has been in the area of communications. They appeal to applications where being able to change a modulation scheme or filter on the fly is desirable. In these areas, SDRs often outperform a traditional hardware only radio with their ability to rapidly change their operations by simply changing their software. Early SDR radios were expensive due to high costs in A/Ds needed and in the Field Programmable Gate Arrays (FPGA) used. In recent years however, the cost of SDRs have decreased due to the cost of these key components dropping in price. Even though the cost has gone done, the performance of many SDRs have increased. This has lead to new applications for SDRs being developed and using SDRs in new and different ways.

The basic concept behind a SDR is that it will digitize the RF signal as soon as possible. Once digitized, it can now be evaluated by a computer, FPGA, or a dedicated System on Chip (SoC). A canonical software radio architecture is one that consists of a power supply, antenna, multi-band RF converter, and a single chip that contains the needed processing and memory to carry out the radio functions in software [**?**)]. This allows us to extract certain hardware functions, such as filtering, into the digital domain which can then be manipulated by software. Since software is now manipulating the signal, we can rapidly change what functions we execute on the signal by changing the software. This gives SDRs a high amount of flexibility as various components that are normally done in hardware can now be done in software and can be changed by simply uploading new software to the system.

Radiometers are radio receivers that simply listen to and record the amount of power received. However, the power received is not a known signal, instead it is amount of noise the radiometer sees. Radiometers at the basic level, listen to noise that is generated naturally from a source. In our case, we will specifically be looking at the soil and the amount of power we receive correlates to how much moisture is in the ground. The amount of noise that is generated is due to the thermal agitation of the charge carriers, usually the electrons, which is directly correlated to the physical temperature of the source[?)]. This correlation is done as a noise temperature. The brighter or warmer the noise temperature is, the more RF noise that has been received which correlates to a drier soil. The less RF noise power received, the cooler the noise temperature and this indicates wetter soil area. Radiometers such as these are already in service on satellites such as the Soil Moisture and Ocean Salinity (SMOS) satellite launched by the European Space Agency (ESA) and are used by scientists to monitor the Earth's soil moisture and ocean salinity.

A traditional radiometer uses several Low Noise Amplifiers (LNAs) and filters and the power is often measured using a square law detector. Even though we are measuring noise, we want to measure the right noise and we want the noise generated by the radiometer hardware itself to be as low as possible. In other words, we are really listening to the noise being generated from an outside source. Additional noise in the system is impossible to eliminate, however we can take steps to reduce it as much as possible. In addition, we can calculate what this noise is and take steps to account for it in our measurements. However, this means that stability is another factor within the radiometer. If the noise generated within the radiometer is constantly changing, this makes it difficult to account for this additional noise. There are steps we can take to work with this though. A traditional method often used is the Dicke radiometer which switches between the measurement of the antenna and a known source. By referencing this known source the Dicke radiometer can calibrate and account for any drift due to variations in the system. Another method is to use highly stable components and keep them stable during the operation of the radiometer. For LNAs, temperature directly effects the overall gain from the LNA. In some radiometer applications we can control the temperature which allows us to keep the LNAs stable during the operation. It is this approach that the current ISU radiometer uses by using thermal electric coolers (TEC) that regulate the temperature of the LNAs to keep them stable.

Iowa State University currently owns a 1.4 GHz, dual polarization, correlating radiometer.

This radiometer is currently in use by Dr. Brian Hornbuckle and his research team. However, in recent years the ISU radiometer has not performed as expected and it was determined that the radiometer should be rebuilt using newer technologies.

The ISU Radiometer was built at the University of Michigan and put into service at ISU in 2006. The ISU radiometer is unique in that it is one of the few direct sampling radiometer. This radiometer takes the RF signal, amplifies and filters the signal, and then sends it directly to an analog to digital converter. At the time the ISU radiometer was built, an A/D that could sample accurately at 1.4 GHz were expensive and hard to come by due to the fact that Nyquist's theorem states that we must sample at least two times the frequency. However, the ISU radiometer does not sample at 2.8 GHz or above, instead it samples at 1.4 GHz. The reason the ISU radiometer can do this is due to the fact that a radiometer is generally only interested in the power of the incoming signal. This means that we are not interested in recreating the entire signal and therefore we can under-sample the signal. The A/D data is then sent to a Field Programmable Gate Array (FPGA) which then processes the data. The ISU radiometer is also a correlating radiometer, which means that it looks at both the V-polarization and the H-polarization and then correlates this information[**?**)].

The problem with this approach however is that we lose information. Although the general purpose of a radiometer is to measure power, we lose the ability to analyze in full detail other parts of the signal coming in. One such application is to know if we have any interference signal that could alter the received power being measured. Without having the frequency information, there is no way to tell if such a signal is present.

1.1 Introduction

The goal of this thesis is to explore other methods that could be used for a remote sensing radiometer. The end goal is to develop a radiometer that is more flexible than most radiometers and still maintain the accuracy and stability of a traditional radiometers if not exceed these specifications. A secondary goal was to use off the shelf components and components that are generally more accessible. This would allow radiometers to be more accessible to a wider scope of researchers in this field. And finally a tertiary goal was to ensure that the system as a whole is fairly easy to use. This ties to our secondary goal of making radiometers more accessible to a wider range of researchers and research topics.

As stated in the background, ISU currently owns a radiometer that is used for the remote sensing of soil moisture. In many ways, the current ISU radiometer is like a SDR. The RF signal is digitized and then processed by a FPGA to give the total power reading needed for radiometry. However, due to the under-sampling of the signal some information is lost. The ISU radiometer assumes that incoming signal is free of any interference and the power recorded is from the target of interest. However, in recent years the ISU radiometer has not been performing as expected, and without additional information it is almost impossible to diagnose these issues out in the field. The ISU radiometer is also not frequency agile, it is fixed at 1.4 GHz and the bandwidth is also fixed at 20 MHz. All of this means that if an interfering signal is present the current radiometer would not be able to determine this and even if it could, it has no means to try to avoid the signal.

This thesis looks to explore the following questions: (1) Can we use a SDR along with GNURadio to recreate a radiometer in software? (2) If so, what performance can we get from the system? (3) What benefits do we gain (if any) from using a SDR from a more traditional radiometer? The results of this research and experimentation are the subject of this thesis.

1.1.1 Software Defined Radios

Although a Software Defined Radio uses software to do the bulk of the signal processing, some hardware is still needed. Specifically hardware is needed to capture the signal and digitize it. A FPGA is often used to process the signal before passing it to a computer for further processing.

The equipment that was selected for researching into this topic was the Ettus Research Group N200 SDR. This SDR uses daughter boards as the RF front end to the SDR and up to two daughter boards may be installed into a N200. This is an important consideration as one of the requirements is to be able to look at both the V-polarization and the H-polarization signal coming from the antenna. This allows us to correlate the signal and other signal analysis can also be done. Another important reason the N200 was selected was due to the fact that it can handle up to 50 MHz of bandwidth to the computer. This means that it is very possible to have two receive cards that can stream up to 25 MHz bandwidth each.

The N200 utilizes a flexible architecture for a variety of RF interface systems based on the frequency range desired and if receive and/or transmission is needed. These daughter boards directly receive the RF signal and then outputs the analog I and Q signals that are then sampled by the N200 A/D converter for reception or receives the I and Q values from the N200 D/A converter for transmission.

The daughter board selected was the DBSRX2 card. This card is a receive only card that operates between 800 MHz and 2.4 GHz and would thus work in the 1.4 GHz band we are interested in. The DBSRX2 also has built in amplification that is adjustable through software.

1.1.2 GNURadio Software

For the software, we settled to use GNURadio, an open source software package that is well supported by the community and by the Ettus Research Group and the N200 SDR. GNURadio also comes with what is known as GNURadio Companion or GRC. This program provides us with a GUI interface and allows for the drag and drop of blocks that represent certain functions that can be used with the SDR. GNURadio and GRC use Python as it's main scripting language and GNURadio uses C++ code for directly accessing the hardware.

GNURadio has been ported to several platforms including Windows, Mac OS X and Linux. Linux is by far the most popular platform to work on and most of this thesis research was done within the Linux environment. Testing was also done though with a MacBook Pro running OS X 10.8. Although GNURadio would compile and run on OS X, there were several issues with GRC and some graphs would not display properly under OS X.

Through GNURadio, we can now write code that will get take the data given to us from the SDR and manipulate the signal as we need to mimic a radiometer. The power detection, filtering and recording of this data is all done through GNURadio. This also means that we are shifting more of the computational power done on the signal from the FPGA to a computer running GNURadio. This is different from the current ISU radiometer where the FPGA did almost all of the work and then simply sent the data to a computer to be stored. The benefit of this however is that components of the radiometer can be updated by simply changing things in GNURadio and does not require reprogramming the FPGA. It does however mean that the host computer must be powerful enough to handle the signal.

1.1.3 Current ISU Radiometer

The current ISU radiometer was designed and built by the University of Michigan and was designed for soil moisture radiometry at 1.4 GHz. It is a digital direct sample radiometer that directly samples the incoming RF signal after filtering and amplification. What makes the ISU radiometer unique is that the A/D converters are not fast enough for a 1.4 GHz signal. This is actually done on purpose and is part of the design. With radiometers, we are generally only concerned with power of the incoming signal. Therefore, the current ISU radiometer can undersample the signal as the only information we need from the signal is the power information. The ISU radiometer is also a correlating radiometer, where both the vertical and horizontal polarization from the antenna can be correlated.

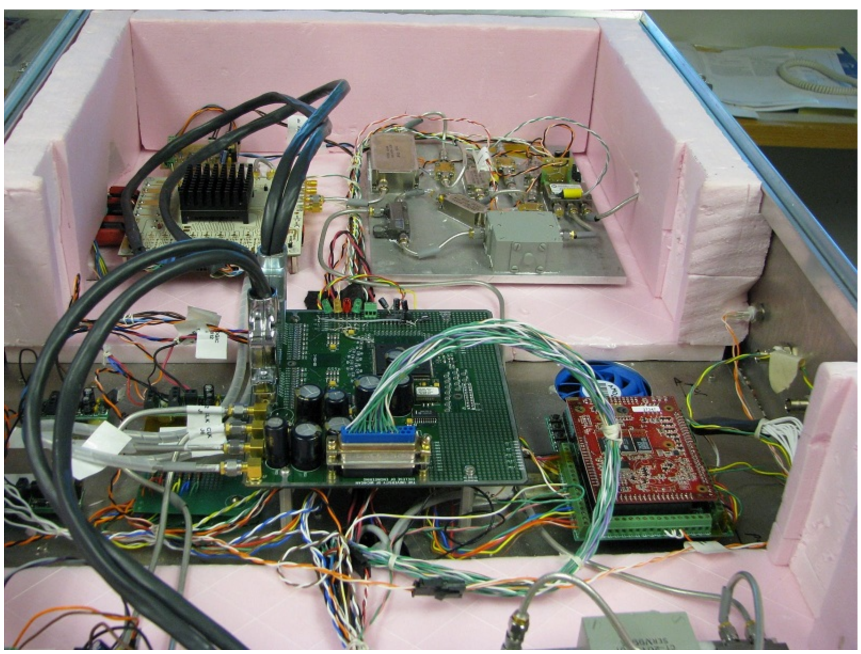

Figure 1.1 Inside the ISU Radiometer showing the FPGA, micro-controller and one of the RF front ends

The current ISU Radiometer uses a 3 stage Low Noise Amplifier (LNA) to amplifier the noise while keeping the noise contributed to the system as low as possible. As with any radiometer, the first LNA is the most critical as it contributes the most to the overall system noise temperature. For this reason the University of Michigan used a LNA that did not have a large gain but had a very low noise figure. The second and third LNA has higher gain values at the cost of a higher noise figure. However, since they are further down the chain, they do not contribute as much to the total system noise. The reason for this is further explained in chapter 3.

A driving force behind this thesis research was due to the fact that the digital side of the ISU radiometer was not functioning. Problems were developed with the FPGA board which resulted in the board being sent back to the University of Michigan several times. In addition, the micro-controller that stored the data from the FPGA and also controlled various functions within the radiometer also started to malfunction. In the end, the radiometer was deemed unusable and it was determined that this would be a good time to re-evaluate the current radiometer and see what improvements could be done.

Since the current radiometer was very much like a SDR, it was determined to look into the feasibility of using a commercial off the shelf SDR and using it in a radiometer application. To our knowledge this has not been attempted before and it was determined this would serve as a good thesis platform.

It was assumed in the beginning that the RF front end of the existing radiometer was still functioning properly. Additional tests on the radiometer did show some intermittent problems which did result in some complications in testing. These will be explained in more detail later in this thesis. However, the use of the SDR did help in seeing some of these problems which added an additional benefit to using a SDR to the more traditional radiometers.

1.1.4 Related Works

As mentioned before, software defined radios have been used in a number of applications. While as far as the author has determined, this is the first application of using an off the shelf software defined radio for a radiometer in remote sensing, there has been similar applications done. The closest application that has been found is with radio astronomy. There are many similarities between radio astronomy and remote sensing of the ground. Both are using a radiometer to listen to a source of interest. In radio astronomy the basic principal is that a ”hot” source such as a star will produce more noise than the cooler background of space. In remote sensing we are looking at the overall change of the source to determine it’s characteristics. In both cases we are measuring the total power of the noise and based on that information we can determine some properties of the source we are looking at.

The Shirleys Bay Radio Astronomy Consortium (SBRAC) located in Smiths Falls, Ontario is currently using a USRP software defined radio in conjunction with GNU-radio. SBRAC has successfully used this configuration to obtain radio astronomy data by looking at the hydrogen line at 1420.4058 MHz [**?**)]. The person in charge of this facility, Marcus Leech, contributed software to the GNURadio specifically for radio astronomy applications. It was this software branch that was used as the base for the GNURadio program that was used in this thesis.

It should be noted that much of the related works found worked with using a radiometer for astronomy or for looking at the sky. For the ISU radiometer however, we are looking at the ground and we are using the radiometer for soil moisture instead of measuring stars and other points of interest in the sky. While the fundamentals is the same for either radiometer, some adjustments need to be made due to the fact that a radiometer looking at the sky often sees a ”cool” brightness temperature whereas a radiometer looking at the ground sees a much ”warmer” brightness temperature. This is due to the albedo of the Earth and the fact that it has a much warmer noise temperature then what you find with radio astronomy.

CHAPTER 2. THEORY OF OPERATION

The work of this thesis is to use a software defined radio (SDR) to replace the bulk of the digital components used on the ISU radiometer. In many ways, the existing ISU radiometer is very much like a SDR in that it employed an A/D converter and FPGA to process the signal. However, these devices were designed specifically to look at the overall power of the incoming signal. It was desired that we have the capability to look at the full signal coming so that a more in depth analysis can be performed. For this reason, a new approach is devised to use off the shelf hardware and the logical choice was an off the shelf SDRs which can offer similar performance while expanding on the functionality of the radiometer. The advantages of being able to look at the whole signal instead of just power will be discussed later in this thesis.

The hardware that was selected for this research was the Ettus Research N200 Software Defined radio. Information and rationale behind the selection of this unit will be covered later. The N200 however gives us the standard building blocks for a typical software defined radio which includes a A/D converter and on-board FPGA. The N200 however also gives a flexible front end by selecting various daughter boards that fit our application. In addition, the N200 supports up to two daughter boards to be installed.

A software defined radio of course is only as good as the software that is written to work with the hardware. As the name implies, the software defines how the radio will act and function. From a computer engineering perspective though, software can take on several roles in a system. There is software that runs on the hardware, in our case the FPGA, and then there is software that may run on a computer to interface with the hardware. In some cases, the software may only reside on the hardware, however, to increase the user usability of the system, we will be using a combination of firmware that is running on the FPGA and software that runs on a computer. GNURadio is an open source software define radio framework that runs on multiple OSes and offers a rich set of features. In addition, GNURadio is well supported by the Ettus Research group and is the preferred software for interfacing with their hardware. An easy to use interface was another driving requirement for our implementation of a radiometer in a SDR. GNURadio helps us with this through the use of the GNU Radio Companion or GRC. This was important as it was anticipated that many operators of the radiometer would not know much about programming. GNURadio uses a simple to use graphical system that is very similar to things such as LabView, offering a drag and drop system for adding various radio components such as filters. Like LabView, you can also simply wire up the boxes and complete the circuit path for the RF signal as it gets processed.

2.1 Radiometer Theory of Operation

The primary goal of a radiometer is to measure power. While that statement sounds easy, there are in fact many factors that go in to how well a radiometer can measure the power it sees. A better statement would be that a radiometer's primary goal is to accurately measure power within a certain degree of accuracy. In order to accurately and within a high degree of precision measure power, a radiometer must take into account various factors such as the system noise, the bandwidth of the signal and the stability of the system as a whole.

2.1.1 Measuring RF power

To measure power in a radiometer, several factors are taken into consideration. To begin with we have the noise signal coming from the antenna. Our antenna is assumed to be looking at our target of interest and it is assumed that we can relate the antenna noise to the noise from the source. It is often easier to refer to this noise as the brightness temperature. Therefore the brightness temperature of the source can be related to the brightness temperature at the antenna. We will refer to this brightness temperature as T_A.

Figure 2.1 shows us an ideal radiometer. That is a radiometer that has an input from the antenna, T_A, a known bandwidth denoted as B and a known gain denoted as G. At the end of the block is the detector, which measures the power from the radiometer.

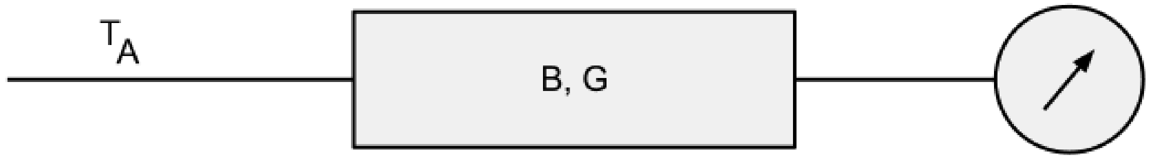

Figure 2.1 The ideal radiometer block diagram

Only a certain selection of the radio spectrum is observed by the radiometer. This is referred to as the bandwidth of the radiometer and is denoted as B or as β. This bandwidth is then centered around a center frequency. In our case, we center around 1.405 GHz. There is a reason why 1.405 GHz is selected. The range from 1.4 to 1.426 MHz is protected internationally. This reduces interference from outside sources such as transmitters that can interfere with the operation of the radiometer.

The power coming from the antenna is amplified so it is easier to determine changes in the brightness temperature. The overall gain of the radiometer system is referred to as G in this case. Finally, we need to apply Boltzmann's constant, referred to as k. With these values, we can now compute the power the radiometer will see for an ideal radiometer. This can be shown in equation 2.1

$$P = k * \beta * G * (T_A) \tag{2.1}$$

However, since we do not have an ideal radiometer, we have another key component that needs to be addressed. This is the noise that is added to the system by the radiometer itself, primarily from the amplifiers used to increase the signal. Most of the additional noise is from the Low Noise Amplifiers (LNA) that are used to increase the signal while attempting to keep the noise added to a minimum. However, noise is also added from virtually every component in the RF front end. However, by far the largest contribution usually comes from the LNA, which is why the selection of the LNAs is a critical decision. Figure 2.2 shows the additional noise that is injected into the system.

As it can be seen, this additional noise is added to the noise coming from the antenna source. Therefore T_N is added to T_A and our final equation for the power measured is shown

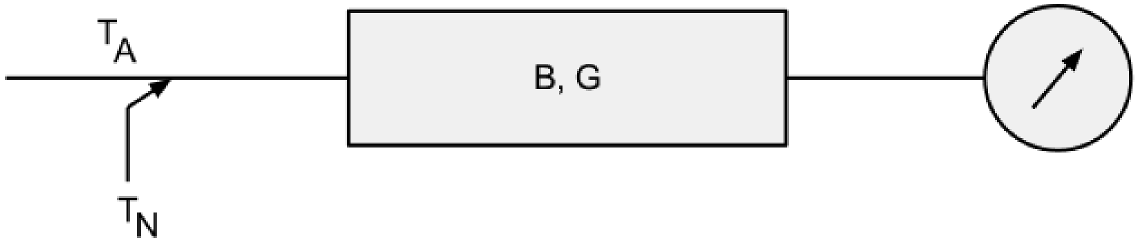

Figure 2.2 A more realistic radiometer model

in equation 5.1.

$$P = k * \beta * G * (T_A + T_N) \tag{2.2}$$

The issue with all radiometers is that it must detect small signal changes in a noisy environment. To understand this, let us look at the example of T_A has a value of 200 K and T_N has a value of 800 K. Since T_N is added to our antenna signal, we have a total noise temperature of 1,000 K. This means that if we want to detect a change as small as 1 K, we must be able to measure the difference between 1,000 K and 1,001 K. [?)]

The ability of a radiometer to detect these small changes is the radiometer's sensitivity, or the standard deviation of the output signal from the radiometer. This sensitivity is also referred to as the Noise Equivalent Δ Temperature or NEΔT.

$$NE\Delta T = \frac{T_A + T_N}{\sqrt{\beta + \tau}} \tag{2.3}$$

The sensitivity of the radiometer is based on both the bandwidth, β, of the incoming signal and the integration time, τ. As it can be seen in the equation, we would want to have as much bandwidth as possible. In a traditional radiometer, this bandwidth is often fixed and is dependent on the band-pass filters used in the radiometer. We can however control τ and a longer integration time will help improve the sensitivity of the radiometer.[?)]

This covers a very simplified radiometer, however most radiometers are more complicated than the one shown in Figure 2.2.

A more typical radiometer can be shown in Figure . Here we have expanded the blocks used and have separated the bandwidth and gain blocks. In addition, we have detection, shown as X^2, and integration. These last two blocks make up the detection and results in the power we can measure, which is a voltage represented as V_{out}. This results in equation 2.4.

$$V_{OUT} = c * (T_A + T_N) * G \tag{2.4}$$

Here V_{OUT} is shown by the addition of both the noise from the system T_N and the noise from the antenna, T_A and multiplied by the gain in the system, G.

2.2 Software Defined Radio Theory of Operation

A software defined radio (SDR) attempts to mimic radio functions in software instead of relying on dedicated hardware. As stated in the previous section, a radiometer is a radio that can detect changes in power. Therefore the SDR needs to be able to measure power coming from the source that we are looking at.

The SDR is able to do all of this but in a slightly different way than a traditional radiometer. A traditional radiometer will use a device called a square-law detector to measure the incoming RF signal. This device is simply a diode where the input voltage is squared and the output from the diode is proportional to the AC input voltage. Therefore a 3 dB increase in the RF power will result in a two times increase in the voltage. This power measurement will be fluctuating rapidly, therefore we will then run the output from the square-law detector to an integrator and integrate over a set time period. As shown in equation 2.5, this integration time also affects the NEΔT or sensitivity of the radiometer as well.

For the SDR, the incoming signal is sampled and converted to IQ values. The IQ values represent the amplitude and phase information of the signal. In GNURadio we are then able to square these values within software. This block in GNURadio mathematically performs the following:

$$I^2 + Q^2 = P_{out} \tag{2.5}$$

Like the analog square-law detector, this signal will fluctuate rapidly and to improve the sensitivity of the radiometer we wish to integrate this signal. A RC filter is analogous to an integrator where the R and C values determine our time constant and our integration time for the filter. A SDR however operates in the digital domain at discrete intervals. One type of filter that can be used in the Infinite Impulse Response (IIR) filter.

To begin with, we look at what an analog RC filter looks like.

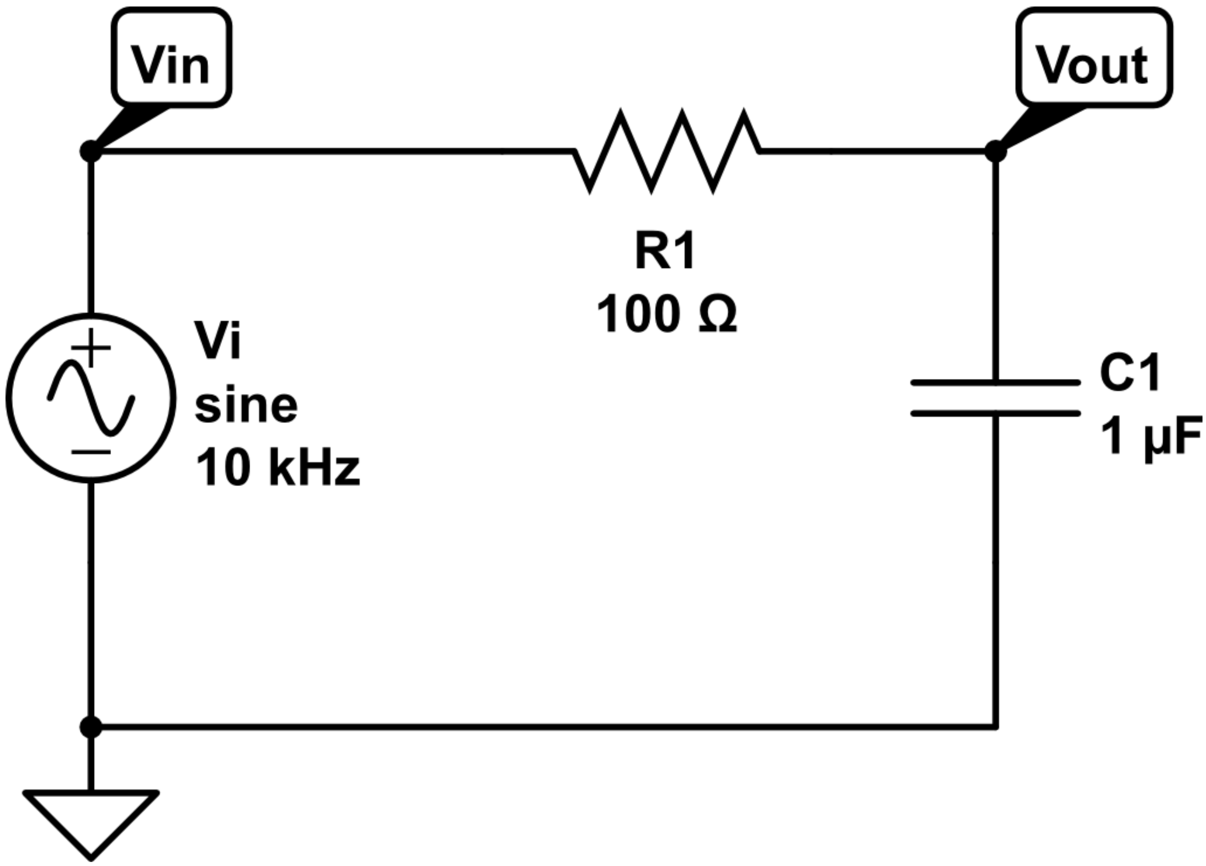

Figure 2.3 A simple RC circuit

This circuit can be represented by the following equation.

$$\frac{V_{in} - V_{out}}{R} = C\frac{dV_{out}}{dt} \tag{2.6}$$

A Finite Impulse Response (FIR) filter is a digital filter that can take an impulse signal and decays to zero after a finite number of iterations. This type of digital filter can be represented by the following equation:

$$y_n = \sum_{i=o}^{P-1} c_i x_{n-i} \tag{2.7}$$

This simply says that the nth output is a weighted average of the most recent P inputs.

An Infinite Impulse Response (IIR) filter is the same as the FIR filter, except that we add an additional summation term which feeds back the previous output.

$$y_n = \sum_{i=o}^{P-1} c_i x_{n-i} + \sum_{j=1}^{Q} d_j y_{n-j} \tag{2.8}$$

It can be seen that a FIR filter is really a IIR filter except that $Q = 0$.

To get a better understanding on how our digital IIR filter relates to our RC filter analog, we can look at the Fourier Transform and the relationship of the input to the output in the frequency domain.

$$H(f) = \frac{\sum_{j=o}^{P-1} c_j e^{-2\pi i j f T}}{1 - \sum_{k=1}^{Q} d_k e^{-2\pi i k f T}} \tag{2.9}$$

Here f is our frequency in Hz and T is the time between samples in seconds.

We now want show the link between our analog RC circuit and the IIR filter. Looking at equation **??**, which represents the differential equation relating the input voltage V_{in} to the output voltage V_{out}, we can substitute for input and output of our IIR filter. Since we are now in the time domain, we need to define what T is.

$$T = timebetweensamples = \frac{1}{samplingrate} \tag{2.10}$$

We can now relate our input voltage to the input to our IIR filter and the output voltage to the output of our IIR filter.

$$x_n = v_{in}(nT) \tag{2.11}$$

$$y_n = v_{out}(nT) \tag{2.12}$$

We can now rewrite our difference equation with x_n and y_n.

$$\frac{x_n - y_n}{R} = C\frac{y_n - y_{n-1}}{T} \tag{2.13}$$

Now, we can solve for y_n which results in our final equation for showing how a IIR filter is related to an RC filter.

$$y_n = \frac{T}{T + RC}x_n + \frac{RC}{T + RC}y_{n-1} \tag{2.14}$$

It can be seen that an IIR filter can have the same frequency response as we would expect from an analog RC filter. As our sampling rate approaches infinity, the approximation gets closer to the original response from the analog RC circuit.

For the cutoff frequency of a RC circuit, we know that it has the following relationship.

$$f_c = \frac{\sqrt{3}}{2\pi RC} \rightarrow RC = \frac{\sqrt{3}}{2\pi f_c} \tag{2.15}$$

The RC term gives us our time constant of the circuit and can be used to calculate out our coefficients. We are not concerned about the actual values of R and C with our IIR filter, instead we just need the product of R and C.

For GNURadio most of the work is done for us. We can simply enter in our desired cutoff frequency and GNURadio will calculate our IIR filter coefficients. However, this shows that an IIR filter works very much like an analog RC low pass filter.

Like a traditional radiometer, the SDR will use an antenna to look at the target of interest. SDRs still use a RF stage that takes the power from the source and amplifies it. The difference though begins after that. A SDR will then sample and generate I and Q values that represents the amplitude and phase of the signal. From there, this data is sent to a computer to be processed. We can then use this information to calculate the power that is being seen. In addition, we can manipulate the signal in other ways such as applying a filter to filter out an unwanted source.

The N200 is a SDR developed and built by Ettus Research. It is one of the newer models in the companies USRP line of SDRs. It offers several key features that were desirable for a radiometer type of application while still being an economical option. It also offered a highly flexible architecture which will allow this radio to be up-gradable in the foreseeable future.

Figure 2.4 The USRP N200 from Ettus Research

2.2.1 General Specifications

The N200 has the following features that made it well suited for our specific application.

- Dual 14-bit ADC
- Dual 16-bit DAC
- 50 MS/s Gigabit Ethernet streaming
- Modular daughter-board system for RF front end

These specifications had a large impact on the selection of the N200 for this application. Specifically the 14-bit ADC, the 50 MS/s and the modular daughter-board system were the largest factors in the decision to use the N200 SDR. Further explanation on these specifications are explained in the following sections.

2.2.1.1 14-bit ADC

The analog to digital converters (ADC) allow us to take the analog I and Q values from the daughter boards and digitize this information. Once digitized, we can now work with the signal both on the on-board FPGA board or stream it to the computer so that our software can manipulate the signal. In radiometry, we are primarily looking at the overall power of the signal and this does not require us to accurately recreate the signal. However, it will be discussed in this paper in further detail that there are times where being able to recreate the signal for additional analysis can be quite beneficial. For that reason, the resolution of the ADC becomes more important.

2.2.1.2 50 MS/s Bandwidth

For this application, the N200 is required to receive a signal at 1.4 GHz and is at least 20 MHz wide. Bandwidth plays an important role in remote sensing and to the amount of power that we receive. It also plays a key role in the sensitivity of the radiometer. To begin, an examination of how much power that is received and the correlation to the bandwidth will be looked at.

$$P = k * \beta * G * (T_A + T_{sys}) \tag{2.16}$$

Here we can see that the bandwidth of the signal received directly relates to the amount of power the radiometer will measure. The higher the bandwidth the more power the radiometer will see.

Equally important however is the the sensitivity or how small of a change the radiometer can detect. This sensitivity or NEAT function is shown in equation 2.2

$$NE\Delta T = \frac{T_A + T_{sys}}{\sqrt{\beta + \tau}} \tag{2.17}$$

Again, the amount of bandwidth the radiometer receives plays a large part in the performance of the radiometer.

Another factor however is the amount of bandwidth the existing RF front end of the ISU radiometer is able to provide to us. The current filters on the ISU Radiometer keep the bandwidth of the signal to 20 MHz. This meant that at the very minimum a 20 MS/s bandwidth from the SDR is required.

The N200 is capable of working with up to 100 MS/s signal and can stream up to 50 MS/s through the Gigabit Ethernet connection. The N200 also has the ability to have up to 2 daughter boards installed. If we assume each will have up to a 20 MHz signal, this means up to 40 MS/s of data will be required. This means that the N200 meets are minimum requirements for working with needed bandwidth of the radiometer. In addition, the FPGA on the N200 is capable of working with up to 100 MS/s, so there is room handling additional bandwidth by using the on board FPGA to process the signal.

2.2.1.3 Daughter Boards

The daughter boards allow for easy replacement of the RF front end to the software defined radio. Ettus Research makes a number of daughter boards that range from a a wide range of frequencies and is available with transmitters, receivers and transceivers. Because these boards are modular and do not touch the analog to digital converters or the on-board FPGA, very little change is required in the software. The daughter boards however do have the required RF hardware for the signal to be processed. In this application it was required that the signal was detected at 1.4 GHz with a bandwidth of 20 MHz. The DBSRX2 receiver met this requirement and was selected to be used with the N200. In this radiometer application transmission is not needed and is in fact illegal in the 1.4 GHz band, which is reserved for radiometer applications.

2.2.2 The DBSRX2 Receiver

The DBSRX2 receiver board is capable of receiving signals between 800 MHz and 2.3 GHz. The board is able to plug into one of expansion slots available on the N200 that we are using.

The DBSRX2 has an onboard Programmable Gain Amplifier (PGA) that is accessible from the software and allows us to control the gain through GNURadio.

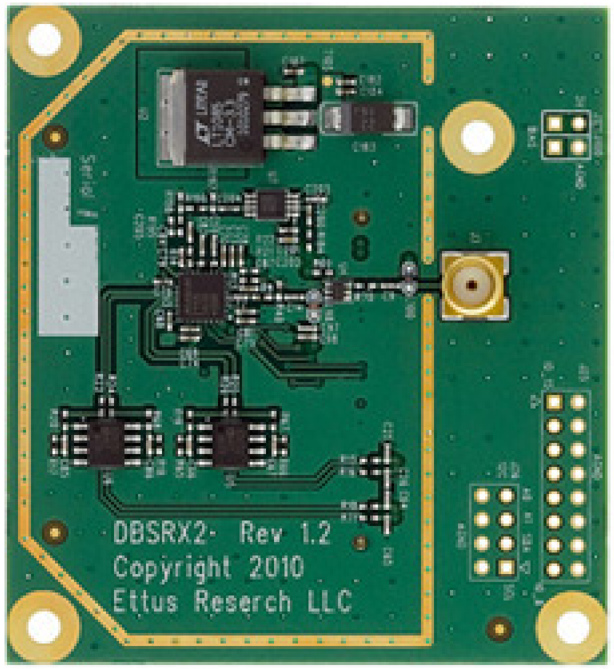

Figure 2.5 The DBSRX2 daughter board from Ettus Research

2.3 GNURadio

GNURadio fills in the software side of the software defined radio. Although there is firmware that runs on the FPGA in the N200, this firmware is designed to communicate with a host PC. It is this software that does most of the work in terms of the calculations that are done with the signal. The FPGA simply sends the raw IQ data to the host PC, which then performs the necessary math functions. Again, the reason why software defined radios are desirable is the ability to change the behavior of the radio very quickly. In our case we can change functionality by simply loading a new software program in the host PC.

This functionality is ideal for communication type of radios where different modulation schemes and encoding and decoding methods can easily be changed out. However, in a radiometer we are not interested in this aspect of the SDR. However, one functionality is available that can be very valuable for a radiometer, and that is with filtering. Although we often use frequencies that should be free from interference, this is not always the case. Interference can and often does still occur, even in these protected frequencies. With the SDR, we are able to quickly adapt to changing conditions by moving the frequency, changing our bandwidth and even filter out an offending signal.

GNURadio was selected as it is an open source software platform. GNURadio is licensed under the GPL license and has a strong community that continually updates the software. It is also well supported by third parties such as Ettus Research Group, National Instruments and other SDR developers. In addition GNURadio has a strong set of tools that can be used to develop programs that run under GNURadio. Tools such as the GNURadio Companion (GRC) allows for an easy to use GUI to develop code for GNURadio. GNURadio is also written in Python, which allows for easy modification and access to additional tools that can be used with GNURadio.

CHAPTER 3. A GNURadio AND N200 TOTAL POWER RADIOMETER

One of the principal goals with this research was to implement a fully functioning total power radiometer within software. The N200 provides us the link between the RF signal captured by the antenna and converts that to a format that the computer can now use to manipulate the signal. Once the signal has been passed to the computer, GNURadio will implement the correct algorithms to detect the power within the signal. One of the advantages of course with a SDR is that filtering can also be done within the software. In addition, thanks to the WxGUI that GNURadio uses, we can also build a user interface that can control several key variables that are useful for us. This includes controlling the gain on the programmable gain amplifier on the DBSRX2 card, the sampling or bandwidth of the signal, the center frequency, and also the integration time. All of these can now be controlled in real time as well. GNURadio will also store the power data so that we are able to do further analysis of the data using a software program like MatLab. Because GNURadio is a very flexible system, much more can be done with the signal if needed and future updates may add more capabilities or additional analysis on the signal can happen.

3.1 Requirements

Requirements for this system was based largely on what the current ISU radiometer was capable of doing and was also based on requirements of a typical radiometer. These requirements are outlined in table **??** below.

3.1.1 Hardware Requirements

The selection of the N200 SDR from Ettus Research was based on many of the requirements outlined in table **??**. In addition, we wanted the hardware to be flexible but also affordable.

Table 3.1 ISU Radiometer requirements

Requirement	Value	Units
Frequency Range	1400 - 1425	MHz
Bandwidth	25	MHz
Polarization	Dual	
Sensitivity	-30	dBm
Accuracy	1	Kelvin

There are many different kinds of software defined radios on the market. However, we choose the N200 based on the availability of the device, the large community support especially with regards to support by GNURadio and because it meet and often exceeded the requirements stated above.

Flexibility was another key aspect of the N200 that made it an excellent selection for our project. The N200 uses a daughter board setup for configuring the RF interface to the rest of the electronics. Several different daughter boards are available that have different frequency ranges and offer both receive, transmit and transceiver designs. For the radiometer we need to operate around 1.4 GHz and receive only. Based on those requirements we choose the DBSRX2 daughter board. This board is designed from 800 MHz to 2.3 GHz and has a fairly low noise figure.

The current RF front end to the radiometer is designed for a 20 MHz wide signal. This requirement was one of the main driving points for selecting the N200 SDR as it can support up to 50 MHz in bandwidth between the N200 and the host computer. This is accomplished by using a 1 Gbps Ethernet connection between the N200 and the host computer.

3.1.2 Software Requirements

The driving force for the software requirement was to have a system that was easy to use. The ISU radiometer originally used LabWindows, to retrieve data and control the radiometer. A revision was made that changed the interface control to LabView. In both cases the interface was a graphical interface that allowed the user to easily set parameters and download the data.

The original ISU Radiometer did most of it's processing on board the FPGA. The graphical interface simply downloaded the data and allowed for control of certain parameters. With the switch to a software defined radio platform, we switched to having most of the processing done by the host computer. However, we still wanted to keep the interface hardware fairly simple. GNURadio was selected as it includes GNURadio Companion (GRC) which uses a graphical interface for creating the radio environment. It also includes options to create a user interface during the operation of the N200 as well. This allowed us to rapidly create both the critical radio components needed for the radiometer and also a control interface.

3.2 Hardware

There are two main hardware components that make up the radiometer. The first and probably the most critical is the RF front end of the radiometer. This critical component is what reads the RF coming in from the antenna or source. It is then amplified so that it can then be processed. Processing is then handled by the software defined radio, which digitizes the signal so it can be processed by the onboard FPGA and computer. As we will discuss next, due to noise temperature considerations, we need to make sure the incoming RF signal is as noise free as possible so we are able to process the signal that we want.

3.2.1 Noise Temperature Consideration

For any radiometer noise temperature is a large consideration and is critical to the design of the radiometer. One method to determine how well a radiometer is to look at the sensitivity of the radiometer. We can do this by looking at the smallest change in temperature the radiometer can see. We will call this the Noise Equivalent ΔT or $NE\Delta T$ of the radiometer. The equation for this is shown below.

$$NE\Delta T = \frac{T_A + T_{sys}}{\sqrt{\beta + \tau}} \tag{3.1}$$

For our radiometer, we will assume that our bandwidth, β, is fixed at 20 MHz. This is due to the current RF hardware that is placed in front of the N200 which has band-pass filters that restrict the incoming signal to between 1.400 and 1.420 GHz.

Need to add information about the ISU Radiometer antenna and what it adds to the noise temperature

For T_{sys}, both the RF front end and the software defined radio needs to be included and calculated to see the impact to the overall $NE\Delta T$ of the radiometer.

Finally τ is the integration time for the radiometer. This parameter is set by the user and is the only variable that we have direct control of.

3.2.2 Existing ISU Radiometer RF Front End

The existing ISU radiometer RF Front end uses a series of cascading low noise amplifiers (LNAs) that increase the power from the antenna while contributing a minimum amount of noise to the system. In addition, the current ISU Radiometer uses bandpass filters to narrow the bandwidth to the desired 1400 to 1425 MHz that we wish to monitor. While these are not needed with the addition of the software defined radio, since we can filter in software, they do not contribute much in terms of the noise temperature, and as passive components do not impact the performance of the radiometer.

3.2.3 DBSRX2 Noise Temperature Consideration

The N200 uses daughter board cards that allow for us to easily swap out different RF front ends. The DBSRX2 was selected due to the fact that it is receive only and that it will work in the frequency spectrum that we are interested in, primarily in the 1.4 GHz range. Since we are using the DBSRX2 after the LNAs that are already on the, the noise temperature added by the DBSRX2 will be quite small. The DBSRX2 adds approximately 5 dB to the noise factor of the system. Again though, since this is at the end of the RF chain, the total contribution of the DBSRX2 to the overall system noise temperature is small, and has been calculated to be 1.05 dB to the overall noise factor of the system.

While the DBSRX2 does have additional gain, this gain is not as critical since we have the additional gain from the ISU radiometer front end. While the gain the DBSRX2 is significant, the noise figure on the DBSRX2 by itself is quite high. Therefore, the DBSRX2 would not be a good candidate if the N200 would be used solely by itself. Ideally, there would be at least one if not multiple LNA's that have much better noise figure numbers than what the DBSRX2 and would be placed in front of the DBSRX2 daughter-board. However, the filters that are on the ISU radiometer are not as critical and, in fact, could be removed. Once we have the signal in the digital domain, it is possible to build and even adjust band-pass filers in software instead of using hardware. This can have the advantage of reducing some of the noise that these band-pass filters add, although their contribution is small. One drawback to this however is that it will require additional computation cycles. In our case, since the ISU radiometer already has band-pass filters, we assume the signal will be valid in that range. We do however, artificially apply a band-pass to the system in software by defining the sample rate that we receive the data from the N200. For this reason, we do need to know what our sample rate is, as often that will be smaller than the bandwidth coming from the ISU radiometer.

3.3 Software

A software defined radio by its definition requires software for the proper operation of the radio. In our case there is the software or firmware that resides on the FPGA and then the software that runs on the computer. For this thesis, we focused on the software running in GNURadio. This software package provides us with the ability to write the code needed to implement a total power radiometer but also gives us a method to control the SDR as well.

3.3.1 Obtaining the signal and GUI controls

The N200 sends all data across the 1 Gbps connection to be read in by GNURadio. This data is the raw I/Q values that is read by the on board A/D and processed by the on board FPGA. A very simple GNURadio implementation would simply take this data and store the data to the drive in a file. This can be very handy if we want to simply record the data and then process it later. However, depending on the sample rate, it can consume a large amount of storage. A short recording can easily consume 1-2 GB with a sample rate of 10 Msps. It also does not give us any immediate feedback on the

3.3.2 Total Power Radiometer Implementation

To implement a total power radiometer in software, we first need to look at how we implement a total power radiometer traditionally. Traditionally, a square law detector is used to detect the average power that is seen by the radiometer. This simple device uses a diode that gives a small voltage output based on the RF power present. This small voltage is then amplified and can now be calibrated with a known source to give us a noise temperature.

To implement this in software, we need to build a square law detector mathematically. We can then give this to the software defined radio to process the information accordingly. A square law detector mathematically, is the sum of the squares. Once the signal has been digitized, it is expressed in data bits of I and Q, which represent in-phase and quadrature-phase of the signal. By squaring each term, we get the desired result of the power of the signal.

Another step that we typically do in a traditional radiometer is to integrate the signal over time. This gives us an average of the signal and helps to smooth out the output. In addition, we will show later that the integration time can be adjusted to help improve our sensitivity of the radiometer.

In a traditional radiometer can integrate by using a simple integrator circuit, which consists of an op-amp, resistor and capacitor. This type of circuit is also equivalent to a low pass filter circuit as well, and the two are interchangeable. We can then look at how we filter in the digital domain, and this is down with a infinite impulse response filter or IIR. We can use this type of digital filter to then integrate the signal for our total power radiometer.

3.3.3 Display and Data Storage

Once we have the data that has been processed by the software defined radio, we will want to display this information and be able to store the data so we can analyze it later if needed. Data display can be handled GNURadio, where we can plot the total power over time. This allows the user to be able to visualize the total power and be able to determine if the total power has increased or decreased over the time window shown.

Although not usually needed for a total power radiometer, we also have the ability to look at the signal in terms of a frequency versus amplitude. This allows to look for any unusual signals that may be interfering with or causing erroneous data with our radiometer.

Finally we will want to store the data so we can do additional analyses on it at a later time. The GNURadio program allows us to store the data in two formats. The first format is storing the raw I/Q data from the radiometer. This format allows us to "playback" the data through GNURadio at a later time. This can be useful for if we wish to change parameters in GNURadio such as bandwidth or integration time. It can also be a good diagnostic tool as we can check that the signal coming in is clean or if we need to apply additional filters to remove an unwanted signal. It should be noted that this file can be quite large, consuming several gigabytes of data for a 20 MHz wide signal in a matter of minutes of record time.

The second format is the total power that has been calculated by the radiometer. This file is much smaller in size since much of the signal information has now been reduced to simple power versus time information. This allows for easy manipulation through any type of math program such as matlab for analysis.

CHAPTER 4. TESTING AND VERIFICATION

Testing and verification ensures that the new components that we have added to the system are working as intended and has not caused a negative impact on the overall system performance. To do this, several tests were conducted and verification was obtained by using a well known method of detecting power in a radiometer, a square-law detector.

Testing began with the square-law detector as this was one of the first components that was obtained. Next, testing was done with the the software defined radio as a whole. Further testing was done including a test run of the system and was used in the Dr. Brian Hornbuckle's E E 518 class. Finally a cold bath test was done with both the software defined radio and with the square law detector as a system to verify both functionality and to compare the results from both devices.

4.1 Square-law Detector

To verify the results that the software defined radio is obtaining, a square-law detector is used to measure the power of the incoming signal. The incoming signal is split using a power divider so that the signal will be the same to both devices with the exception of the 3 dB plus insertion loss the power divider adds. This allows us to verify the software defined radio with a proven system. Square-law detectors have traditionally been used in radiometers and have been proven to work in radiometer applications. They are also a very simple device which means there is little that can go wrong with using them.

4.1.1 Analog Devices ADL5902

The square-law detector that we obtained is the Analog Devices ADL5902. The ADL5902 is a true rms responding power detector that has a square law detector, a variable gain amplifier and an output driver. It also has a temperature sensor and will compensate for temperature variations. The output driver allows for the small signal from the square law detector to be amplified to a level that most analog to digital devices can detect. This driver however does have low noise and has a noise output of approximately $25nV/\sqrt{Hz}$ at 100 kHz. The ADL5902 operates from 50 MHz to 9 GHz and in most cases can detect down to -60 dBm. This works well in our application since the radiometer operates at 1.4 GHz and after the amplification stage we usually see between -40 to -30 dBm of power.

The specifications of the ADL5902 can be seen in Table 4.1 shown below.

Table 4.1 ADL5902 Specifications

Parameter @ 900 MHz	**Value**	**Units**
Frequency Range	50 to 9000	MHz
Dynamic Range	61	dB
Minimum Input Level, ±1.0 dB	60	dB
Maximum Input Level, ±1.0 dB	1	db
Logarithmic Slope	53.7	mV/dB
Output Voltage Range	0.03 to 4.8	V

The ADL5902 outputs an analog signal that falls between 0.03 volts and 4.8 volts. It outputs a change of 53.7 millivolt per dB detected by the ADL5902.

4.1.2 Data acquisition and storage

In order to analyze the data, a method was developed to acquire the data and store it for later use. For the N200 software defined radio, this is done automatically by the GNURadio program. Both the complete signal and the power information is stored to a file for later analysis. The square-law detector however outputs information as an analot voltage that is linearly proportional to the RF power measured. This required a system that can capture the analog signal from the ADL5902 and then send the data to be stored. This was done using a PIC32 processor, specifically the PIC32MX795F512L processor which has a 10-bit analog to digital converter built in. To aid in rapid development of the system, a BitWhacker board purchased through Sparkfun was used. Once the data was captured by the A/D converter, the PIC32 processor then sent this information through a RS232 port which can then be read by a computer. The computer would then store this information to a file. While it is certainly possible for the PIC32 to store this information to a SD card or other external memory device, the computer was used so that it would be possible to synchronize the data with the data being send from the SDR.

4.1.2.1 PIC32 Micro-controller system

The PIC32MX795F512L is the processor that was selected to acquire the analog voltage the ADL5902 outputs. This processor was chosen for the reason that it would allow for additional radiometer functions to be added to it later. The ISU radiometer currently uses a Rabbit based micro-controller. However, it was soon discovered that this processor was not able to handle all the tasks given to it efficiently. A new micro-controller system was known to be needed and thus this processor was chosen to handle not only the task for of obtaining information from the square-law detector but would have plenty of room for doing other tasks needed in the ISU radiometer.

The ISU radiometer has several analog temperature sensors and two thermal electric coolers (TEC) that also need to be monitored and controlled. In addition, control lines are needed to control the RF switches within the radiometer as well turning on and off the noise diode source. The TECs used in the ISU radiometer require a RS232 serial connection. In addition, it was also known that a GPS unit would also be added. This meant that at least 4 RS232 ports were needed including sending the data from the square-law detector. The micro-controller would also need to monitor at least 6 analog temperature sensors that are found at various points within the ISU radiometer. These requirements combined with the ability to add future sensors and/or tasks to the micro-controller led us to select this processor.

For the testing that was conducted for this thesis work, the PIC32MX795F512L is setup to capture the analog signal from the square-law detector using one of the 10-bit analog to digital ports it has. This information is then sent through one of the PIC32MX795F512L UART ports so that it can be read by the host computer.

4.1.3 Tests on the ADL5902

To test the ADL5902 a signal generator was used that had a controlled output. The specific signal generator that was used was an older model that allowed us to change the output in 10 dBm increments. The signal generator was also configured for 1.4 GHz as that is the frequency the ISU radiometer is configured to listen at. Ideally a noise generator would have been desirable, however a noise generator with adjustable power output could not be located on campus.

The ADL5902 is available from Analog Devices in an evaluation board. This board pairs the ADL5902 with a AD7466 12-bit analog to digital converter. This board can then be mated with Analog Devices BlackFin processor which acts as a USB gateway for the AD7466 data. A test program written in LabView is also provided as well.

The test program provided by Analog Devices allows us to query the ADL5902 and record the raw ADC value. The test program also allows us to enter in the frequency used during testing and the temperature during the test. The test program also allows us to calibrate the system as well. All of the data is then stored into an Excel spreadsheet which can be accessed later.

For this test we used the signal generator set to 1.4 GHz and started at −60 dBm for the output signal. This was selected as this is the lowest the square law detector can detect. The output power was then incremented on the signal generator in 10 dBm steps. There was no change to any other parameter. This was done up to 0 dBm. Any higher and there was risk of damage to the ADL5902. This test was then repeated several times and was done with the signal generator stepping up from −60 dBm to 0 dBm and from 0 dBm to −60 dBm.

The data collected was then graphed using Excel. The graph shown in 4.1 Shows that the ADL5902 has a linear output and matches the expected value based on the input.

The ADL5902 test board was considered to be used in the final design for the ISU radiometer and in additional tests. However, Analog Devices decided to make the information to communicate with the BlackFin processor proprietary and this hampered attempts to integrate that system into our overall system design. For this reason, it was decided to simply record

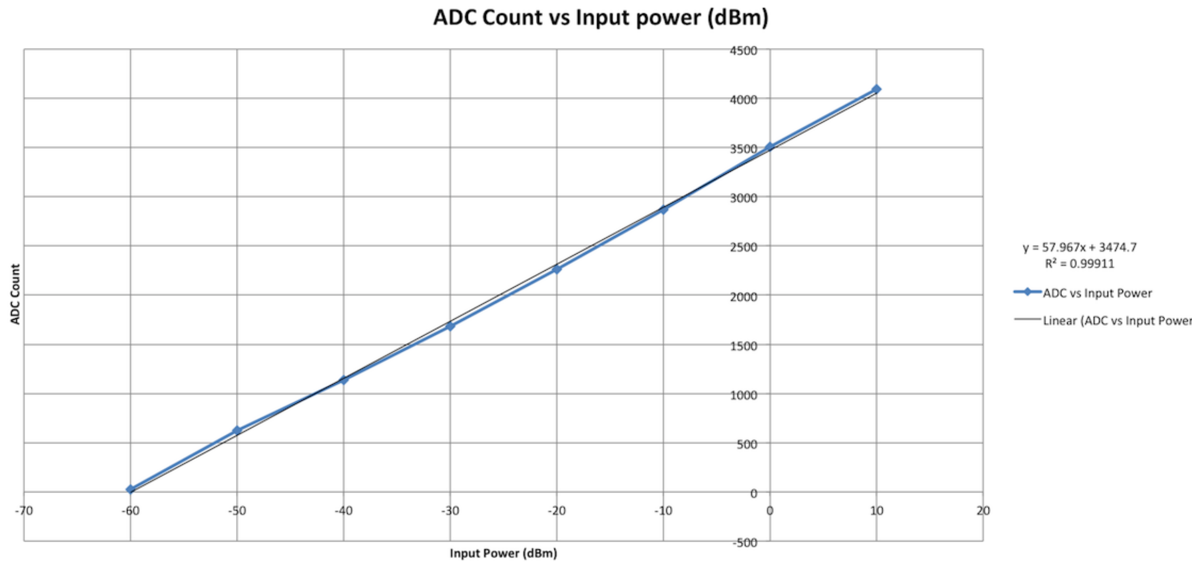

Figure 4.1 Graph showing the linearity of the ADL5902

the analog data that the ADL5902 itself puts out and the PIC32 described above was used to capture and send this information.

4.2 Software Defined Radio tests

Once it was confirmed that the square-law detector was working within the specifications that were given, testing then moved to the software defined radio. Once the Python program was established for replicating a total power radiometer in software, this was then loaded into GNURadio and used to control the N200 SDR. GNURadio has a built in noise generator that was then used to test the program and it's ability to measure small changes in noise. This simulated noise verified that the program written was able to detect changes in noise power using a simulated Gaussian source. It was desired to also use a hardware based noise generator, however a suitable noise generator could not be located on campus.

4.3 E E 518 Lab Tests

Further testing of the square-law detector was performed by using the radiometer in a real world event. This test was exercised in conjunction with the Microwave Remote Sensing class (E E 518) under Dr. Brian Hornbuckle. For this test the radiometer was installed on the roof of Agronomy Hall and was configured so that it could be rotated so it will have a clear view of the sky. This allowed us to make measurements with a cold source and then at the ground to record a warm source.

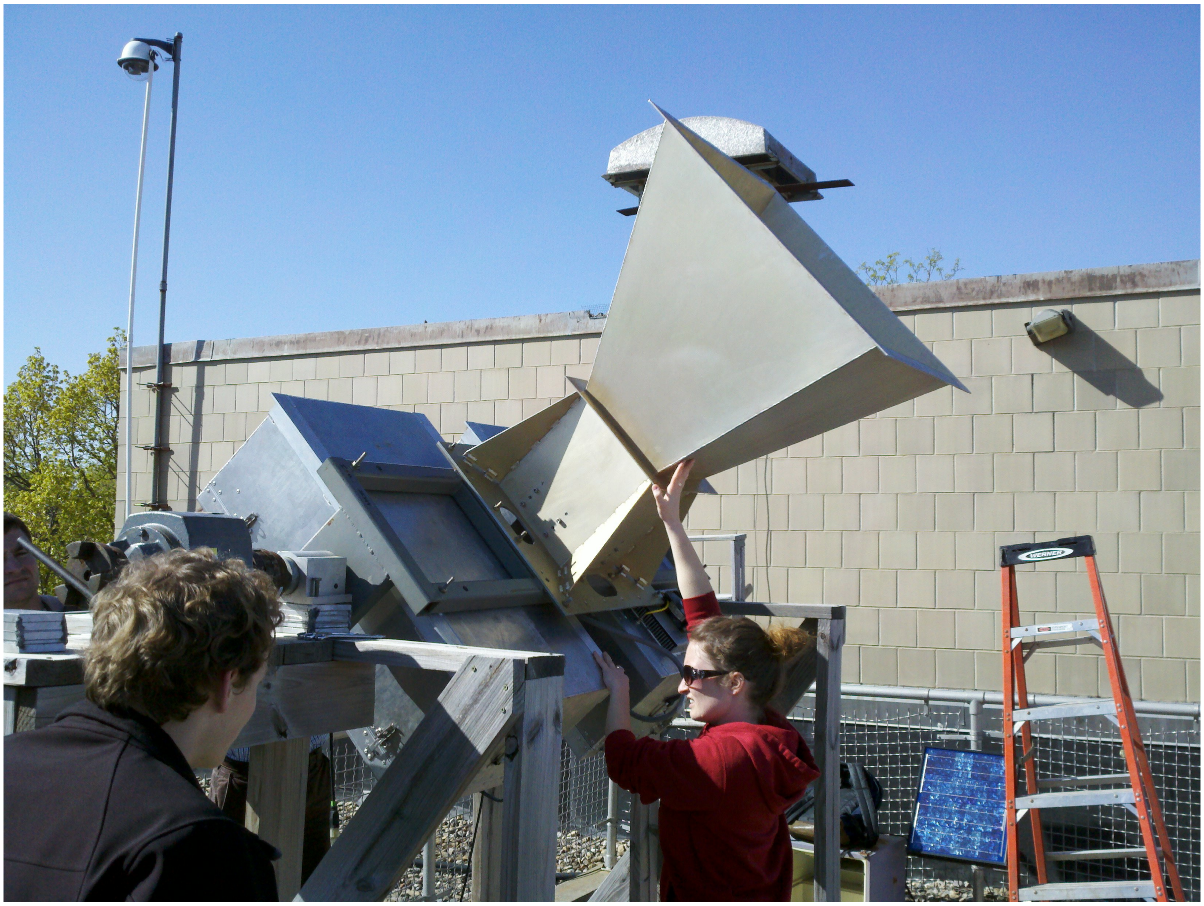

Figure 4.2 Students rotating the radiometer for an experiment on Agronomy Hall

4.4 Total Power Radiometer Test with Cold Water Bath

To fully test both the total power radiometer in the software defined radio and the square-law detector, a cold water bath experiment was established to verify that both the square-law detector and SDR was able to measure real world changes in the noise. In addition, this would also give us a calibration test (confirm this) line to work with.

4.4.1 Laboratory Setup

For this experiment the, ISU radiometer was setup in a laboratory that had access to measurement tools. To measure the change in noise, a 50 ohm load was attached to the radiometer. The ISU radiometer, with the current filters and LNAs would then amplify and filter the signal. This signal was then measured by the square-law detector and the N200 SDR. The 50 ohm load was then submersed into a hot or cold bath. These water baths were temperature controlled for 100 degrees Centigrade and to 10 degrees Centigrade. The load was submersed in each bath for 2 minutes to allow for it to reach the same temperature as the bath. The noise measured was then recorded using GNURadio with the N200 SDR or with the PIC32 setup on the square-law detector. In addition to the total power measurements, the raw I-Q data was also recorded. This allowed us to replay the experiment through GNURadio for further study.

4.4.2 Test Results

Several experiments were conducted with this cold-bath configuration to establish that the results were reproducible. In each case, the experiment showed that both the square-law detector and the SDR were able to measure a change of noise temperature. Once experiment used a change of only 50 C and the change was easily recorded on the SDR. Using the experiment described in which the temperatures were held constant, this also gave us a calibration point that we can use to calibrate the radiometer.

4.5 Liquid Nitrogen Test

The Liquid Nitrogen Test was conducted to verify that the radiometer was able to operate within the specifications that were given for the radiometer. This test is also a fairly common test for testing and calibrating a traditional type of radiometer. With this test, we can test the radiometer by measuring extreme values for the noise temperature. To do this, we submerge a matched load attached to the radiometer into a liquid nitrogen bath. This cools the load, which represents our source, to a physical temperature of approximately 77 Kelvin. We can then select several ”warmer” temperatures such as a ice water bath, room temperature or even boiling water. Since we expect that the radiometer is linear in how it responds to these noise temperatures, we can then build a calibration line for the radiometer.

4.5.1 Testing Apparatus

To run this test, we need to have some equipment for this. First and foremost, we need a radiometer. For this test we used the ISU Radiometer front end, with the LNAs and bandpass filters in place. The noise diode was turned off for these experiments. A fifty ohm matched load was then attached to low loss coax and represented our source to the radiometer. Finally, the output of the radiometer was run to the Ettus N200 software defined radio instead of running the on-board Analog to Digital converter and FPGA. The data from the N200 was then sent to a MacBook Pro running GNURadio and the custom software that I had written. This data is sent to the MacBook Pro through a 1 gigabit Ethernet connection due to the large bandwidth coming from the N200. For these experiments, we only used one ”side” of the radiometer. In theory, both sides of the ISU radiometer should be identical as far as results go.

4.5.2 Test Run 1

4.6 Further testing

CHAPTER 5. PERFORMANCE AND EVALUATION

Radiometers measure power and it is this information that we can use to determine information about a certain target. This power however is often expressed in terms of an equivalent temperature and if we are looking at an object, this is the brightness temperature of that object. The primary function of the radiometer is to measure the power that is seen at the radiometer's antenna. Ideally this is the brightness temperature of an object of interest. In our case, this is usually looking at the ground or soil sample. Thus the goal of the radiometer is to measure this antenna temperature with sufficient resolution and accuracy that a correlation can be made between the antenna temperature and the object's temperature that we are studying.

5.1 Performance of a radiometer

The performance of a radiometer can be measured by looking at both the sensitivity and the accuracy of the radiometer. In addition, we need to be concerned with the stability of the radiometer as this affects the accuracy of the radiometer.

5.1.1 Sensitivity

Sensitivity of the radiometer relates to the amount of power that the radio selects from the antenna. This selection is then dependent on the bandwidth that the radiometer is able to listen to. The radiometer however detects not only the signal of interest but also receives a noise signal as well. This noise is added to the signal and can not be separated from the signal. Because this noise is added to the signal, we must be able to determine a change in the signal while the noise signal is also present.

Power from the radiometer can be expressed in equation 5.1 from chapter 2, where we take into consideration bandwidth, gain and the input from the antenna plus the noise added from the antenna.

$$P = k * \beta * G * (T_A + T_N) \tag{5.1}$$

Sensitivity relates to being able to distinguish one signal from the other. In other words, we must be able to distinguish, or detect, our wanted signal from noise that is present in the signal. To demonstrate this, consider a system that has a system noise temperature of 700K and an antenna temperature of 300K. This gives us a total noise temperature of 1000K. Therefore, if we wish to detect a change of 1K, we need to be able to detect between 1000K and 1001K.

Since our signal is really random fluctuations about a mean power input to our system, we can reduce the fluctuations by averaging or integrating the signal. This results in equation 5.2 which is derived in chapter 2.

$$\Delta T = \frac{T_A + T_N}{\sqrt{\beta * \tau}} \tag{5.2}$$

This equation gives us the radiometer sensitivity based on the input, which is both the noise and input signal with consideration to the bandwidth and integration time, or averaging, that is done to the signal.

5.1.2 Accuracy and Stability

Stability and accuracy are additional problems that need to be considered when looking at the radiometer system. To begin we can once again look at the power received equation

$$P = k * B * G * (T_A + T_N) \tag{5.3}$$

As we look at this equation, we can see that if k, B, G, and T_N are constant, then stability can be assured. k is a known constant and we can also assume that our bandwidth, B, will also remain constant, or at least while we are taking our measurements. Gain and the noise temperature however can vary.

Gain is usually our largest factor that can change on a radiometer and even with a software defined radio this is still a large source of variation. This is due to the analog nature of the amplifiers that affect a large portion of the gain in our system. Various things can affect our gain, but the two largest factors is the physical temperature of the amplifier and the voltage that feeds the amplifier. Voltage can be controlled to a degree. High accuracy voltage regulators can help control fluctuations in voltages that can in turn affect the gain. A factor however that is harder to control is temperature. It is because of this that the current ISU radiometer has gone to great strides to control the temperature of the amplifiers to maintain a contstant temperature.

5.2 Required Performance Requirements

To help us quantify the required performance of the radiometer, we looked to what the existing ISU radiometer had in terms of performance. These performance numbers were in part determined by Dr. Brian Hornbuckle but also derived from existing radiometers. As stated earlier, although a radiometer measures power, we often convert this to an equivalent brightness temperature. Specifically, we are looking at the brightness temperature of the antenna added to the brightness temperature of the object of interest. We also have to be able to detect a minimum amount of power. Since changes in noise can be small, the better the sensitivity of the radiometer, the better we can detect these small changes.

5.3 Square-law Detector Performance

The Square-law detector was added to our system in order to give us another reference point and to help verify the power output that the software defined radio. The performance of our square-law detector is based on two items. The first is the actual square-law detector itself. The sensitivity of this device accounts for most of the performance factor of the system. In our system the output of this square-law detector is then feed directly into an analog to digital converter. Therefore the performance of this A/D converter needs to be accounted for as well.

5.4 Software Defined Radio Performance

A summary of the performance of the SDR and this can conclude the results.

CHAPTER 6. Conclusion and Future work

6.1 Future work

The main purpose of this research is to demonstrate and prove that an off the shelf software defined radio is capable of operating like a radiometer and can do so within the same or better specifications that are seen in most radiometers today. To do that, we used a vary basic radiometer setup and configured our SDR to behave as a single input radiometer. It was also assumed that the input from the RF front end of the radiometer was stable, which in our case was accomplished by stabilizing the temperature of the active components in the RF Front end.

However, this temperature stabilization requires extra weight and bulk to be added to the radiometer. In the application that the ISU radiometer was designed for, this was not a major drawback to the system. However other applications may require a system that does not have this type of temperature stabilization. For those type of systems, other radiometers use different methods to account for and adjust for fluctuations in the RF front end.

6.1.1 Improvements to the ISU radiometer

Most of the work done with this thesis used much of the existing RF hardware that was already on the ISU radiometer. Improvements to LNAs, especially those in the GHz and above range, have been able to reduce noise figures while still keeping relatively high gains. It would be advantageous to reconsider the RF chain while keeping the SDR in mind for the design. Since the SDR that we used has some gain, which can be adjusted, it may be possible to reduce some of the gain that the ISU radiometer provides. This may help in the overall system performance. In addition, by removing some things such as the filters that are used with the

ISU radiometer, we can now make the system more frequency agile. Although use of the 1.4 GHz spectrum is ideal for most remote sensing tasks, removing these restrictions would further enhance the functionality of the radiometer by being able to look at other frequencies. This can also aid in the radiometer needing to shift around to avoid possible interference as well.

There also needs to be additional improvements to the support systems to the ISU radiometer if it will continue to be the main radiometer for remote sensing. Currently, the system to control the thermal properties is out-dated and does not work reliably in terms of being able to set the thermal temperature. There is also now additional "dead weight" as there is equipment currently in the ISU radiometer that can be removed once the SDR system has been fully integrated into the ISU radiometer.

6.1.2 Improving on the FPGA firmware

For this thesis we focused on the software that would run on a PC or comparable computer system running a full OS like Linux. While this aids to speed up development and works just fine for testing the theory on a off the shelf SDR acting as a radiometer, it does require additional hardware. For some applications of a radiometer, this is not a huge concern. In the case for the ISU radiometer, the concern is not that large since the radiometer is not designed to be "portable" and requires additional support equipment such as a generator anyway. However, other remote sensing applications, such as space based applications, would require a more efficient method. It is very possible to move the software generated in GNURadio into the firmware of the N200. This will help to offload the work needed by the computer and would allow for the computer or similiar system to act as more of a control method and for data storage.

6.1.3 Correlation

One such method is to correlate the information with another input from the antenna. This can be accomplished by using a dual polarization antenna which the ISU radiometer currently uses. The previous ISU radiometer setup did just that, it correlated the data from the vertical polarization and from the horizontal polarization on the antenna. This is very plausible with the SDR we have chosen for this work, the N200. The N200 can have up to two daughter-boards installed and can stream both ports to the computer to be analyzed.

6.1.4 Noise injection

Another method to stabilize the signal is to inject the signal with a known noise diode as a reference point. The current ISU radiometer does this by turning on and off a noise diode source and injecting this noise in the RF front end.

Another method that can be explored is using a simulated noise source which is then done by the software instead of using a hardware based source. This has two advantages; one it eliminates the need for additional hardware and two we now have control over the noise source and are able to change the amplitude if needed.

6.2 Conclusion

In this thesis we have shown that an off the shelf SDR can be used to perform as a radiometer. Using a SDR has several advantages such as a more flexible system and can result in a less expensive system. Since a SDR offers high flexibility, changes to the system can be done very quickly and helps in future proofing the system.

APPENDIX A. Source code

The following is the Python code used with the Ettus N200 software defined radio. This python code should work on any platform that has the GNURadio libraries installed and also the USRP drivers for communicating with the N200. This code can also be easily modified to communicate with any SDR that GNURadio can communicate with.

This script was developed using GNURadio Companion. This copy of the script may be out of date. The latest version of the script file can be found on the author's personal website, http://www.rfgeeks.com/research/master-s-thesis.

Python code for total power radiometer

Total Power Radiometer

```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Total Power Radiometer - N200
# Author: Matthew E Nelson
# Description: Total power radiometer connecting to a N200 SDR
# Generated: Sat Mar 16 20:14:21 2013
##################################################

from datetime import datetime
```

```
from gnuradio import blks2
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import uhd
from gnuradio import window
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from gnuradio.wxgui import fftsink2
from gnuradio.wxgui import forms
from gnuradio.wxgui import numbersink2
from gnuradio.wxgui import scopesink2
from grc_gnuradio import blks2 as grc_blks2
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class N200_TPR(grc_wxgui.top_block_gui):

    def __init__(self, dcg=1, devid="addr=192.168.10.2", srate=10.0e6, spa=1
        grc_wxgui.top_block_gui.__init__(self, title="Total Power Radiometer
        _icon_path = "/usr/share/icons/hicolor/32x32/apps/gnuradio-grc.png"
        self.SetIcon(wx.Icon(_icon_path, wx.BITMAP_TYPE_ANY))

        ##################################################
        # Parameters
        ##################################################
        self.dcg = dcg
        self.devid = devid
```

```
self.srate = srate
self.spa = spa
self.rxant = rxant
self.tpint = tpint
self.rfgain = rfgain
self.frequency = frequency
self.decln = decln
self.subdev = subdev
self.maxg = maxg
self.fftsize = fftsize
self.clock = clock

##################################################
# Variables
##################################################
self.israte = israte = srate
self.samp_rate = samp_rate = int(israte)
self.prefix = prefix = "tpr_"
self.variable_static_text_0_0_0_0 = variable_static_text_0_0_0_0 = c
self.variable_static_text_0_0_0 = variable_static_text_0_0_0 = devid
self.variable_static_text_0_0 = variable_static_text_0_0 = subdev
self.variable_static_text_0 = variable_static_text_0 = israte
self.spec_data_fifo = spec_data_fifo = "spectrum_" + datetime.now().
self.spavg = spavg = int(spa)
self.scope_rate = scope_rate = 2.0
self.recfile_tpr = recfile_tpr = prefix + datetime.now().strftime("%
self.recfile_kelvin = recfile_kelvin = prefix+"kelvin" + datetime.no
self.rec_button_tpr = rec_button_tpr = 1
```

```
self.rec_button_iq = rec_button_iq = 1
self.noise_amplitude = noise_amplitude = .5
self.integ = integ = tpint
self.idecln = idecln = decln
self.gain = gain = 23
self.freq = freq = frequency
self.file_rate = file_rate = 2.0
self.fftrate = fftrate = int(samp_rate/fftsize)
self.det_rate = det_rate = int(20.0)
self.dc_gain = dc_gain = int(dcg)
self.calib_2 = calib_2 = -342.774
self.calib_1 = calib_1 = 4.0755e3
self.add_noise = add_noise = 0

##################################################
# Blocks
##################################################
self.Main = self.Main = wx.Notebook(self.GetWin(), style=wx.NB_TOP)
self.Main.AddPage(grc_wxgui.Panel(self.Main), "Continuum + Controls"
self.Main.AddPage(grc_wxgui.Panel(self.Main), "Spectral")
self.Main.AddPage(grc_wxgui.Panel(self.Main), "Meter")
self.Add(self.Main)
_spavg_sizer = wx.BoxSizer(wx.VERTICAL)
self._spavg_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_spavg_sizer,
    value=self.spavg,
    callback=self.set_spavg,
```

```
    label="Spectral Averaging (Seconds)",
    converter=forms.int_converter(),
    proportion=0,
)
self._spavg_slider = forms.slider(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_spavg_sizer,
    value=self.spavg,
    callback=self.set_spavg,
    minimum=1,
    maximum=20,
    num_steps=20,
    style=wx.SL_HORIZONTAL,
    cast=int,
    proportion=1,
)
self.Main.GetPage(0).GridAdd(_spavg_sizer, 1, 1, 1, 1)
self._rec_button_tpr_chooser = forms.button(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.rec_button_tpr,
    callback=self.set_rec_button_tpr,
    label="Record TPR Data",
    choices=[0,1],
    labels=['Stop','Start'],
)
self.Main.GetPage(0).GridAdd(self._rec_button_tpr_chooser, 4, 1, 1,
self._rec_button_iq_chooser = forms.button(
    parent=self.Main.GetPage(0).GetWin(),
```

```
    value=self.rec_button_iq,
    callback=self.set_rec_button_iq,
    label="Record I/Q Data",
    choices=[0,1],
    labels=['Stop','Start'],
)
self.Main.GetPage(0).GridAdd(self._rec_button_iq_chooser, 4, 0, 1, 1
self._israte_chooser = forms.radio_buttons(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.israte,
    callback=self.set_israte,
    label="Sample Rate (BW)",
    choices=[1e6,2e6,5e6,10e6,25e6],
    labels=['1 MHz','2 MHz','5 MHz','10 MHz','25 MHz'],
    style=wx.RA_HORIZONTAL,
)
self.Main.GetPage(0).GridAdd(self._israte_chooser, 1, 3, 1, 1)
_integ_sizer = wx.BoxSizer(wx.VERTICAL)
self._integ_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_integ_sizer,
    value=self.integ,
    callback=self.set_integ,
    label="Integration Time (Seconds)",
    converter=forms.float_converter(),
    proportion=0,
)
self._integ_slider = forms.slider(
```

```
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_integ_sizer,
    value=self.integ,
    callback=self.set_integ,
    minimum=1,
    maximum=60,
    num_steps=100,
    style=wx.SL_HORIZONTAL,
    cast=float,
    proportion=1,
)
self.Main.GetPage(0).GridAdd(_integ_sizer, 0, 2, 1, 1)
_gain_sizer = wx.BoxSizer(wx.VERTICAL)
self._gain_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_gain_sizer,
    value=self.gain,
    callback=self.set_gain,
    label="RF Gain (dB)",
    converter=forms.float_converter(),
    proportion=0,
)
self._gain_slider = forms.slider(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_gain_sizer,
    value=self.gain,
    callback=self.set_gain,
    minimum=0,
```

```
        maximum=maxg,
        num_steps=100,
        style=wx.SL_HORIZONTAL,
        cast=float,
        proportion=1,
    )
    self.Main.GetPage(0).GridAdd(_gain_sizer, 0, 1, 1, 1)
    self._freq_text_box = forms.text_box(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.freq,
        callback=self.set_freq,
        label="Center Frequency (Hz)",
        converter=forms.float_converter(),
    )
    self.Main.GetPage(0).GridAdd(self._freq_text_box, 0, 0, 1, 1)
    self._dc_gain_chooser = forms.radio_buttons(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.dc_gain,
        callback=self.set_dc_gain,
        label="DC Gain",
        choices=[1, 10, 100, 1000, 10000],
        labels=[],
        style=wx.RA_HORIZONTAL,
    )
    self.Main.GetPage(0).GridAdd(self._dc_gain_chooser, 1, 0, 1, 1)
    self._calib_2_text_box = forms.text_box(
        parent=self.Main.GetPage(0).GetWin(),
        value=self.calib_2,
```

```
    callback=self.set_calib_2,
    label="Calibration value 2",
    converter=forms.float_converter(),
)
self.Main.GetPage(0).GridAdd(self._calib_2_text_box, 3, 1, 1, 1)
self._calib_1_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.calib_1,
    callback=self.set_calib_1,
    label="Calibration value 1",
    converter=forms.float_converter(),
)
self.Main.GetPage(0).GridAdd(self._calib_1_text_box, 3, 0, 1, 1)
self.wxgui_scopesink2_0 = scopesink2.scope_sink_f(
    self.Main.GetPage(0).GetWin(),
    title="Total Power",
    sample_rate=scope_rate,
    v_scale=0,
    v_offset=0,
    t_scale=450,
    ac_couple=False,
    xy_mode=False,
    num_inputs=1,
    trig_mode=gr.gr_TRIG_MODE_STRIPCHART,
    y_axis_label="Power Level",
    size=(800,300),
)
self.Main.GetPage(0).Add(self.wxgui_scopesink2_0.win)
```

```
self.wxgui_numbersink2_0 = numbersink2.number_sink_f(
    self.Main.GetPage(2).GetWin(),
    unit="Kelvin",
    minval=0,
    maxval=400,
    factor=1,
    decimal_places=6,
    ref_level=0,
    sample_rate=scope_rate,
    number_rate=15,
    average=False,
    avg_alpha=None,
    label="Number Plot",
    peak_hold=False,
    show_gauge=True,
)
self.Main.GetPage(2).Add(self.wxgui_numbersink2_0.win)
self.wxgui_fftsink2_0 = fftsink2.fft_sink_c(
    self.Main.GetPage(1).GetWin(),
    baseband_freq=freq,
    y_per_div=10,
    y_divs=10,
    ref_level=50,
    ref_scale=2.0,
    sample_rate=israte,
    fft_size=1024,
    fft_rate=5,
    average=True,
```

```
    avg_alpha=0.1,
    title="Spectrum",
    peak_hold=False,
    size=(800,400),
)
self.Main.GetPage(1).Add(self.wxgui_fftsink2_0.win)
self._variable_static_text_0_0_0_0_static_text = forms.static_text(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.variable_static_text_0_0_0_0,
    callback=self.set_variable_static_text_0_0_0_0,
    label="USRP Clock",
    converter=forms.float_converter(),
)
self.Main.GetPage(0).GridAdd(self._variable_static_text_0_0_0_0_stat
self._variable_static_text_0_0_0_static_text = forms.static_text(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.variable_static_text_0_0_0,
    callback=self.set_variable_static_text_0_0_0,
    label="Device",
    converter=forms.str_converter(),
)
self.Main.GetPage(0).GridAdd(self._variable_static_text_0_0_0_static
self._variable_static_text_0_0_static_text = forms.static_text(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.variable_static_text_0_0,
    callback=self.set_variable_static_text_0_0,
    label="SubDev",
    converter=forms.str_converter(),
```

```
)
self.Main.GetPage(0).GridAdd(self._variable_static_text_0_0_static_t
self._variable_static_text_0_static_text = forms.static_text(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.variable_static_text_0,
    callback=self.set_variable_static_text_0,
    label="Samp rate",
    converter=forms.float_converter(),
)
self.Main.GetPage(0).GridAdd(self._variable_static_text_0_static_tex
self.uhd_usrp_source_0 = uhd.usrp_source(
    device_addr=devid,
    stream_args=uhd.stream_args(
        cpu_format="fc32",
        channels=range(1),
    ),
)
self.uhd_usrp_source_0.set_samp_rate(samp_rate)
self.uhd_usrp_source_0.set_center_freq(freq, 0)
self.uhd_usrp_source_0.set_gain(gain, 0)
_noise_amplitude_sizer = wx.BoxSizer(wx.VERTICAL)
self._noise_amplitude_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_noise_amplitude_sizer,
    value=self.noise_amplitude,
    callback=self.set_noise_amplitude,
    label='noise_amplitude',
    converter=forms.float_converter(),
```

```
    proportion=0,
)
self._noise_amplitude_slider = forms.slider(
    parent=self.Main.GetPage(0).GetWin(),
    sizer=_noise_amplitude_sizer,
    value=self.noise_amplitude,
    callback=self.set_noise_amplitude,
    minimum=.01,
    maximum=1,
    num_steps=100,
    style=wx.SL_HORIZONTAL,
    cast=float,
    proportion=1,
)
self.Main.GetPage(0).GridAdd(_noise_amplitude_sizer, 3, 2, 1, 1)
self._idecln_text_box = forms.text_box(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.idecln,
    callback=self.set_idecln,
    label="Declination",
    converter=forms.float_converter(),
)
self.Main.GetPage(0).GridAdd(self._idecln_text_box, 1, 2, 1, 1)
self.gr_single_pole_iir_filter_xx_0 = gr.single_pole_iir_filter_ff(1
self.gr_multiply_const_vxx_1 = gr.multiply_const_vff((calib_1, ))
self.gr_multiply_const_vxx_0 = gr.multiply_const_vff((dc_gain, ))
self.gr_keep_one_in_n_3 = gr.keep_one_in_n(gr.sizeof_float*1, int(de
self.gr_keep_one_in_n_2 = gr.keep_one_in_n(gr.sizeof_float*1, int(de
```

```
self.gr_keep_one_in_n_1 = gr.keep_one_in_n(gr.sizeof_float*fftsize,
self.gr_keep_one_in_n_0 = gr.keep_one_in_n(gr.sizeof_float*1, samp_r
self.gr_file_sink_3 = gr.file_sink(gr.sizeof_gr_complex*1, prefix+"i
self.gr_file_sink_3.set_unbuffered(False)
self.gr_file_sink_2 = gr.file_sink(gr.sizeof_float*1, recfile_kelvin
self.gr_file_sink_2.set_unbuffered(True)
self.gr_file_sink_1 = gr.file_sink(gr.sizeof_float*fftsize, spec_dat
self.gr_file_sink_1.set_unbuffered(True)
self.gr_file_sink_0 = gr.file_sink(gr.sizeof_float*1, recfile_tpr)
self.gr_file_sink_0.set_unbuffered(True)
self.gr_complex_to_mag_squared_0 = gr.complex_to_mag_squared(1)
self.gr_add_const_vxx_0 = gr.add_const_vff((calib_2, ))
self.blks2_valve_2 = grc_blks2.valve(item_size=gr.sizeof_gr_complex*
self.blks2_valve_1 = grc_blks2.valve(item_size=gr.sizeof_float*1, op
self.blks2_valve_0 = grc_blks2.valve(item_size=gr.sizeof_float*1, op
self.blks2_logpwrfft_x_0 = blks2.logpwrfft_c(
    sample_rate=samp_rate,
    fft_size=fftsize,
    ref_scale=2,
    frame_rate=fftrate,
    avg_alpha=1.0/float(spavg*fftrate),
    average=True,
)
self._add_noise_chooser = forms.button(
    parent=self.Main.GetPage(0).GetWin(),
    value=self.add_noise,
    callback=self.set_add_noise,
    label="Noise Source",
```

```
    choices=[0,1],
    labels=['Off','On'],
)
self.Main.GetPage(0).GridAdd(self._add_noise_chooser, 3, 3, 1, 1)

##################################################
# Connections
##################################################
self.connect((self.gr_complex_to_mag_squared_0, 0), (self.gr_single_
self.connect((self.gr_single_pole_iir_filter_xx_0, 0), (self.gr_keep
self.connect((self.uhd_usrp_source_0, 0), (self.blks2_logpwrfft_x_0,
self.connect((self.gr_keep_one_in_n_0, 0), (self.gr_multiply_const_v
self.connect((self.blks2_logpwrfft_x_0, 0), (self.gr_keep_one_in_n_1
self.connect((self.gr_keep_one_in_n_1, 0), (self.gr_file_sink_1, 0))
self.connect((self.gr_keep_one_in_n_2, 0), (self.wxgui_scopesink2_0,
self.connect((self.gr_multiply_const_vxx_0, 0), (self.gr_keep_one_in
self.connect((self.gr_multiply_const_vxx_0, 0), (self.gr_keep_one_in
self.connect((self.gr_keep_one_in_n_3, 0), (self.gr_multiply_const_v
self.connect((self.gr_add_const_vxx_0, 0), (self.wxgui_numbersink2_0
self.connect((self.gr_multiply_const_vxx_1, 0), (self.gr_add_const_v
self.connect((self.gr_keep_one_in_n_3, 0), (self.blks2_valve_0, 0))
self.connect((self.blks2_valve_0, 0), (self.gr_file_sink_0, 0))
self.connect((self.gr_add_const_vxx_0, 0), (self.blks2_valve_1, 0))
self.connect((self.blks2_valve_1, 0), (self.gr_file_sink_2, 0))
self.connect((self.uhd_usrp_source_0, 0), (self.blks2_valve_2, 0))
self.connect((self.blks2_valve_2, 0), (self.gr_file_sink_3, 0))
self.connect((self.uhd_usrp_source_0, 0), (self.gr_complex_to_mag_sq
self.connect((self.uhd_usrp_source_0, 0), (self.wxgui_fftsink2_0, 0)
```

```
def get_dcg(self):
    return self.dcg

def set_dcg(self, dcg):
    self.dcg = dcg
    self.set_dc_gain(int(self.dcg))

def get_devid(self):
    return self.devid

def set_devid(self, devid):
    self.devid = devid
    self.set_variable_static_text_0_0_0(self.devid)

def get_srate(self):
    return self.srate

def set_srate(self, srate):
    self.srate = srate
    self.set_israte(self.srate)

def get_spa(self):
    return self.spa

def set_spa(self, spa):
    self.spa = spa
```

```
        self.set_spavg(int(self.spa))

    def get_rxant(self):
        return self.rxant

    def set_rxant(self, rxant):
        self.rxant = rxant

    def get_tpint(self):
        return self.tpint

    def set_tpint(self, tpint):
        self.tpint = tpint
        self.set_integ(self.tpint)

    def get_rfgain(self):
        return self.rfgain

    def set_rfgain(self, rfgain):
        self.rfgain = rfgain

    def get_frequency(self):
        return self.frequency

    def set_frequency(self, frequency):
        self.frequency = frequency
        self.set_freq(self.frequency)
```

```
def get_decln(self):
    return self.decln

def set_decln(self, decln):
    self.decln = decln
    self.set_idecln(self.decln)

def get_subdev(self):
    return self.subdev

def set_subdev(self, subdev):
    self.subdev = subdev
    self.set_variable_static_text_0_0(self.subdev)

def get_maxg(self):
    return self.maxg

def set_maxg(self, maxg):
    self.maxg = maxg

def get_fftsize(self):
    return self.fftsize

def set_fftsize(self, fftsize):
    self.fftsize = fftsize
    self.set_fftrate(int(self.samp_rate/self.fftsize))

def get_clock(self):
```

```
        return self.clock

    def set_clock(self, clock):
        self.clock = clock
        self.set_variable_static_text_0_0_0_0(self.clock)

    def get_israte(self):
        return self.israte

    def set_israte(self, israte):
        self.israte = israte
        self.set_samp_rate(int(self.israte))
        self._israte_chooser.set_value(self.israte)
        self.set_variable_static_text_0(self.israte)
        self.wxgui_fftsink2_0.set_sample_rate(self.israte)

    def get_samp_rate(self):
        return self.samp_rate

    def set_samp_rate(self, samp_rate):
        self.samp_rate = samp_rate
        self.set_fftrate(int(self.samp_rate/self.fftsize))
        self.gr_single_pole_iir_filter_xx_0.set_taps(1.0/((self.samp_rate*se
        self.gr_keep_one_in_n_0.set_n(self.samp_rate/self.det_rate)
        self.blks2_logpwrfft_x_0.set_sample_rate(self.samp_rate)
        self.uhd_usrp_source_0.set_samp_rate(self.samp_rate)

    def get_prefix(self):
```

```
        return self.prefix

    def set_prefix(self, prefix):
        self.prefix = prefix
        self.gr_file_sink_3.open(self.prefix+"iq_raw" + datetime.now().strft
        self.set_recfile_tpr(self.prefix + datetime.now().strftime("%Y.%m.%d
        self.set_recfile_kelvin(self.prefix+"kelvin" + datetime.now().strfti

    def get_variable_static_text_0_0_0_0(self):
        return self.variable_static_text_0_0_0_0

    def set_variable_static_text_0_0_0_0(self, variable_static_text_0_0_0_0)
        self.variable_static_text_0_0_0_0 = variable_static_text_0_0_0_0
        self._variable_static_text_0_0_0_0_static_text.set_value(self.variab

    def get_variable_static_text_0_0_0(self):
        return self.variable_static_text_0_0_0

    def set_variable_static_text_0_0_0(self, variable_static_text_0_0_0):
        self.variable_static_text_0_0_0 = variable_static_text_0_0_0
        self._variable_static_text_0_0_0_static_text.set_value(self.variable

    def get_variable_static_text_0_0(self):
        return self.variable_static_text_0_0

    def set_variable_static_text_0_0(self, variable_static_text_0_0):
        self.variable_static_text_0_0 = variable_static_text_0_0
        self._variable_static_text_0_0_static_text.set_value(self.variable_s
```

```
def get_variable_static_text_0(self):
    return self.variable_static_text_0

def set_variable_static_text_0(self, variable_static_text_0):
    self.variable_static_text_0 = variable_static_text_0
    self._variable_static_text_0_static_text.set_value(self.variable_sta

def get_spec_data_fifo(self):
    return self.spec_data_fifo

def set_spec_data_fifo(self, spec_data_fifo):
    self.spec_data_fifo = spec_data_fifo
    self.gr_file_sink_1.open(self.spec_data_fifo)

def get_spavg(self):
    return self.spavg

def set_spavg(self, spavg):
    self.spavg = spavg
    self._spavg_slider.set_value(self.spavg)
    self._spavg_text_box.set_value(self.spavg)
    self.blks2_logpwrfft_x_0.set_avg_alpha(1.0/float(self.spavg*self.fft

def get_scope_rate(self):
    return self.scope_rate

def set_scope_rate(self, scope_rate):
```

```
        self.scope_rate = scope_rate
        self.wxgui_scopesink2_0.set_sample_rate(self.scope_rate)
        self.gr_keep_one_in_n_2.set_n(int(self.det_rate/self.scope_rate))

    def get_recfile_tpr(self):
        return self.recfile_tpr

    def set_recfile_tpr(self, recfile_tpr):
        self.recfile_tpr = recfile_tpr
        self.gr_file_sink_0.open(self.recfile_tpr)

    def get_recfile_kelvin(self):
        return self.recfile_kelvin

    def set_recfile_kelvin(self, recfile_kelvin):
        self.recfile_kelvin = recfile_kelvin
        self.gr_file_sink_2.open(self.recfile_kelvin)

    def get_rec_button_tpr(self):
        return self.rec_button_tpr

    def set_rec_button_tpr(self, rec_button_tpr):
        self.rec_button_tpr = rec_button_tpr
        self._rec_button_tpr_chooser.set_value(self.rec_button_tpr)
        self.blks2_valve_0.set_open(bool(self.rec_button_tpr))

    def get_rec_button_iq(self):
        return self.rec_button_iq
```

```
def set_rec_button_iq(self, rec_button_iq):
    self.rec_button_iq = rec_button_iq
    self._rec_button_iq_chooser.set_value(self.rec_button_iq)
    self.blks2_valve_2.set_open(bool(self.rec_button_iq))

def get_noise_amplitude(self):
    return self.noise_amplitude

def set_noise_amplitude(self, noise_amplitude):
    self.noise_amplitude = noise_amplitude
    self._noise_amplitude_slider.set_value(self.noise_amplitude)
    self._noise_amplitude_text_box.set_value(self.noise_amplitude)

def get_integ(self):
    return self.integ

def set_integ(self, integ):
    self.integ = integ
    self._integ_slider.set_value(self.integ)
    self._integ_text_box.set_value(self.integ)
    self.gr_single_pole_iir_filter_xx_0.set_taps(1.0/((self.samp_rate*se

def get_idecln(self):
    return self.idecln

def set_idecln(self, idecln):
    self.idecln = idecln
```

```
        self._idecln_text_box.set_value(self.idecln)

    def get_gain(self):
        return self.gain

    def set_gain(self, gain):
        self.gain = gain
        self._gain_slider.set_value(self.gain)
        self._gain_text_box.set_value(self.gain)
        self.uhd_usrp_source_0.set_gain(self.gain, 0)

    def get_freq(self):
        return self.freq

    def set_freq(self, freq):
        self.freq = freq
        self._freq_text_box.set_value(self.freq)
        self.wxgui_fftsink2_0.set_baseband_freq(self.freq)
        self.uhd_usrp_source_0.set_center_freq(self.freq, 0)

    def get_file_rate(self):
        return self.file_rate

    def set_file_rate(self, file_rate):
        self.file_rate = file_rate
        self.gr_keep_one_in_n_3.set_n(int(self.det_rate/self.file_rate))

    def get_fftrate(self):
```

```
        return self.fftrate

    def set_fftrate(self, fftrate):
        self.fftrate = fftrate
        self.gr_keep_one_in_n_1.set_n(self.fftrate)
        self.blks2_logpwrfft_x_0.set_avg_alpha(1.0/float(self.spavg*self.fft

    def get_det_rate(self):
        return self.det_rate

    def set_det_rate(self, det_rate):
        self.det_rate = det_rate
        self.gr_keep_one_in_n_0.set_n(self.samp_rate/self.det_rate)
        self.gr_keep_one_in_n_3.set_n(int(self.det_rate/self.file_rate))
        self.gr_keep_one_in_n_2.set_n(int(self.det_rate/self.scope_rate))

    def get_dc_gain(self):
        return self.dc_gain

    def set_dc_gain(self, dc_gain):
        self.dc_gain = dc_gain
        self._dc_gain_chooser.set_value(self.dc_gain)
        self.gr_multiply_const_vxx_0.set_k((self.dc_gain, ))

    def get_calib_2(self):
        return self.calib_2

    def set_calib_2(self, calib_2):
```

```
        self.calib_2 = calib_2
        self.gr_add_const_vxx_0.set_k((self.calib_2, ))
        self._calib_2_text_box.set_value(self.calib_2)

    def get_calib_1(self):
        return self.calib_1

    def set_calib_1(self, calib_1):
        self.calib_1 = calib_1
        self.gr_multiply_const_vxx_1.set_k((self.calib_1, ))
        self._calib_1_text_box.set_value(self.calib_1)

    def get_add_noise(self):
        return self.add_noise

    def set_add_noise(self, add_noise):
        self.add_noise = add_noise
        self._add_noise_chooser.set_value(self.add_noise)

if __name__ == '__main__':
    parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
    parser.add_option("", "--dcg", dest="dcg", type="eng_float", default=eng
        help="Set DC (post-detector) gain [default=%default]")
    parser.add_option("", "--devid", dest="devid", type="string", default="a
        help="Set USRP Device ID [default=%default]")
    parser.add_option("", "--srate", dest="srate", type="eng_float", default
        help="Set Sample Rate [default=%default]")
    parser.add_option("", "--spa", dest="spa", type="eng_float", default=eng
```

```
    help="Set Spectral Averaging Constant [default=%default]")
parser.add_option("", "--rxant", dest="rxant", type="string", default=""
    help="Set RX Antenna selection [default=%default]")
parser.add_option("", "--tpint", dest="tpint", type="eng_float", default
    help="Set Integration Time [default=%default]")
parser.add_option("", "--rfgain", dest="rfgain", type="eng_float", defau
    help="Set Gain of RF Front-End [default=%default]")
parser.add_option("", "--frequency", dest="frequency", type="eng_float",
    help="Set Center Frequency [default=%default]")
parser.add_option("", "--decln", dest="decln", type="eng_float", default
    help="Set Declination [default=%default]")
parser.add_option("", "--subdev", dest="subdev", type="string", default=
    help="Set USRP Subdevice ID [default=%default]")
parser.add_option("", "--maxg", dest="maxg", type="intx", default=50,
    help="Set maxg [default=%default]")
parser.add_option("", "--fftsize", dest="fftsize", type="intx", default=
    help="Set fftsize [default=%default]")
parser.add_option("", "--clock", dest="clock", type="eng_float", default
    help="Set Clock rate [default=%default]")
(options, args) = parser.parse_args()
tb = N200_TPR(dcg=options.dcg, devid=options.devid, srate=options.srate,
tb.Run(True)
```

APPENDIX B. STATISTICAL RESULTS

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

Supplemental Statistics

More stuff.