Wideband RF Front End Daughterboard Based on the Motorola RFIC

Terrence J. Brisebois

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Dr. Charles W. Bostian
Dr. Allen B. MacKenzie
Dr. William A. Davis

July 7, 2009
Blacksburg, Virginia

Keywords: software radio, USRP, radio hardware, SPI interface, Python, GNU Radio, Motorola RFIC, public safety, land mobile radio

Wideband RF Front End Daughterboard Based on the Motorola RFIC

Terrence J. Brisebois

ABSTRACT

The goal of software-defined radio (SDR) is to move the processing of radio signals from the analog domain to the digital domain – to use digital microchips instead of analog circuit components. Until faster, higher-precision analog-to-digital (ADCs) and digital-to-analog converters (DACs) become affordable, however, some analog signal processing will be necessary. We still need to convert high-radio frequency (RF) signals that we receive to low intermediate-frequency (IF) or baseband (centered on zero Hz) signals in order for ADCs to sample them and feed them into microchips for processing. The reverse is true when we transmit. Amplification is also needed on the receive side to fully utilize the dynamic range of the ADC and power amplification is needed on the transmit side to increase the power output from the DAC for transmission. Analog filtering is also needed to avoid saturating the ADC or to filter out interference when receiving and to avoid transmitting spurs. The analog frequency conversion, amplification and filtering section of a radio is called the RF front end. This thesis describes work on a new RF front end daughterboard for the Universal Software Radio Peripheral, or USRP. The USRP is a software-radio hardware platform designed to be used with the GNU Radio software radio software package. Using the Motorola RFIC4 chip, the new daughterboard receives RF signals, converts them to baseband and does analog filtering and amplification before feeding the signal into the USRP for processing. The chip also takes transmit signals from the USRP, converts them from baseband to RF and amplifies and filters them. The board was designed and laid out by Randall Nealy. I wrote the software driver for GNU Radio. The driver defines the interface between the USRP and the RFIC chip, controls the physical settings, and calculates and sets the hundreds of variables necessary to operate this extremely complex chip correctly. It allows plug-and-play compatibility with the current USRP daughterboards and supplies additional functions not available in any other daughterboard.

Acknowledgments:

There are many people I would like to thank – starting with Dr. Bostian, my advisor and the man who put me in this position and who made me an offer I couldn't refuse. I would also like to thank Dr. MacKenzie and Dr. Davis, my other committee members. All three have taught me more than I could ever explain, in and out of the classroom, and I owe them a tremendous debt. Dr. Bostian, in particular, has been a constant source of support and understanding.

Next, I would like to thank my co-workers at CWT and MPRG. Tom Rondeau and Bin Le, the lab's own royalty, were never too busy to explain anything I could possibly want to know. They were phenomenally intelligent and talented and there is no way I could have survived without them.

My current co-workers, Alex Young, Bin Li, Mark Silvius, Almohanad Fayez, Gladstone Marballie, Qin-qin Chen, Ying Wang, Sujit Nair, Rohit Rangnekar, and Aravind Radhakrishnan have all been wonderful to work with. They made the lab a friendly, open and welcoming place even under the direst of deadlines. Plus, Alex can go back in time.

This board couldn't have come about without the work of S.M. Shajedul Hasan and Randall Nealy. Randall did nearly all of the circuit design work, largely based on Hasan's board, and all of the layout work without a grumble and helped me enormously after the board had been made. Hasan's reference design for and work on the RFIC and his ever-willingness to share his knowledge was critical to the process.

Judy Hood's tireless and brilliant organizational and administrative efforts formed the backbone of the whole lab. Without her, we would be lost.

I would like to thank Matt Ettus and Eric Blossom as well. Matt is responsible for the development of the USRP, without which this project would not exist. His open-source designs and control software provided the basis for the design of the board and led me down the dreaded path of programming. Matt and Eric are also responsible for GNU Radio, from which every line of code I wrote is derived or stolen. They are also very nice guys.

At last, I would like to thank my friends and family. Most of you know who you are. To my parents, there are no words to describe how grateful I am or how much I am in your debt for your decades of love and support. To Neil Schafer, the Youngbloods, the Joneses, the Misitzises and the Johnsons, your friendship has kept me together over the years. I feel lucky to know every one of you and doubly so to consider you friends. To Mel Johnson, in particular, I thank you for always asking me what I was working on and pretending to understand what I said. Among other things.

And thank you, gentle reader. I hope you find this paper interesting or informative.

Grant Information:

This research was sponsored by the National Institute of Justice grant 2005-IJ-CX-K017 “A Prototype Public Safety Cognitive Radio for Universal Interoperability.” Any opinions, findings or recommendations expressed in this thesis are those of the author. They do not necessarily reflect the views of the National Institute of Justice.

It was also sponsored by the National Science Foundation grant CNS-0519959, “An Enabling Technology for Wireless Networks – the VT Cognitive Engine.” Any opinions, findings or recommendations expressed in this thesis are those of the author. They do not necessarily reflect the views of the National Science Foundation.

Contents:

List of Multimedia Objects:

List of Tables:

1. Introduction

The Universal Software Radio Peripheral, or USRP, is a hardware platform for software-defined radio applications. Called the "motherboard," the USRP itself has high-speed digital-to-analog converters (DACs) and analog-to-digital converters (ADCs). The ADCs allow it to sample, in order to receive and process, radio signals up to about 32 MHz in frequency (the ADC produces 64 million samples per second) and the DACs allow it to create radio signals, in order to transmit, up to about 64 MHz (the DAC produces 128 million samples per second) [1]. Because most radio signals are higher in frequency than 64 MHz, these frequency limitations means that the USRP needs an RF front end to down-convert received signals and to up-convert transmitted signals. Called "daughterboards," the interchangeable RF front end cards plug in to the motherboard and allow the USRP to operate in higher frequency bands and therefore transmit and receive real-world radio signals. A more comprehensive description of the USRP can be seen in Section 2.2 The USRP.

I wanted to build a new daughterboard for the USRP. This daughterboard was to make switching boards a thing of the past. The daughterboards currently available for the USRP operate in severely limited frequency ranges. Examples of boards we currently use in the Cognitive Wireless Technologies (CWT) lab at Virginia Tech are the RFX400 (400-500 MHz) and the RFX900 (800-1000 MHz) [5]. As a result, we frequently have to use multiple daughterboards when we want to transmit or receive in multiple frequency ranges. CWT has been developing software-defined radio solutions to the public safety interoperability problem. That problem occurs when different public safety radios are unable to communicate with one another. One aspect of the problem is that some public safety radios operate in the VHF range, around 150 MHz, others in the 700/800 MHz public safety band, others in the 400 MHz band. It is also desirable to operate in the FRS (Family Radio Service – off-the-shelf, commercially available walkie-talkies) range, around 460 MHz. Typical public safety radios can operate in one of these ranges, but not the other two. They can communicate with other public safety radios in only one frequency range. For our software radio solution, using the standard USRP daughterboards from Ettus Research, we would need at least three boards to cover those ranges. Since a USRP holds two daughterboards, and USRP2 holds only one [5], we would need to use multiple USRPs with multiple daughterboards or to switch out daughterboards, which requires unplugging the USRP and stopping any software radio application, in order to operate in all three bands. This is a serious problem. In order to build a practical public safety interoperability solution, we must be able to operate in all of the public safety frequencies without swapping boards. I wanted to build a new daughterboard which would be able to do that.

We looked into a variety of solutions. The RFX400 board can be modified to cover different frequency ranges. Simply replacing a set of inductors connected to the voltage-controlled oscillators (two inductors connected to the VCO on the transmit side, two connected to the VCO on the receive side) changes the center frequency of the board [4]. The frequency range remains about 25% of the center frequency, though, so with any one set of inductors, the modified RFX400 would still have a narrow frequency range. Based on the knowledge that the RFX400 could be modified to operate in different frequency ranges by swapping out inductors on the board, I worked with Innovative Wireless Technologies (IWT) of Lynchburg, Virginia to come up with a multi-band modification to the RFX400. The idea was to be able to swap in different sets of inductors on the fly and therefore to be able to switch frequency ranges without swapping daughterboards. IWT produced four prototype boards with four sets of inductance

values, which could be switched without removing the daughterboard or stopping GNU Radio. We never solved the problem of how to control the switches automatically with GNU Radio. Currently, they must be switched by hand, which is not acceptable for a real-world interoperability solution.

I heard about the Motorola RFIC in my Software-Defined Radio class, taught by Dr. Jeff Reed. This magical chip was purportedly able to do direct-conversion transmission and reception between 100 MHz and 2.5 GHz. It could do filtering and amplification, had five RF inputs and three RF outputs, and could be controlled through a single serial peripheral interface, or SPI, connection [5]. I immediately thought it should be the basis for a new USRP daughterboard. With a board based on this chip, we would be able to transmit and receive on independent channels simultaneously. We could receive a radio signal in the VHF band, re-modulate the data and re-transmit on the 700/800 MHz band without resorting to multiple daughterboards. This would be perfect for public safety. We could easily bridge between VHF, FRS and 700/800 MHz bands. The Motorola RFIC could solve all of our frequency problems.

2. Background

2.1 GNU Radio

Software-defined radio moves signal-processing tasks from analog circuits to digital circuits. ADCs and DACs transform data received by a radio front-end to the digital domain and from the digital domain to a radio front-end to be transmitted. Analog data must be processed by electronic circuits. Digital data can be processed by microchips such as general-purpose processors (GPPs), digital-signal processors (DSPs) and field-programmable gate arrays (FPGAs). These devices are flexible whereas analog circuits are not. Computers can be programmed to perform many different tasks, as long as they are defined by mathematical algorithms. Filtering, mixing, modulation and demodulation and phase-locking are just a few signal processing tasks that can be handled by computers. Each of those operations is essentially mathematical. In the last decade, computers have become fast enough and inexpensive enough to be able to perform those operations quickly and cheaply. Software radio has become practical.

GNU Radio is a free, open-source software radio development package. The GNU Radio homepage is http://www.gnu.org/software/gnuradio/. The full documentation and download instructions are available on that site. GNU Radio includes tools like filters, modulators, demodulators, phase-lock loops (PLLs) and many more. It also provides a framework for connecting these tools, called signal-processing blocks, together into a cohesive software-defined radio. The blocks are connected together in flow-graphs. Data received by, or transmitted through, a flow-graph goes through each block in a specified order. Calculations and conversions are performed by each block, preparing data for the next. This data flows from "sources" to "sinks." A data source can be a radio front-end, a noise-generator, a sequence of data, a sound card receiving audio from a microphone, or a file, among other things. A data sink might also be a radio front-end, a graph or chart shown to the user, a file, or a sound card, which would then output to a loudspeaker [6].

A visual representation of a GNU Radio flow graph, in Figure 1, may be seen below. This is the flow graph for a wideband analog FM receiver, containing three signal processing blocks, a source and a sink. GNU Radio first sets up each component. The USRP Source must be set so that the RF frequency is correct, the analog amplification is correct, the decimation is correct and the digital down-conversion is correct. The decimation rate and coefficients must be set for the channel filter. Another block, called *Guts*, contains a demodulator, an audio filter and a decimator. The demodulation rate and decimation rate must be set in the Guts block. The audio filter is a standard component and need not be set up. GNU Radio sets up the Volume Control block to output the desired audio volume, and sets up the Audio Sink, which may be any audio device. It then connects the flow graph together. When the flow graph starts running, an RF signal is transformed into digital samples by the USRP. Samples from the USRP Source block are sent to the Channel Filter. Filtered samples are sent to the Guts. Demodulated, filtered and decimated samples are sent to the Volume Control. Volume-controlled samples are sent to the Audio Sink. Audio is output, most likely by the sound card to headphones or speakers. This represents a complete, real-world radio receiver [7].

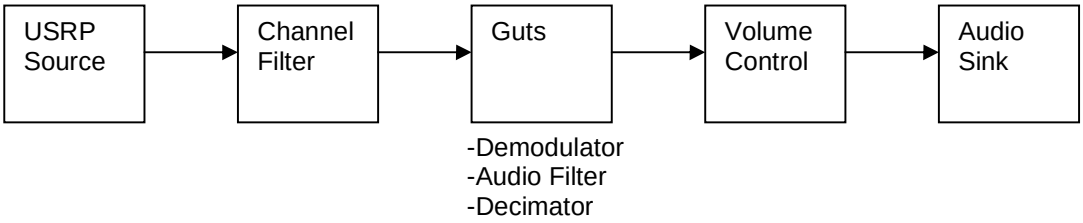

Figure 1: GNU Radio Block Diagram

A software-radio developer can use existing blocks, or create his own, to put together a software-defined radio with GNU Radio. It runs on standard PC hardware and is primarily used with Linux, but has also been ported to Macs and Windows PCs. It sets the bar for entry into software radio development to be pretty low. Anyone with a relatively modern computer can download it and start using it with a minimum of time and energy expended. Furthermore, it is widely used by researchers, hobbyists, students, teachers, and professionals, so many applications have been written, modifications made, and problems and solutions documented [6].

GNU Radio is written in Python and C/C++. Typically, processing blocks and other low-level functions are written in C/C++, because it runs faster than Python. Flow-graphs and high-level functions are written in Python, because it is easier to write. The flexibility of GNU Radio is nearly limitless. Because anyone can write a processing block and can connect blocks however they see fit, nearly any application is possible, limited only by the available radio front-end hardware and processing speed. Processing speed limits how fast any given block can be calculated. It also limits the ability of the software to perform multiple calculations, as in multiple signal-processing blocks, simultaneously. The radio hardware must either retrieve data from an antenna or send it out the same way or both. It limits data throughput, signal bandwidth, RF frequency, transmitting power, dynamic range, switching speed between transmit and receive as well as between frequencies, and the minimum detectable signal [6].

2.2 The USRP

The Universal Software-Radio Peripheral, or USRP, is a radio front-end designed to be used with GNU Radio. Like GNU Radio, the USRP is open-source, but unlike GNU Radio it is not free. Its design schematic, layout, and software controls are open-source and freely available with GNU Radio. The board itself, at $700, is inexpensive and flexible. Called a "motherboard," the USRP provides an interface to a host computer, a stage of interpolation/decimation and digital frequency up-conversion and down-conversion, analog-to-digital and digital-to-analog conversion (via ADCs and DACs), and several interfaces to "daughterboards." A daughterboard is an analog radio front-end designed to plug into the USRP. It does analog amplification, mixing and filtering. The daughterboard passes a signal from an antenna to the USRP, or vice-versa. The USRP passes a signal from the daughterboard to the computer, or vice-versa. Each USRP has two "sides," each with two connectors for daughterboards. Each side has a transmit (TX) and receive (RX) connector. Some daughterboards are transmit-only and use only a TX connector, some are receive-only and use

only an RX connector, and some are transceivers and use both. The USRP can support full-duplex communication on both sides simultaneously, or any subset thereof. Two USRPs may be connected together in a MIMO configuration, synchronizing clocks and daughterboards. Achieving MIMO, however, requires slight modification to the hardware, including the addition of SMA RF connectors on the motherboard to provide a clock input or output [8].

Figure 2, below, shows the major components of the USRP. The DC power port and USB 2.0 port are at the bottom. In the middle of the board is the Altera Cyclone FPGA. The Analog Devices Mixed Signal Processors, on either side of the FPGA, contain the ADCs and DACs. Four daughterboards are connected in the picture: two receive-only daughterboards and two transmit-only daughterboards. The upper-left and lower-right daughterboards are receivers. The RF interface, such as a connection to an antenna (in this case via SMA connectors), of the upper-left board is highlighted. On the upper-right and lower-left of the USRP, there are transmitter daughterboards. The RF interface of the upper-right board is also highlighted. A transceiver daughterboard would take the place of the TX and RX daughterboards shown – either the two boards on the right side (Side A) or the two boards on the left side (Side B) [9].

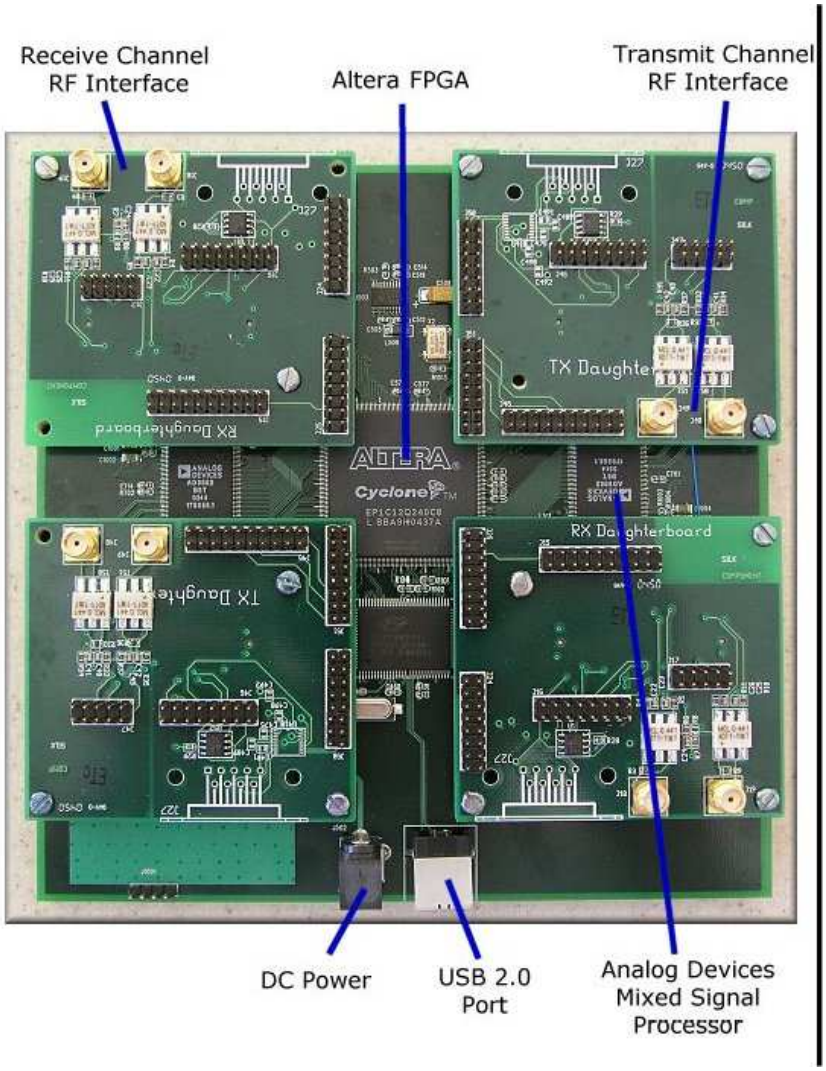

Figure 2: Picture of USRP, © Matt Ettus. Used with permission. See Appendix D: Permission from Matt Ettus.

A flow graph of the function of the USRP while receiving is shown in Figure 3. Initially, the USRP receives an analog signal from the attached receiver daughterboard, or RF front end. The signal received is a radio signal, which the user wishes to receive. It has been manipulated, typically in frequency and amplitude and through filtering, by the attached receiver daughterboard. This signal is located at a low intermediate frequency (IF), typically around 4 MHz, or at baseband (centered about 0 Hz). It can be in the form of a single signal or as two, quadrature (I "in-phase" and Q "quadrature" or 90-degree offset) signals. Quadrature signals make demodulation easier. The high-speed ADC chip digitizes the received signal or signals. This digital information is sent to the FPGA. First, the FPGA decimates the high data rate signal. It reduces the number of samples per second and either increases the precision to 16 bits, which is normal operation, or it can reduce the precision to 8 bits, which allows a higher sampling rate with the same overall data rate, which is desirable for some receiver implementations. If the digital received signal is at an IF, it is digitally down-converted to baseband. If the received signal is not quadrature, it is I-and-Q mixed to become quadrature. If

the received signal is quadrature and baseband, no down-conversion is necessary. The resulting digital signal sent out of the FPGA is digital, quadrature baseband. It is sent to the USB 2.0 controller, which sends the digital, quadrature baseband data to the host computer. GNU Radio, or a software radio package like it, can process this information. The baseband information would typically go into a filter followed by a demodulator, and then the raw data would be processed.

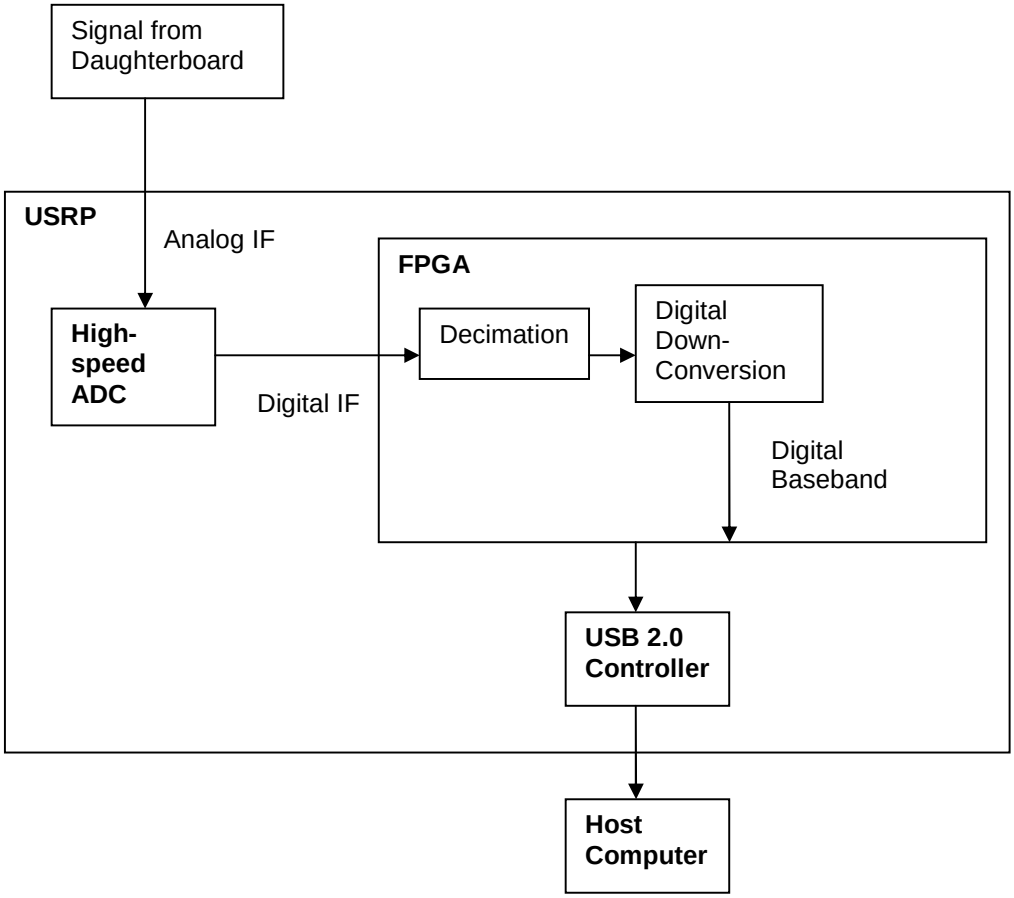

Figure 3: USRP Receive Block Diagram

The flow graph below, Figure 4, describes the operation of the USRP while transmitting. The host computer, presumably using GNU Radio, creates a digital signal to transmit over the air. This is a digital representation of the analog RF signal the user wishes to send. The data typically takes the form of I-and-Q, or quadrature, samples at RF. These I-and-Q samples would be created by the modulator. Usually 16-bit samples, they are sometimes 8-bit samples to allow a higher sampling rate with the same data rate, for instance if bandwidth is more important to the user than precision. First, the USB 2.0 controller receives the samples from the host computer. They are sent to the FPGA. The attached transmitter daughterboard may use a low IF or it may use baseband data. It may also require a single signal or quadrature signals. The GNU Radio software driver for the daughterboard would indicate whether the daughterboard uses a single signal or quadrature signals and what the intermediate frequency (IF) should be. If the daughterboard uses a low IF, then the FPGA will digitally up-convert the data to the IF. If it requires quadrature signals, the FPGA will leave the data as quadrature data streams. Otherwise,

the FPGA will combine the signals into a single stream. It interpolates (increases the sampling rate of) the data to take advantage of the high-speed DAC and converts the data to 14-bit precision, which the DAC uses. Out of the FPGA and into the DAC is sent a digital IF or baseband signal. The DAC converts it to an analog signal and sends it to the attached transmitter daughterboard to be transmitted [8].

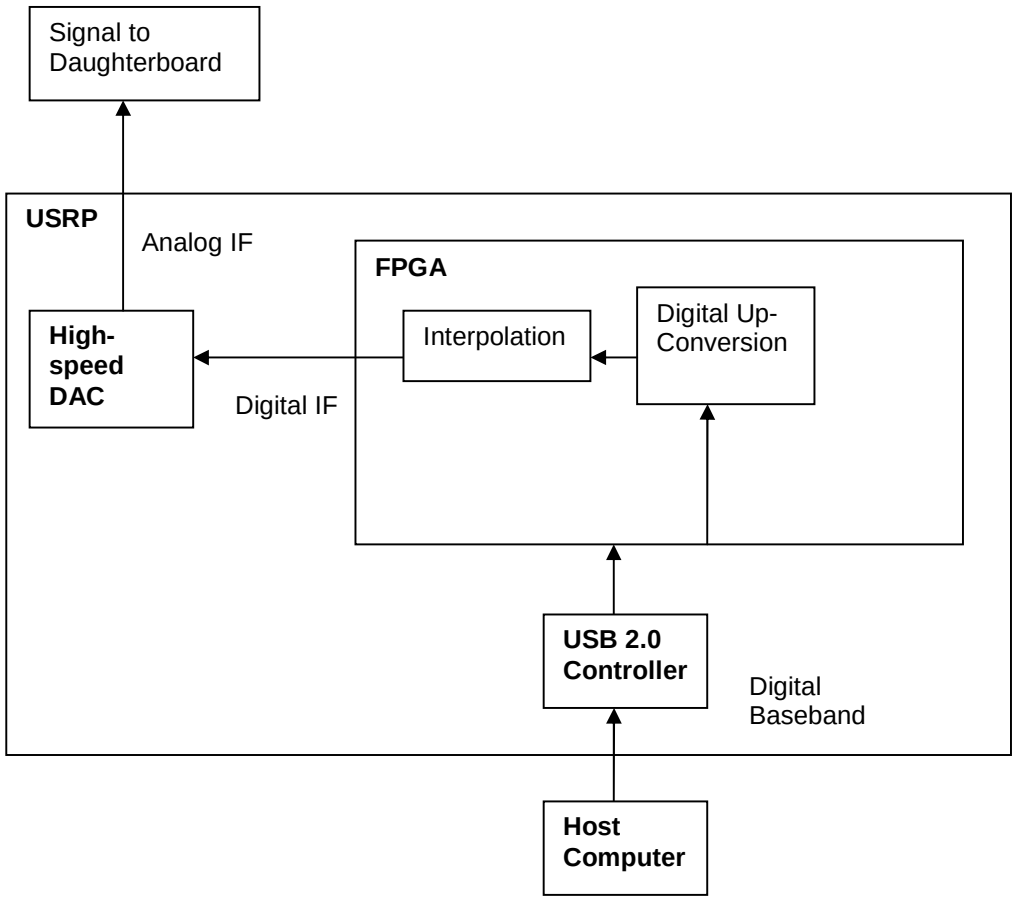

Figure 4: USRP Transmit Block Diagram

USB 2.0 provides the connection between the USRP and host computer. The maximum transfer rate over USB 2.0 is 32 MB/s, which includes both directions of communication between the host computer and USRP. Since samples are usually sent and received by GNU Radio as 16-bit, I-and-Q samples, this connection limits the sampling rate to about 8 Msamples/s. This means that the maximum RF bandwidth that can be transmitted or received at one time is 4 MHz. This number is reduced if the user wishes to transmit and receive simultaneously. It is also possible to transmit and receive with two daughterboards simultaneously, for up to 4 simultaneous radio connections: any combination of up to two receivers (one on each daughterboard) and two transmitters (one on each daughterboard) running at the same time, further reducing the data rate available to any one connection. The USB controller chip also includes SPI and I2C interfaces, which control the FPGA and can control functions on the daughterboards. Timing latency and limited data throughput are two major limitations of the USB connection [10].

The FPGA is connected to the USB controller. In a receiver, the FPGA takes high-speed samples from the ADC, typically representing data around a low intermediate frequency (IF), decimates (reduces the sampling rate), and does digital frequency down-conversion to baseband. The samples from the ADC are a digital representation of the low-frequency analog signal produced by the daughterboard. The samples sent out by the FPGA are a digital representation of the analog signal centered on DC. It also does I-and-Q mixing in the down-conversion stage, if needed, and adjusts resolution of the incoming samples. The decimation is necessary because data from the ADC is at a rate too high to transmit over the USB connection. Further, if the receiver and transmitter must be used simultaneously, or if both ADCs are in use simultaneously, the data rate must be reduced even further. Most daughterboards send analog data to the ADC at a low IF, rather than at baseband. This eliminates potential problems from DC offset, 1/f noise and shot noise that may otherwise occur in a daughterboard that converts the radio-frequency (RF) signal directly to baseband. GNU Radio, however, processes signals at baseband, which is to say that the signal is centered on 0 Hz, whereas the IF signal is typically centered near 4 MHz. In a transmitter, the FPGA receives data from the host computer over USB, does interpolation, frequency up-conversion and, if necessary, I-and-Q mixing. The interpolation occurs because the USB connection cannot send data to the DAC fast enough. Interpolation is needed to increase the sampling rate and provide the correct resolution. The frequency up-conversion is done for the same reason as the down-conversion in the receiver. Most USRP daughterboards take an analog IF signal from the DAC and up-convert that signal to RF, to avoid DC offset and other noise sources.

In the final part of the USRP, it converts data from analog to digital, or vice-versa. To do this, it uses a high-speed ADC/DAC chip. The ADC runs at 64 Msamples/s at a resolution of 12 bits per sample. The DAC runs at 128 Msamples/s at a resolution of 14 bits per sample. This stage converts analog, low-IF received signals to digital low-IF samples in the receiver and digital low-IF transmitted samples to analog low-IF in the transmitter. Both the ADC and DAC have two channels: one for I data and one for Q data. Any given daughterboard need only use one channel, but most use both. The FPGA, USB connection and computer can only work with digital samples. Daughterboards can only work with analog signals. The ADC/DAC chip also includes several low-speed ADCs and DACs, which can be used to control or monitor signals on the daughterboards. Low-speed ADCs can monitor received-signal strength indicators (RSSIs) or phase-lock detectors on the daughterboards. DACs can bias amplifiers or oscillators or control switches or on/off pins on chips. The ADC/DAC chip connects directly to the daughterboards [8].

The USRP2 is very similar to the original USRP, but with several marked improvements and two notable disadvantages. It uses higher-speed ADCs and DACs: 100 Msamples/s at 14 bits per sample and 400 Msamples/s at 16 bits per sample, respectively. A larger FPGA allows many more functions to occur on the board itself. The gigabit Ethernet interface improves timing accuracy and increases data throughput, which means a broader RF bandwidth, may be used. The built-in SRAM memory allows some degree of autonomous operation, that is, without a host computer. MIMO connections are easier due to a standard cable interface. The two disadvantages are cost and the fact that the USRP2 has only one set of daughterboard connectors. It can do full-duplex communication, but only with one daughterboard [3].

2.3 The Daughterboards

Some daughterboards receive radio signals from antennas, amplify and filter them, and down-convert them to a low IF or baseband and send them to the ADC. Others take baseband or low-IF signals from the DAC, up-convert them to a high radio frequency (RF) and amplify and filter them before transmitting them over-the-air with an antenna. Some do both. Some merely offer interfaces to external RF front ends. Nearly all USRP daughterboards are made by Ettus Research. Since the designs, schematics, layouts and controls are open-source, though, some researchers have built their own custom daughterboards. We at CWT have customized some of our own daughterboards, as described in the introduction, and in Section 2.4 Modifications, below, in partnership with Innovative Wireless Technologies. I have also modified several myself, by hand. The daughterboard I helped design, essentially from scratch, and wrote the controls for will have to be introduced in Section 3. The Driver, below. It will be the focus of this thesis.

A typical receiver daughterboard, or the receiver section of a transceiver daughterboard, operates similar to the flow graph below, Figure 5. It receives an analog RF signal via an attached antenna. This signal is filtered, typically with either a low-pass filter or a band-pass filter, to mitigate the effects of interfering signals. Because most received signals are low-amplitude, the low-noise amplifier increases the signal strength in order to use as much of the ADC's dynamic range as possible. The mixer down-converts the received signal to baseband or a low IF. It may also do quadrature mixing. This signal is, or these signals are, sent to the USRP for analog-to-digital conversion.

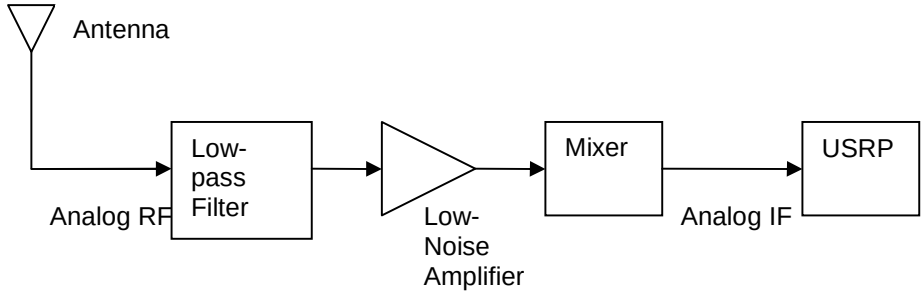

Figure 5: Daughterboard Receive Block Diagram

The flow graph below, Figure 6, describes a typical transmitter daughterboard, or the transmitter section of a transceiver daughterboard. An analog signal, which may or may not be quadrature, is sent from the USRP, produced by the DAC, to the daughterboard. The mixer up-converts the low-IF frequency or baseband signal to an RF frequency. If the signal from the USRP is quadrature, then the I and Q signals are typically combined in the mixer. I and Q differential signals are sent to the mixer, along with a differential local oscillator signal. The I and Q signals are simply summed in the mixer, in order to output a single RF signal [11]. The analog RF signal from the mixer is amplified in order to be powerful enough to be received by the intended receiver. A low-pass filter or band pass filter removes unwanted signals produced by non-linearities or noise in the daughterboard or USRP. The powerful signal is then sent over the air by an attached antenna.

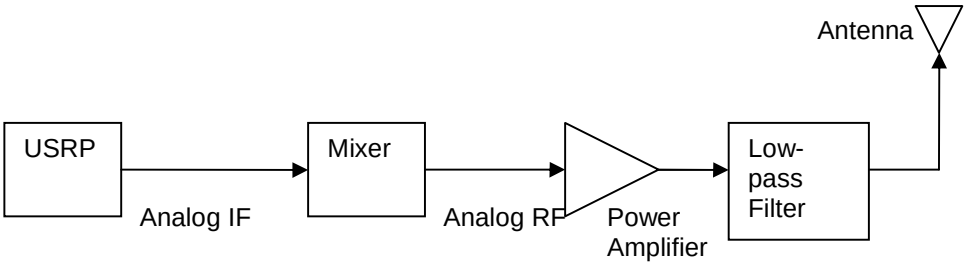

Figure 6: Daughterboard Transmit Block Diagram

Ettus Research produces several daughterboards along six product lines (soon to be seven). The first is the Basic series, with the BasicTX and BasicRX. These boards are half-duplex, but both may be installed into a single daughterboard slot (e.g. side A or side B). Their primary purpose is to interface with an external RF front end. They have neither amplifiers nor mixers nor filters. They do provide two SMA connectors each, to feed analog data into both channels of the ADC or to retrieve analog data from both channels of the DAC. Headers are also provided to easily access the SPI and I2C interfaces, the IO ports from the FPGA, the auxiliary low-speed ADCs and DACs, and both analog and digital ground. It is also possible to attach an RS232 serial communications connector. The boards can transmit or receive from about 1 MHz to 250 MHz, ideally connected to the IF stage of an external RF front end. These boards are capable of MIMO operation.

The LF series includes the LFTX and LFRX. These boards are nearly identical to the Basic boards, except that they include amplifiers and filters. They can transmit or receive from DC to 30 MHz, where the low-pass filters cut off. TVRX is a receiver only. With a frequency range of 50 MHz to 860 MHz, it is ideal for receiving TV signals or any signals in the VHF or UHF bands. The F-connector on this board provides a 75-ohm input for any standard TV or radio antenna. Its bandwidth is 6 MHz and includes automatic gain control (AGC), which may be controlled in software. It is not capable of MIMO. The DBSRX is a receiver that works from 800 MHz to 2.4 GHz. Bandwidth is adjustable in software from 1 MHz to 60 MHz. It is capable of MIMO operation. The SMA connector on this board can power an active antenna.

WBX boards, which are not yet available, include the WBX0510 and the WBX0822. They are half-duplex boards, so they can transmit and receive, but cannot do both simultaneously. Transmit power is expected to be 100 mW for both boards. The WBX0510 will operate from 50 MHz to 1 GHz and the WBX0822 from 800 MHz to 2.2 GHz. The wide frequency range of both transceivers makes them much anticipated. A recent addition to the USRP daughterboard line is the XCVR2450. It has two operating ranges: 2.4 to 2.5 GHz and 4.9 to 5.9 GHz. Also capable of transmitting 100 mW, it is similarly half-duplex. Both WBX and XCVR boards are MIMO capable.

Probably the most widely-used, most useful daughterboard line is the RFX series. These boards are full-duplex transceivers, capable of MIMO operation. They are all capable of transmitting about 100 mW. RFX boards include: the RFX400, which operates between 400 and 500 MHz; the RFX900, which operates between 800 MHz and 1 GHz, and includes a filter around the 902-928 MHz ISM band which can be bypassed; the RFX1200, which operates from 1150 to 1450 MHz; the RFX1800, which operates from 1.5 to 2.1 GHz; and the RFX2400, which

operates from 2.3 to 2.9 GHz, and includes a filter around the 2400-2483 MHz unlicensed band, which can be bypassed [2].

The picture below, Figure 7, shows an RFX400 daughterboard (formerly known as FLEX400). Being a transceiver daughterboard, it takes up two slots on the USRP: either the TX and RX slots on the right side (Side A) or the TX and RX slots on the left (Side B). Most of the circuitry on the upper half of the board is related to the transmitter. Most of the circuitry on the lower half of the board is related to the receiver. In the middle, there are switching circuits to enable the board to be used as a transmitter or a receiver or as both at the same time. Two SMA connectors are shown: labeled TX/RX and RX2. When used in half-duplex operation (meaning that it may transmit or receive, but not both at the same time), the TX/RX port is used for both transmitting and receiving signals. This allows a single antenna to be used, connected to this port, for both transmitting and receiving [12]. The RFX-series driver automatically operates the switches, in half-duplex operation, to make sure the TX/RX port is connected to the transmitter when transmitting and to the receiver when receiving. When used in full-duplex mode, the RX2 port is enabled. The transmitter uses the TX/RX port and the receiver uses the RX2 port. In this mode, two antennas must be used. If the transmitter and receiver were connected to the same port, and were operating simultaneously, the power from the transmitter would over-drive the receiver and possibly destroy it. Hence, when used in full duplex mode, both ports are enabled and the transmitter and receiver need not share. The 32-pin headers and nearby EEPROM chips are in the boxes near the upper left and lower left of the board. The headers allow access to the 16 digital input/output (I/O) pins on the FPGA, some of which are also used to control functions, such as switching, on the daughterboard. The EEPROM chip contains a unique identification for each type of daughterboard and subdevice [8]. A subdevice is either a transmitter or a receiver, so the EEPROM on the upper part of the board contains the identifier that it is an RFX400 daughterboard, transmitter subdevice. The EEPROM on the lower part of the board contains the identifier that it is an RFX400 daughterboard, receiver subdevice. These EEPROM chips connect directly to the TX and RX connectors, respectively, on the USRP and the information stored within them is used by GNU Radio to determine which software driver to use with which subdevice [13] [14].

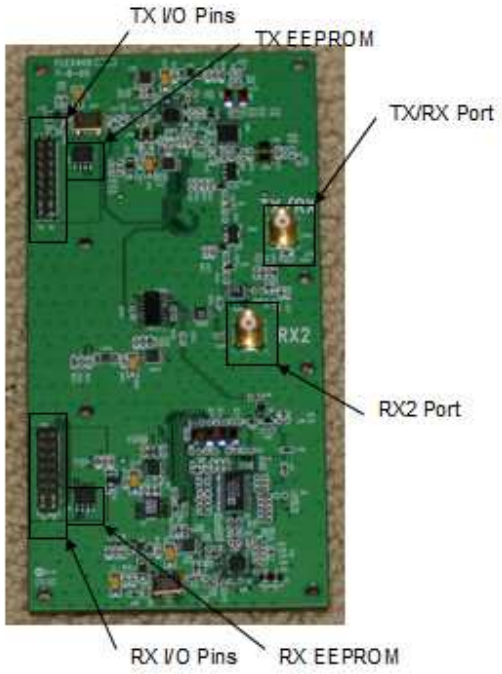

Figure 7: Picture of Daughterboard, © Matt Ettus. Used with permission. See Appendix D: Permission from Matt Ettus.

2.4 Modifications

The RFX400 can easily be modified to operate in different frequency ranges by changing the center frequency of the oscillator, though the frequency range remains about 25% of the center frequency. To control the center frequency of the VCO, one must replace two inductors. Figure 8, below, shows the locations of the inductors and VCO/PLL chip on the RX side of the board. The transmitter and receiver use independent VCOs, so in order to make a daughterboard send and receive on a specific, modified frequency band, one must replace inductors on both sides. By replacing the existing inductors with lower-value inductors, a higher center frequency is achieved. By replacing the inductors with shorts or 0 ohm resistors, thereby minimizing inductance, I have made RFX400 boards operate in bands as high as 693-1011 MHz. The RFX400 boards have low-pass filters with cutoff frequency around 520 MHz, so, in order to use them at higher frequencies, the filter must be disabled. Replacing the inductors with higher values, up to 33 nH (the maximum allowable, according to the VCO data sheet [15]), I have gotten RFX400 boards to operate in bands as low as 143-186 MHz. I believe these are the two extremes – the upper and lower limits of the RFX400's operating range [4].

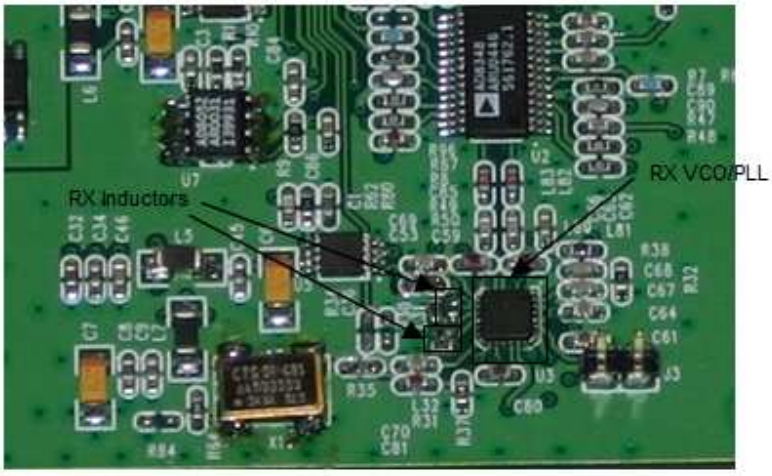

Figure 8: Close-up Picture of Daughterboard, © Matt Ettus. Used with permission. See Appendix D: Permission from Matt Ettus.

GNU Radio code is agnostic as to the frequency range of the board. When a program tells it to set a specific center frequency, GNU Radio simply tries to make the daughterboard attain that frequency. The RFX-series driver in GNU Radio is set up to drive the VCO frequency to multiples of 1, 2 or 4 MHz [16] [15]. This local oscillator (LO) frequency is typically 3 to 5 MHz above the desired center frequency when the daughterboard is transmitting. It is set to 3 to 5 MHz below the desired center frequency when the daughterboard is receiving. Using this reference frequency, the daughterboard converts the RF signal to a low IF, which is then translated to baseband by the FPGA on the USRP. After trying to set the LO frequency, the driver checks whether the PLL on the VCO has achieved lock at this frequency – whether it has successfully attained the desired frequency. If it has, GNU Radio reports success along with the actual LO frequency, so the FPGA may be set to digitally convert the IF frequency to baseband, and the program keeps going. If it hasn't achieved lock, the program reports that and quits. The driver tries to set the desired frequency regardless of what the frequency is or whether it is in the ostensible range of the specific daughterboard being used. For example, if a user with an RFX-series daughterboard wants to tune to a center frequency of 450 MHz, the driver will try to tune the LO to 454 MHz. If the user wants to tune to a center frequency of 150 MHz, the driver will try to tune the LO to 154 MHz. If the user wants to tune to a center frequency of 2000 MHz, the driver will try to tune the LO to 2004 MHz. It does this whether the board in use is an RFX400, an RFX900, an RFX1200, an RFX1800 or an RFX2400. The only difference in this regard between the RFX-series daughterboards and any modified boards is in whether the VCO will successfully attain those frequencies. Therefore, the modified RFX400 boards require no modification to the GNU Radio code. They are plug-and-play compatible with the original boards [16].

Based on this principle, I worked with Innovative Wireless Technologies (IWT) to produce a modified RFX400 board with multiple sets of inductors, which could be switched in at will. Since each set of inductors could have different values, switching between them would effectively change the frequency range of the daughterboard. IWT developed a "grand-daughterboard." This small PCB attaches to the inductor pads on the original RFX400. It has four sets of inductors and a solid-state switch to switch between them. In the prototypes, of which four were delivered, three frequency ranges were selectable: 181-218 MHz, 345-459

MHz, and 393-537 MHz. The last frequency range roughly emulates that of the original, unmodified board. The highest frequency range possible with this modification was limited by the inductance inherent in the grand-daughterboard circuitry. No switch setting could provide the low level of inductance that a 0 ohm resistor or a short could provide so no switch setting could achieve the highest frequency range possible on the RFX400.

We originally planned on using either the auxiliary DACs on the ADC/DAC chip or some of the accessible data IO pins on the FPGA to control the switch. The prototypes currently have manual, sliding switches. They work well, and consistently, but switching the frequency range by hand is awkward, especially if the daughterboard is inside an enclosure and the switches are not readily accessible. We had planned on inserting GNU Radio code to control the DACs or FPGA pins, but that change would have to be made in every program that used the modified boards. The program would have to know in advance that it was to be run only with these modified RFX400s, because using the digital IO pins with a daughterboard that uses them for another purpose could damage the board or the FPGA. It would also have to know in advance which frequencies were available with each switch setting and be able to make the switch before trying to achieve the desired frequency. Another possibility was to modify the RFX daughterboard controls that come with GNU Radio. Again, these changes would have to be made in every computer that used the modified boards. We also never quite figured out how to control the DACs or FPGA pins at the time, so the point was moot. These boards are able to hit some of the VHF band and the entire FRS band, but they have never been used in a practical situation. They work with unmodified GNU Radio code. No code changes are necessary to use these boards.

The RFX900, 1200, 1800 and 2400 use the same series of VCO, the ADF4360-x series. The ADF4360-3, -2, -1 and -0 are pin-identical. Unlike the ADF4360-7 in the RFX400, these chips do not have external inductors to set their frequency ranges. They are interchangeable in the RFX series boards. The RFX900 uses a -3, with a divide-by 2 frequency divider, to go from 800 to 1000 MHz. The RFX1200 uses a -0, with a divide-by-two, to go from 1150 to 1450 MHz. The RFX1800 uses a -3, with no frequency divider, to go from 1.5 to 2.1 GHz. The RFX2400 uses a -0, with no frequency divider, to go from 2.3 to 2.9 GHz. Each of these uses the same up-converter and down-converter mixer and amplifier. Exploiting this similarity, I worked with IWT to modify the RFX1800. It was chosen because it has no band-pass filter and its original frequency range is close to the one we desired. We replaced the original -3 chip with a -2 chip. The new boards were able to operate from 1770 to 2569 MHz [17]. This frequency range was desirable for a demo, and was not covered by the original RFX boards. Covering roughly 800 MHz of RF frequency, this range is fairly broad but does not cover several desirable frequency ranges for public safety, such as VHF, FRS and UHF. Again, these boards are compatible with GNU Radio and require no modifications to the code.

2.5 The RFIC

The problem with the current generation of USRP daughterboards is that they do not cover a sufficient frequency range. I wanted to build a new daughterboard that could cover the entire public safety frequency range. I wanted the new daughterboard to integrate fully into GNU Radio. It would require a driver, and enough changes to the GNU Radio code to recognize the board and the driver, but it would not require any changes to GNU Radio-based software radio implementations. Our current public safety radio programs shouldn't need to be modified. The solution to the problem can be found in the Motorola RFIC. We have been using version

RFIC4a. It is a fully integrated radio transceiver on a chip. Programmable through a Serial Peripheral Interface, or SPI, the direct-conversion transmitter and receiver can operate in RF frequencies from 100 MHz to 2.5 GHz. Adjustable baseband filtering and amplification is available on the receive side, as is adjustable baseband filtering and amplification and RF power amplification on the transmit side. DC offset correction can be done on both sides. Direct digital synthesis (DDS) is available on the transmitter along with a Cartesian feedback system to optimize linearity and DC offset.

Figure 9, below, shows the basic inputs and outputs of the Motorola RFIC chip. On the left side of the diagram, there are five RF receiver inputs. Each of these inputs has different properties and may be used to meet different requirements. On the right side, there are three RF transmitter outputs. These outputs have different properties and, again, may be used to meet different requirements. Also on the right side is the baseband I/O. Having down-converted a received RF signal from one of the inputs, the RFIC outputs the baseband signal for processing. A signal to be transmitted is sent to the baseband transmit input of the RFIC. It is up-converted and then it is put out through one of the RF transmit outputs. On the top is the input for the reference clock. If a 31.25 MHz crystal oscillator is used as the reference, its frequency is multiplied by 32 to result in a 1 GHz frequency reference. Alternatively, a 1 GHz reference frequency may be used. The bottom of the diagram shows the Serial Peripheral Interface (SPI), through which most of the functions of the RFIC are controlled.

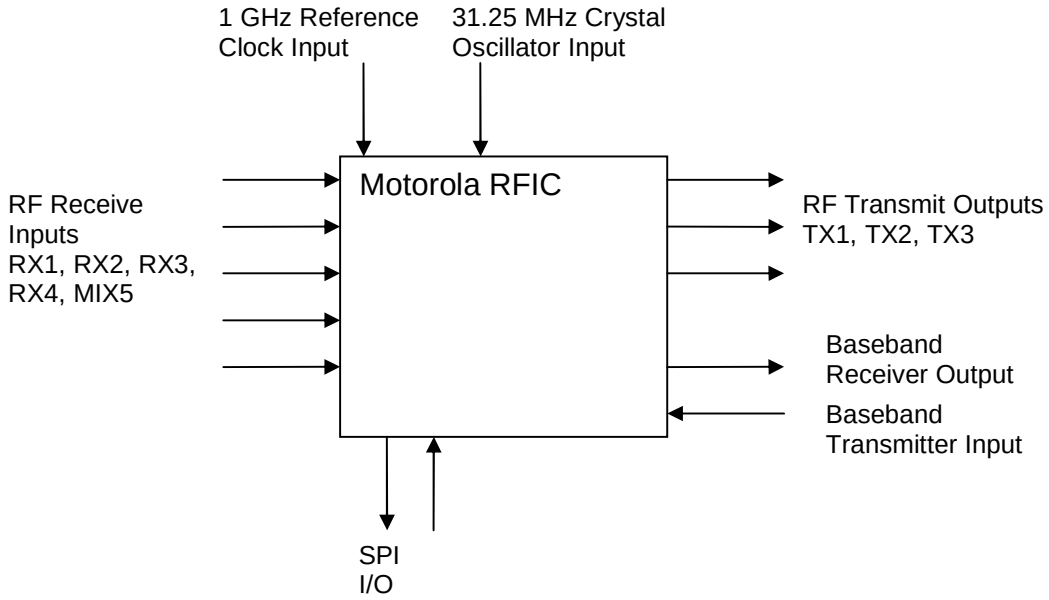

Figure 9: RFIC Input-Output Diagram

SPI, or Serial Peripheral Interface, is a method of communicating between two or more microchips. In this case, the SPI interface controls over 200 8-bit registers, not including memory. These registers, in turn, control nearly every aspect of the chip's operation: from filter bandwidth to amplification; from synthesizer frequency to DC offset correction. Each 8-bit register may contain up to eight variables and therefore may control up to eight aspects of the chip's operation. It is vitally important to set every register accurately for correct operation. A

single variable incorrectly set can easily mean the difference between correct operation and no operation.

The exceptional range of frequencies, 100-2500 MHz, is achieved with a digital frequency synthesizer, adjustable between 200 MHz and 1 GHz. Furthermore, the oscillator frequency can be divided by two, used as-is, multiplied by two, or multiplied by four. This is how it is able to achieve such a large frequency range. Three of these synthesizers act as the local oscillator for the mixers in the receiver, transmitter and transmitter feedback systems. These synthesizers are normally driven by a 31.25 MHz crystal oscillator. The oscillator would be on the same board as the RFIC, but it is not integrated into the RFIC chip itself. Its frequency is multiplied by 32 to provide a 1 GHz reference, from which the synthesizers can produce frequencies from 200 MHz to 1 GHz. It is also possible to connect a 1 GHz external reference. This would serve the same purpose.

There are five receive paths and three transmit paths. The receive paths each go through different LNAs, except for the MIX5 input which has no LNA, and different mixers before being multiplexed into the same baseband path. After the multiplexer on the receive side, the signal, now at baseband, is sent through three amplifying filters. A diagram of the receive path can be seen below, in Figure 10.

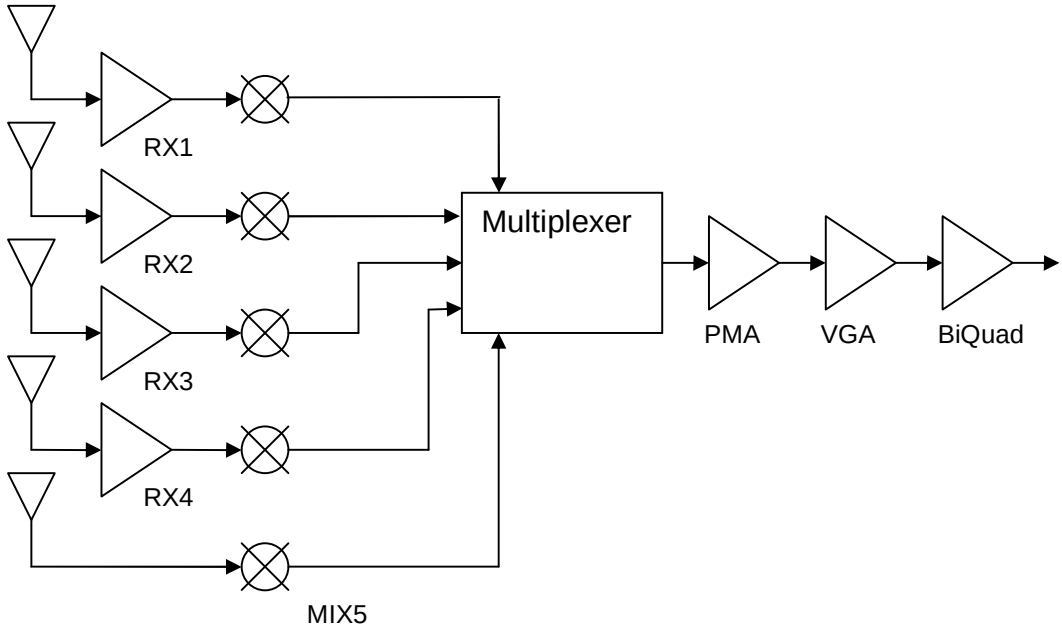

Figure 10: RFIC Receive Block Diagram

Since the chip is made in CMOS, the amplifier in each of these filters would normally add significant noise. DC offset, flicker noise and 2nd-order distortion are all added at low frequency by any CMOS amplifier. Since the RFIC does direct-conversion, this noise would be added to the desired signal. In order to combat this, each of the amplifying filters incorporates a "chopping" function, which can be turned on or off at will in any of the filters. The chopper mixes the desired signal up to a low IF before the amplifier stage, and then mixes it back to baseband after the amplification. Since the signal itself is amplified at a frequency well above DC, the DC offset noise, flicker noise and 2nd-order distortion is added out-of-band. CMOS amplifiers do not add significant distortion at higher frequencies. When the signal is mixed back

to baseband, the noise is mixed out of band and filtered out. This process allows CMOS direct-conversion receivers to avoid the problems that normally plague such implementations. The effects of the chopping mixer can be seen in the picture below, Figure 11: a comparison of the frequency response of the receiver with no input signal with and without the chopper enabled. The figure shows a plot of the noise floor of the RFIC, with the 1/f low-frequency noise represented by the peak in the middle, at 0 kHz. At 0 kHz, the noise is clearly higher when the chopper is disabled. Low-frequency noise in general is also higher when the chopper is disabled.

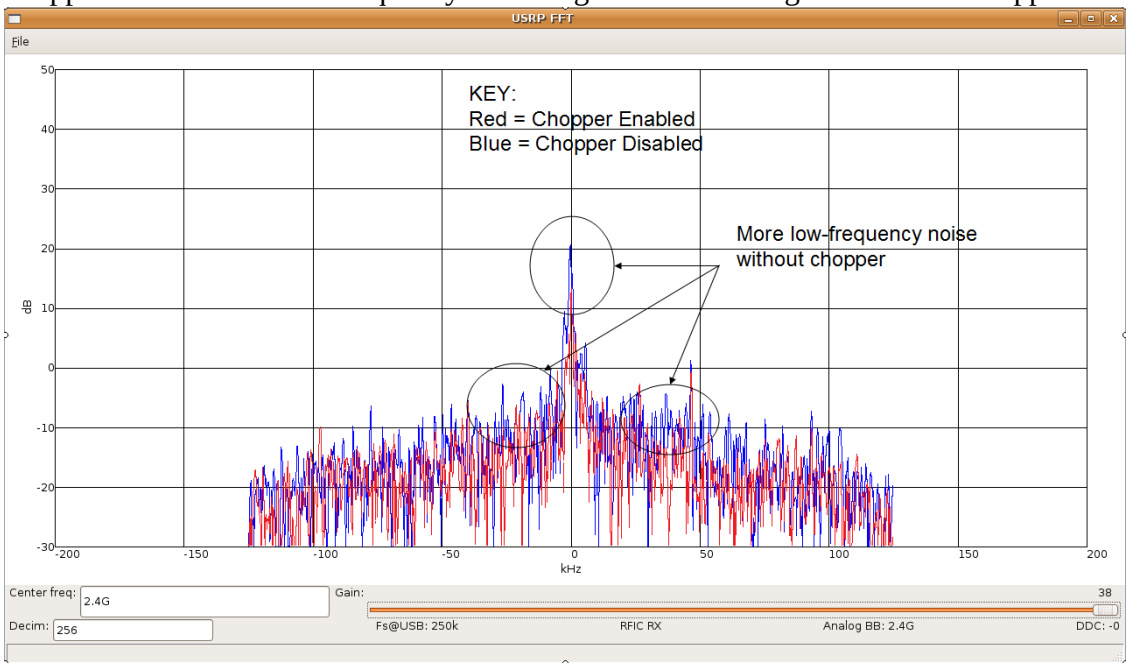

Figure 11: Spectrum Graph, With and Without Chopper

Three different transmit paths are designed to transmit in different RF frequency ranges and with varying levels of power control. Again, they use different mixers and power amplifiers but are multiplexed into the same baseband transmit path. Two of the paths are designed for low-frequency operation, one with a high degree of power control, the other with a lower degree of power control. The first transmit path, TX1, is designed to work from DC to 3 GHz. It has 80 dB of power control, 35 dB of which is continuous, and the other 45 dB of which is stepped in increments of 5 dB. TX2, the second transmit path, is designed to work in the same frequency range as TX1, but with better linearity and only 45 dB of power control, stepped in increments of 5 dB. The last transmit path, TX3, is designed to work from 2 GHz to 6 GHz, the theoretical upper frequency limit of the RFIC. It has lower linearity than TX1 or TX2 and the same 45 dB of stepped power control as TX2. A diagram of the transmit path can be seen below, in Figure 12.

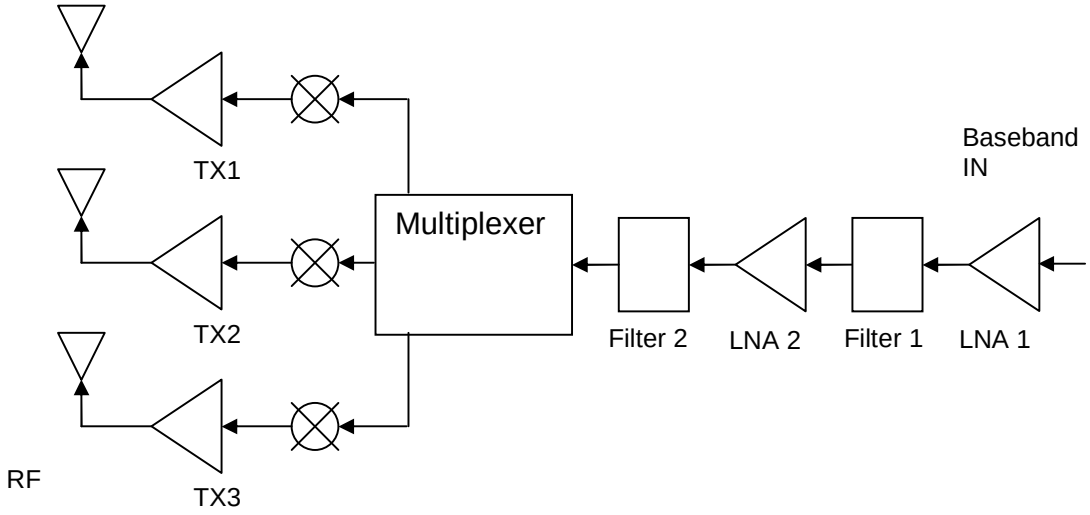

Figure 12: RFIC Transmit Block Diagram

In addition to the transmitter and receiver, the RFIC incorporates a feedback loop for the transmitter. It can take signals from the transmit path, just before they go off-chip, mix them back to baseband, amplify and filter them, and output them on the RX output pins. From there, they can be converted with the off-chip receiver ADCs and processed. This path is designed to allow the user to correct DC offset, gain and phase imbalances and distortion without relying on an external receiver or guess work. DC offset-correction DACs are available on both the transmit side and receive side of the RFIC. The step size is adjustable and they can correct DC offset in the I and Q paths independently [18] [19].

3. The Driver

3.1 Goals

I wanted a new daughterboard for the USRP that would cover all of the frequency bands we use in the lab. The expression my advisor, Dr. Bostian, is fond of is "DC to daylight." At a typical 400-790 THz, visible light frequencies are a bit of a stretch. Nevertheless, typical radio use covers frequencies from VHF to UHF to microwave. A conservative range would be 100 MHz to 2.5 GHz. Our lab frequently uses public safety frequencies in the 140 MHz range and Bluetooth or 802.11 devices in the 2.4 GHz unlicensed band and a multitude of frequencies in between. With our current range of daughterboards, we would need a dozen or so different boards to completely cover the RF spectrum we normally use. This means using, and frequently switching between, multiple types of daughterboard.

The Motorola RFIC offered a way to end the constant swapping of daughterboards. With coverage from 100 MHz to 2.5 GHz, a single daughterboard based on this chip could send and receive signals in every band the RFX400, RFX900, RFX1200, RFX1800 and any number of modified RFX400s could cover if put together. Only the RFX2400, with a frequency range of 2.3 to 2.9 GHz, can hit frequencies outside the range of the RFIC.

Minimum detectable signal (MDS) and output power are just as important as frequency range. The RFX boards have MDS around -130 dBm and output power ranging from about 50 mW to 200 mW. My goal for the RFIC was -120 to -130 dBm MDS. The RFIC can only output about 10 mW, so Randall Nealy (the research engineer who designed and laid out the daughterboard) included optional RF power amplifiers on the board, capable of outputting 100 mW. Achieving these goals would make the RFIC-based daughterboard comparable to the RFX boards in every way.

Most importantly, I wanted the RFIC board to be plug-and-play compatible with the RFX-series and other daughterboards in GNU Radio applications. This was the focus of my own work. I wrote the GNU Radio driver for the RFIC-based daughterboard. Written in Python, the driver uses similar functions to those for the RFX, WBX and XCVR-series daughterboards. At a bare minimum, a transceiver board must be able to control transmitter power, receiver amplification, and transmit and receive frequencies.

The RFIC-based daughterboard, designed by Randall Nealy, incorporates the Motorola RFIC4a chip, as described above in the Section 2.5 The RFIC. It has RF antenna ports for all three transmit paths and for all five receive paths. The version of the RFIC on this board does not have the RX4 receive path enabled, but it may be enabled in other versions of the chip. Therefore, there is a place to install an antenna port for the RX4 path, but no antenna port is currently installed. The daughterboard design incorporates received signal-strength indicator and transmit/receive switching circuitry. Current prototypes of the board only have one EEPROM chip, which is on the receive side. This means that GNU Radio is unable to recognize the transmit subdevice of the daughterboard automatically – it must be forced to use the RFIC daughterboard driver I wrote in any transmitter program.

My code provides functions to independently turn the transmitter and receiver on and off, and switch between any of the five receive paths and any of the three transmit paths on the RFIC. It also provides automatic transmit/receive switching, which is to be added as an external switch in an upcoming revision of the board design and layout [20]. Transmit and receive phase offset functions are also available. The phase offset of each frequency synthesizer may be changed

independently at will. Functions are also available to control the bandwidth of either path independently. Another unique function provided on this board by my code is feedback. The RFIC has a feedback path from the transmitter to the receiver, allowing a user to offset I-Q imbalance, characterize and implement pre-distortion, or check linearity. A function within the receiver subdevice allows the user to bypass the normal receiver path and down-convert the signal transmitted from by the RFIC to baseband for analysis. Separate functions are available to set the RF frequency to be fed back to the receiver, set the bandwidth of the feedback signal, and set the gain of the feedback path. Another function turns off the feedback loop and resumes normal receiver function. Finally, there is a received signal-strength indicator function. The complete code can be seen in Appendix A: The Driver Code.

3.2 Code Overview

The flow graph below, Figure 13, shows a basic representation of how my code works. The thick boxes represent a state. When a state is reached for the first time, a function is performed. The thin boxes represent a function. Arrows represent possible changes of state or functions performed. The thick *Start* box represents the initial condition – GNU Radio may be running but the daughterboard has not been initialized. If the daughterboard is turned off from the *Initialize* state, the synthesizer frequency multipliers are turned off and the program returns to the *Start* state. When the GNU Radio program tries to use a subdevice (transmitter or receiver), the state moves to *Initialize*, where FPGA registers are set to control automatic transmit/receive switching. Many registers are set on the RFIC, but none of the filters or mixers is enabled. Next, the state moves to *Transmit* or *Receive*, depending on whether a transmit subdevice or a receive subdevice is being initialized. In both of these states, mixers and filters are turned on, chopping clocks are turned on and set, and several additional variables are set. At this time, the program must set the transmit frequency or the receive frequency. Power and amplification default to the maximum setting, bandwidth defaults to the widest setting and there is no phase offset by default. Any of these settings can be set by the program from this state. After performing any of these functions, the program returns to the *Transmit* or *Receive* state. If, however, the subdevice is turned off, or deleted, the filters and mixers and choppers are turned off and the driver returns to the *Initialize* state.

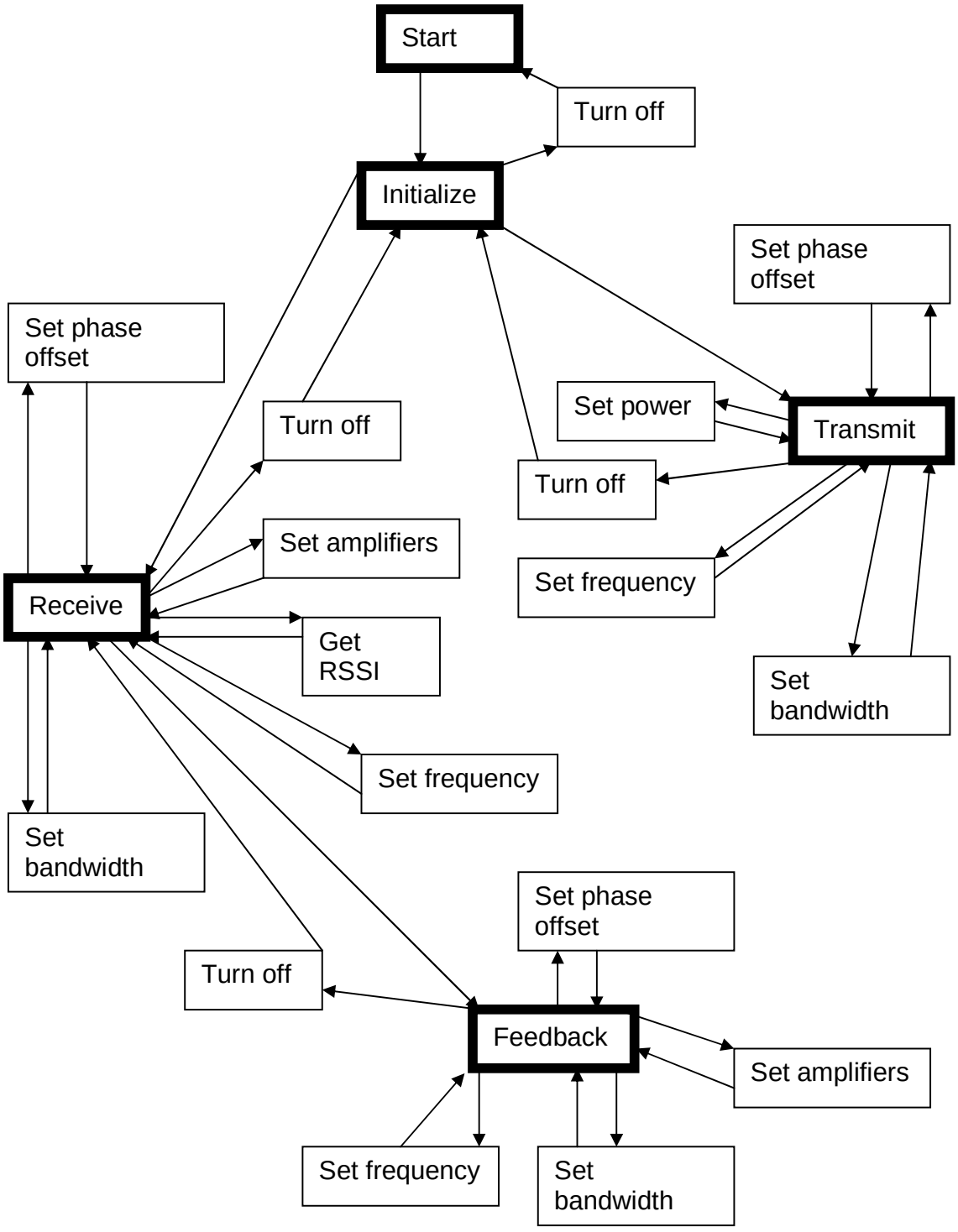

Figure 13: Driver Flow Graph

The receiver has additional functions not found in the transmitter. Feedback can be turned on. Moving to the *Feedback* state, the driver turns off the receiver filters and turns on the feedback from the transmitter. The output of the feedback loop to GNU Radio uses the same pins as that of the receiver, so the receiver must be turned off to analyze information from the feedback loop. Phase offset is set to zero by default, amplification is set to maximum and bandwidth is set to the highest setting. Frequency must be set by the user. At this point, the data received by the program is a representation of the transmitted signal, amplified, filtered and converted to baseband by the feedback mixers and amplifiers and filters. When the program returns to *Receive* mode, the feedback chain is turned off and the receiver filters turned back on. A final function checks a Receive Signal-Strength Indicator (RSSI). This returns two variables: one related to how often the signal is in fade, or has low signal strength; the other related to how often the signal is clipping, or has high signal strength. Both of these values are instantaneous measurements of a low-pass filtered pulse-width modulated (PWM) signal. The PWM signals from the clip and fade detectors are low-pass filtered, then sampled at a single time instant. This results in two instantaneous values related to how often the signal is clipping and fading. The RSSI function will be described in more detail in the Section 3.4.1 The RFIC Object, below.

3.3 Interface In-Depth

Three things control every aspect of the RFIC daughterboard's functions: the SPI interface; the IO pins on the USRP; and the auxiliary ADCs and DACs on the USRP ADC/DAC. The SPI interface is used to control all of the registers and nearly all of the internal settings [18] [19]. IO pin 6 on the receive side controls the automatic TX/RX switching. One of the auxiliary DACs on the ADC/DAC controls the continuous gain on the TX1 transmit path. Two of the auxiliary ADCs poll the received signal-strength indicators [21]. The high-speed ADCs and DACs are the actual I and Q received and transmitted radio signal paths. The diagram below, Figure 14, shows the IO ports on the USRP used by the RFIC daughterboard.

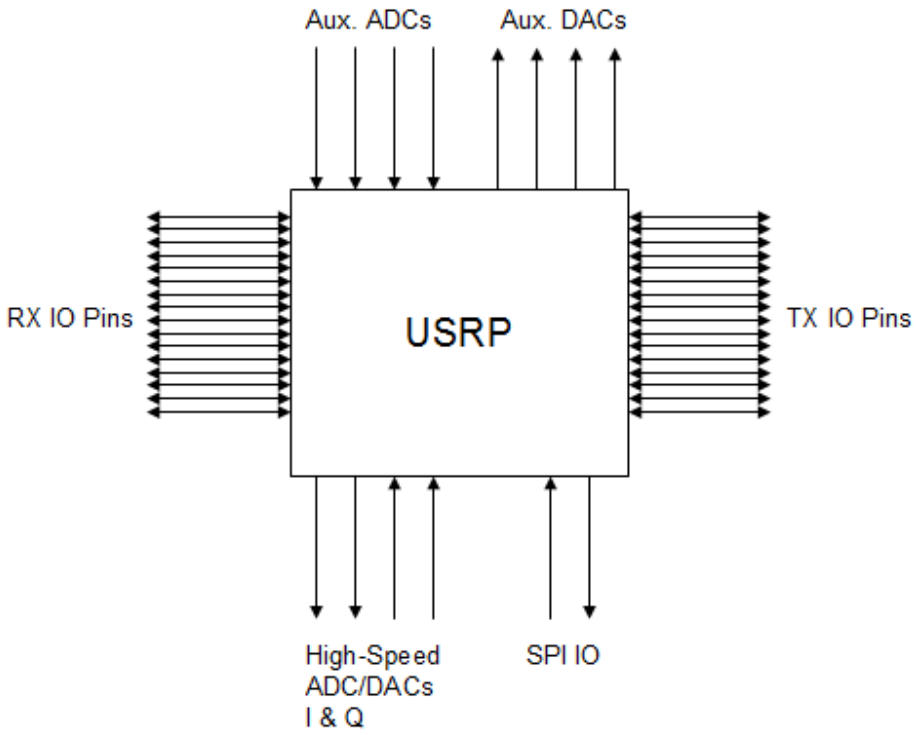

Figure 14: USRP IO Diagram

SPI, or Serial Peripheral Interface, is a standard used to communicate between electronic devices. There are five digital lines: MOSI, or master out, slave in; MISO, or master in, slave out; SCLK, the clock output by the master device; CSEL, or Chip Select; and an optional digital reset line. The master is the FX2 USB 2.0 controller chip on the USRP [22]. The RFIC is a slave. All slave chips are controlled by the master. The MOSI, MISO and SCLK lines are shared between all devices. Master selects which slave device it will output to or take input from with the CSEL lines. SPI is typically used to read and write data registers on microchips. The USRP can interface with four devices via SPI: via the TX port on side A; via RX on side A; via TX on side B; and via RX on side B. The SPI interface is controlled through GNU Radio in Python. GNU Radio includes functions for the USRP to read and write SPI registers with optional headers [21]. The RFIC chip on the RFIC daughterboard is connected to the RX SPI port.

Each SPI register on the RFIC contains one byte, or eight bits, of information. To write a register on the RFIC, two header bytes are written, then up to 64 bytes of data, which would therefore set up to 64 registers. The first bit of the header is the write disable bit. It should be set to zero to write a register. The last bit is an address auto-increment disable bit. If auto-increment is disabled, one may only write or read a single register in a single pass. If auto-increment is enabled, one may write or read up to 64 registers in a single pass. The middle 14 header bits contain the number of the register one wishes to write. The next bytes, up to 64, contain the data to write to the registers. For example, if, in the header, the write disable bit is set to zero, the auto-increment bit is set to zero, and the register number is set to 0 and 64 bytes of data are sent after the header, then registers 0 through 63 on the RFIC would be set with the

64 bytes of data [23]. The table below, Table 1: SPI Write Operation, contains a description of the SPI write operation.

Table 1: SPI Write Operation

SPI Write Operation					
Type:	Header				Data
Byte Number:	0	0	1	1	2 – (1 + n) *(n up to 64)*
Bits (MSB first):	0	1 – 7	0-6	7	0-7
Contents:	Read enable	Register number		Auto-increment disable	Write data

Reading an SPI register requires no header. The starting address is set by the previous write operation. So, too, the auto-increment is set by the previous write operation. Up to 64 bytes, which is to say 64 registers, may be read in a single pass if the auto-increment is enabled. If the auto-increment is disabled, only one register may be read in a single pass. The last register written will determine which register will be read first. For instance, if the previous write operation set register 0 only and enabled the auto-increment, the subsequent read operation could read n registers, from 0 to (n-1), where n is up to 64. However, if the previous write operation set registers 0 through 63 and enabled the auto-increment, the subsequent read operation could read n registers 63 to (62 + n), where n is up to 64 [23]. The format of the data can be seen in Table 2: SPI Read Operation, below.

Table 2: SPI Read Operation

SPI Read Operation	
Type:	Data
Bytes:	0 – (n - 1) *(n up to 64)*
Bits (MSB first):	0 - 7
Contents:	Read data

There are 261 8-bit SPI registers on the RFIC containing 354 separate variables. Some of the registers are not occupied or have not yet been assigned. Others may contain as many as eight separate variables. The variables control most aspects of the RFIC's functions. They control the frequency of each of the three oscillators (transmit, receive and feedback), frequency multipliers and dividers, phase offset, gain in most of the amplifiers, bandwidth, and many other functions. It is vitally important to set every variable correctly in order to ensure correct operation of the chip [18] [19].

There are a total of 32 digital IO pins available to a transceiver daughterboard: 16 on the RX side and 16 on the TX side. Each may be used as an input or an output and may be controlled or polled either manually or automatically through registers on the FPGA. As outputs, they can be set to 3.3 volts or to 0 volts. As inputs, they simply return a 1 or a 0, depending on the voltage applied. At present, the receive IO pin IO_RX_06 is used to control automatic TX/RX switching.

The ADC/DAC chips on the USRP have high-speed ADCs for receiving IF or baseband radio signals and high-speed DACs for transmitting IF or baseband signals. They also have four

auxiliary ADCs and four auxiliary DACs each for controlling various functions on the daughterboards. The TX and RX connectors on each side of the USRP each have two low-speed ADC lines available. They share the four low-speed DAC lines. The ADCs have 10-bit precision and sample at 1.25 Msps while three of the DACs on each side of the USRP have 8-bit precision and the fourth has 12-bit precision. The 12-bit DAC controls 35 dB of gain in the TX1 transmit path. The two ADCs on the receive side sample the on-channel clip and on-channel fade pins, which provide received-signal strength indicators (RSSI) and may be used in automatic gain control [8] [21].

3.4 Code In-Depth

There are four major parts of my RFIC daughterboard driver for GNU Radio: the RFIC object, which is shared by the transmitter, receiver and base class and includes most of the control functions; the base class, from which the transmitter and receiver subclasses are derived; the transmitter subclass; and the receiver subclass. There is a fifth, more minor, part – the auto-instantiation function. Each major part contains multiple functions – at the very least initialize and delete functions – and provides tools for GNU Radio users. Many of the tools are plug-and-play compatible with existing GNU Radio software, but several are unique to the RFIC board and this driver. They are easily accessed and provide increased functionality compared to existing daughterboards and existing daughterboard drivers.

3.4.1 The RFIC Object

Used by the other three parts of the driver [16] [24] [25], the RFIC object contains functions to read and write the SPI registers, generally, on the RFIC chip, functions to write every register specifically on the RFIC, a function to initialize the RFIC with specific values and definitions of all of the RFIC variables, a function to shut down the RFIC, functions to set up the automatic transmit/receive switching, and functions to set the receive and transmit gain, frequency, phase offset, and bandwidth. It also contains functions to enable and disable the feedback loop and set its gain, frequency, phase offset and bandwidth. Finally, it contains a function to poll the RSSI pins and return numbers related to the clip and fade. As shown in the figure below, Figure 15, the TX Subclass, RX Subclass and Base Class use functions contained in the RFIC Object to perform the radio operations of the daughterboard. Furthermore, the complete RFIC object code can be seen in Appendix A: The Driver Code, under the heading **`class rfic(object):`**.

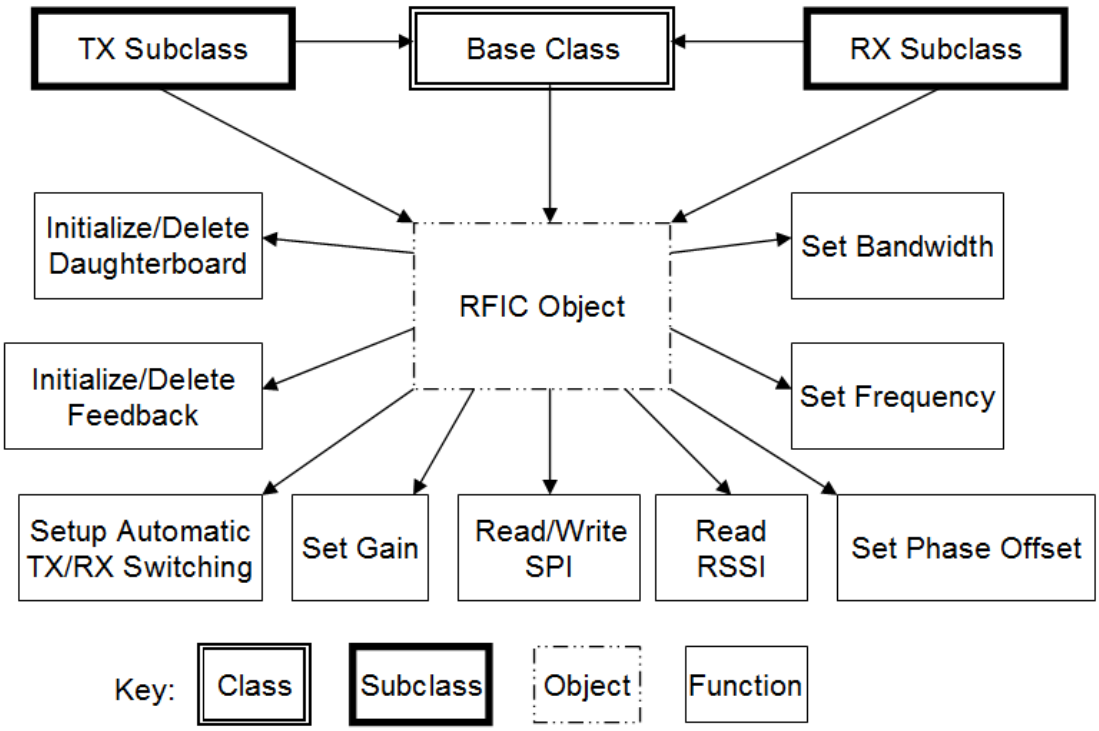

Figure 15: RFIC Object Diagram

The RFIC object calculates two variables before it can write SPI registers – the enables and the format. These variables tell the FX2 USB 2.0 controller chip, which controls the SPI interface, which slave chip to enable on the USRP and how to format the data. The RFIC uses the RX port to interface with SPI. A variable passed to the RFIC object by GNU Radio tells it which side (A or B) the daughterboard is on. The enable variable is calculated from the RX port setting and whether the daughterboard is on side A or B. For the SPI write function, a two-byte header is required and the format is most significant bit first (MSB). The format variable is calculated based on those two requirements. For the SPI read function, no header is required, but the format is still MSB. Another format variable is calculated based on the requirements when reading an SPI register.

When reading an SPI register, there is no header. The register at which reading begins is set by the previous write command. Therefore, the SPI read function first writes register number 0 with variables in the appropriate bit locations. It then reads 64 registers into a string five times: the first read gets the contents of register 0 through register 63 and puts them into the string; the second gets the contents of register 64 through 127; the third gets the contents of register 128 to 191; the fourth gets the contents of register 192 to 255; the fifth gets the contents of register 256 to 319. While the highest register number used is 261, 320 registers are read in case a future version of the chip uses more registers. The function then returns the contents of the chosen register number. Several SPI registers in the RFIC are read-only and it doesn't make sense to write a register immediately before reading it. That is why the first register, a read/write register of which the contents are known at all times, is written before every register is read. It ensures that read-only registers are never written.

The SPI write function has two inputs: the number of the register to write and the data to write in that register. It calculates a two-byte header, as described in Section 3.3 Interface In-Depth, to write before the data. The enable and format variables are calculated within the RFIC object, so they may be used by and are automatically passed to the write function. Finally, the function writes the header, then writes a single byte of data to the register. While the SPI interface is capable of writing up to 64 registers in a single pass, I decided to limit each write to one register. For n up to 64, the function would have to write n bytes of data to n registers. This would allow multiple variables in multiple registers to be changed in a single pass, saving time, but it would vastly complicate the functions calling the SPI write function. This design decision is explained below.

All 354 variables located in SPI registers must be defined by the RFIC object. Every function, class and subclass must have access to every variable. In order to change any variable, and therefore to set any SPI register, the program must know what other variables, if any, occupy the same register. It must also know the value of these variables and what bits they occupy. This makes changing a single register a complicated task. Up to eight variables may share the same register, so a function setting a single register must keep track of up to eight variables and their bit positions. This is why I made one unique function for every register.

There are two kinds of registers on the RFIC: read/write and read-only [18] [19]. Because the value of every written variable is known at all times, it is not necessary to read any read/write register in order to determine the value of an associated variable. Only the read-only registers must be read. The function associated with a read/write register will calculate the data to send to the register from the associated variables and their bit positions. It will send this data along with the register number to the SPI write function. The SPI write function will write the correct data to the register. A function associated with a read-only register will use the SPI read function to determine the contents of the register and then calculate the values of the associated variables from their bit position and bit length. It then sets the associated variables in the RFIC object so that they may be read by any function, class or subclass.

The initialization function for the RFIC object starts by setting a pointer to the instance of the USRP sink or source it is associated with. It also sets up a variable with a zero if the daughterboard is on side A and a one if it is on side B. It calculates SPI format variables for writing and reading SPI registers based on the fact that the RFIC uses MSB formatted data and the write function requires a two-byte header while the read function requires no header. An SPI enable variable, shared by SPI write and read functions, is calculated based the fact that the RFIC uses the SPI interface on the RX side of the daughterboard and whether the daughterboard is on side A or side B. Next, every variable associated with a read/write SPI register on the RFIC is defined with a specific value within the RFIC object. This way, every variable is available to every function. Next, initial values for the transmitter, receiver and feedback frequency variables are set, so as to be available to functions, but the frequency synthesizers are not set up. The clock frequency, 1000 MHz, is also defined here, for reference. Automatic TX/RX switching is set up next, as will be described later in this section, on the IO pin IO_RX_06 (voltage high for TX, voltage low for RX). Finally, every read/write register is written using its individual associated function. All of the registers are set up with reasonable default values, but none of the filters, mixers, choppers or other power-consuming devices is turned on.

The delete function is simpler. It sets the reset variables for each of the three frequency synthesizers, then writes their associated SPI registers. Next, it turns off the frequency multipliers associated with the three frequency synthesizers. It writes the associated registers.

There is no need to turn off filters or mixers or other power-consuming devices – the deletion functions for the transmitter, receiver and feedback will take care of those.

One function is defined for every SPI register on the RFIC. Each of these functions takes no inputs and returns nothing because the variables they use are stored in the RFIC object – they are always available. Because some registers are meant to be written and others are read-only, there are two kinds of functions related to the SPI registers. A write function simply puts the appropriate variables, from the RFIC object, contained within its associated register together into a single, one-byte value. It then uses the SPI write function to write this value to the associated register. A read function uses the SPI read function to determine the contents of its associated register. It then calculates the value of all variables associated with the register and sets the variables in the RFIC object. For example, the function set_reg_0 puts the value of variable Ngt3 into bit 7 of a one-byte number. It puts the value of variable NorNdiv4 into bits 0 through 6 of the one-byte number. Then, it writes the result to register 0. The function read_reg_208 determines the contents of register 208. It sets variable rx_Icmpo to the value of the register, bit 5. Then, it sets variable rx_Iodac to the value of the register, bits 0 through 4. An example of the operation of the functions that set the individual SPI registers is shown in the figure, Figure 16, below. The functions that read individual SPI registers work in the opposite manner.

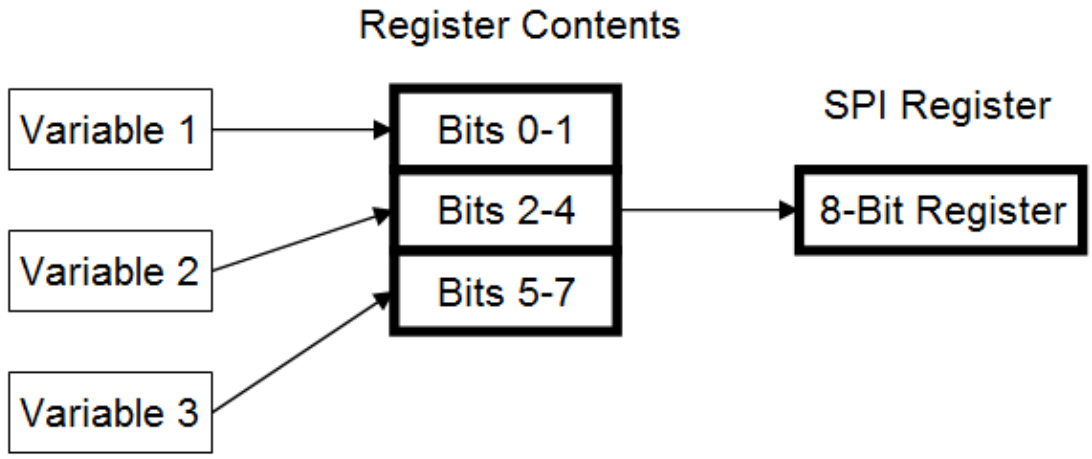

Figure 16: Register Set Function Example

Several functions set up the automatic TX/RX switching. TX/RX switching can use the IO pins on the TX or RX side of the daughterboard, connected to the FPGA. Both the TX and RX connectors on the USRP include 16 digital IO pins which may be set to inputs or outputs. They can be used to turn on and off amplifiers and mixers, to interface with external switches to control antenna arrays, or, as in this case, they can be used for transmit/receive switching.

While the RFIC board has five receive ports and three transmit ports, in the form of MMCX connectors, for the five receive and three transmit paths on the RFIC chip, it is often desirable to use a single port for both transmitting and receiving signals. A single port used for both purposes means only one antenna is required. Using a single port for transmitting and receiving signals means there must be a switch. If a high-power transmitted signal were running to a port being used as a receiver input, the receiver would be over-driven or possibly destroyed. When using a single antenna port for both transmitting and receiving, only one of those functions may be used at a time. To achieve this, Randall installed a switch on the RFIC daughterboard. It has two inputs – one to connect to a TX port and one to connect to an RX port. These inputs also

take the form of MMCX connectors, which means that MMCX cables would have to be run from the RX port and TX port of choice to the RX and TX inputs, respectively. The output is also MMCX, and may be connected to an antenna. The switch is controlled by the IO pin IO_RX_06.

Included in the FPGA programming on the USRP is an automatic transmit/receive switching protocol. It allows the FPGA to automatically control the digital IO pins on the transmit and receive sides of the USRP. This allows much faster switching compared with setting the digital IO pins through GNU Radio and relying on the USB 2.0 connection to set IO registers on the FPGA. Essentially, any of the digital pins on the TX or RX side of the USRP may be set up to be controlled automatically by the FPGA for transmit/receive switching. First, the IO pins that control the switching are set up as output pins. GNU Radio has an output enable function to do this for the 32 IO pins. Next, two masks are set in the FPGA – one for the transmit IO pins, one for the receive IO pins. These masks set up which pins in the TX IO and RX IO are controlled automatically. Finally, two settings for the IO pins are set: one for the transmit condition and one for the receive condition. By default, the pins are set to the receive condition. When data is present in the FPGA's first-in-first-out (FIFO) data buffer for the transmitter, the pins are automatically reset to the transmit condition. When data is no longer present in the FIFO buffer, the pins revert to the receive condition.

For the RFIC daughterboard, the receive side pin 6 is set up as an output pin. Then, a mask is sent to the FPGA setting up the IO_RX_06 to be the only pin controlled by the automatic transmit/receive switching. Next, the receive and transmit conditions are set up. In the receive condition, the pin is set to zero, or 0 volts. In the transmit condition, the pin is set to one, or 3.3 volts. When there is no data in the FPGA's transmit FIFO buffer, the pin is set to zero, ensuring that the RX port of the switch is connected to the output. When there is transmit data in the FIFO buffer, the pin is set to one, ensuring that the TX port of the switch is connected to the output. A diagram of this operation can be seen in the figure below, Figure 17.

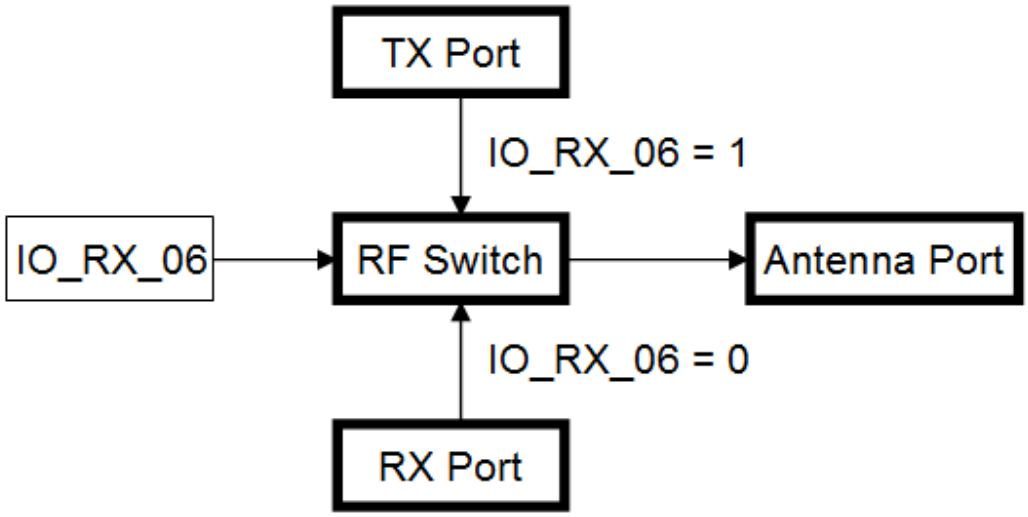

Figure 17: Automatic TX/RX Switching Diagram

There are three functions related to gain: one for the transmitter; one for the receiver; and one for the feedback path. They are located in the common RFIC object so that a receiver flow

graph may set the transmitter gain, or vice-versa. All three operate on similar principles. Each function sets the gain in the filters in the RFIC. They set the variables in the RFIC object, and then call the related functions to set the registers in the chip itself. The transmit path TX1 is unique in that it has two types of gain control: one controlled by this function through the SPI registers; the other controlled by one of the auxiliary DACs on the USRP, which is controlled in the transmitter initializer. This gain control will be discussed in the Section 3.4.3 The TX Subclass, below.

Two variables control the gain in the TX path, the additional gain control via the DAC notwithstanding. On register 176 lie variables that control stepped attenuation in the RF section of the RFIC. They can provide from zero dB of attenuation to 45 dB of attenuation. This translates to 45 dB of gain control. The transmit gain control function is written to set gain in increments of 5 dB, from 0 dB (45 dB of attenuation) to 45 dB (0 dB of attenuation). It adjusts these two variables based on the input, desired gain in dB, to the nearest value available in the 5 dB steps.

The receiver gain is controlled by four variables. The gain is adjustable in the three filters, as shown in Figure 10, above: the BiQuad filter; the VGA filter; and the PMA filter. DC offset step size, another variable, must be adjusted based on the gain of the BiQuad filter. BiQuad filter gain is set in dB from 0 to 18 dB. VGA gain goes from 6 to 14 dB. The PMA filter gain is set by a ratio of resistor settings. It allows up to 10 dB of further gain. Put together, there is 38 dB of gain control available in the receiver. The function uses an input, in the form of dB gain, to calculate the closest available value of gain and set the variables and registers accordingly.

Feedback gain control is simpler. There are four variables on a single register. They control gain directly, in 5 dB steps, from 10 to 20 dB each. Therefore, there is 40 dB of total gain control in the feedback loop. Again, the function uses a dB gain input, calculates the closest available gain value, and sets the variables and RFIC registers accordingly.

Frequency is set by two functions in the RFIC object for each of the three frequency synthesizers. Using direct digital synthesis, and based on the 1 GHz reference, the synthesizers can produce frequencies from 200 MHz to 1 GHz. Those frequencies can then be multiplied by one of four multipliers: a ½x multiplier; a 1x multiplier; a 2x multiplier and a 4x multiplier. There is also an 8x multiplier, but it is not used at this time. The broad frequency range of the RFIC stems from the frequency range of the frequency synthesizer, multiplied by the range of multipliers. At the low end, the frequency synthesizer can produce a 200 MHz signal, which can then be multiplied by ½ to send a 100 MHz signal to its mixer. At the high end, the frequency synthesizer can produce a 1000 MHz signal, which can then be multiplied by 4 to send a 4 GHz to its mixer. Practically, however, the transmitter frequency is limited to 2.5 GHz. Above this frequency, the transmit power drops off sharply. The receiver can operate up to 4 GHz. Theoretically, with the 8x multiplier, this chip would be able to cover a frequency range of 100 MHz to 8 GHz. This would require extremely wideband RF amplifiers and mixers. A larger transmit frequency range, perhaps up to 4 GHz or higher, may be practically implemented in a later revision of the RFIC chip [18] [19].

Setting the frequency of the local oscillator for the receiver, transmitter or feedback loop is therefore two steps: first, the frequency synthesizer must be set to the correct frequency; second, the correct multiplier must be set up. Setting the frequency synthesizer involves calculating three values, which are then placed into six variables on five registers. A unique function exists to calculate the values for each variable. The values are calculated in the same

manner for all three signal paths. The reason three values require six variables and five registers is that one of the values is 26 bits – much larger than the 8-bit registers – and is therefore split up into four different variables on four different registers. Two of the synthesizer values control a divide-by-four frequency divider within the frequency synthesizer. The third value controls the frequency of the synthesizer. This frequency calculator function first determines whether the desired synthesizer frequency is more than $1/4^{th}$ of the reference clock frequency, in this case 1 GHz. If the desired frequency is more than $1/4^{th}$ of the clock frequency, the synthesizer frequency need not be divided by four. The three values are calculated from there. If the desired frequency is less than 1/4th of the clock frequency, the synthesizer frequency must be divided by four and the three values are calculated based on that division. The 26-bit value is split into four parts to fill the four corresponding variables. The values with which to set these four variables are returned, along with values with which to set the other two variables, simply equal to the other two calculated values. All six values are returned by the function. This function is used by the transmitter, receiver and feedback frequency set functions to calculate the variables to set each frequency synthesizer [26].

There are unique functions to set the LO frequency of the receiver, transmitter and feedback loop. They are placed in the shared RFIC object so that, for example, a receiver flow graph may set the transmitter frequency. Each of the three functions operates in a similar manner. The input is the desired frequency, in Hz. A diagram showing the procedure can be seen in the diagram below, Figure 18. First, the function determines which multiplier to use. If the desired frequency is below 500 MHz, the 1/2x frequency multiplier is used. If it is between 500 MHz and 1 GHz, the 1x frequency multiplier is used. If it is between 1 GHz and 2 GHz, the 2x frequency multiplier is used. Above 2 GHz, the 4x multiplier is used. Second, the function sets up the correct frequency multiplier. This involves setting six variables in six registers, unique to the receiver, transmitter or feedback loop. It is a two-to-four step process. In the first step, all six variables are set to specific values. In the second through fourth steps, a single variable is adjusted to its final value. Every time a variable is set or adjusted, its corresponding register is also set. Third, if the 1x, 2x or 4x multiplier is in use, it sets eight variables across six registers to specific values for each multiplier. These are optimizing variables, which will be described in the Section 3.5 Tuning and Optimization [26]. Fourth, the function calculates the frequency synthesizer value, based on the desired frequency and the multiplier value, using the function described above. Fifth, it sets a global variable in the RFIC object to the desired frequency, in Hz. This variable stores the signal path frequency for reference. Sixth, the six variables that set the frequency synthesizer corresponding to the transmitter, receiver or feedback loop are calculated with the above function. Seventh, the corresponding registers are set. Finally, the function returns a true value and the desired frequency in Hz. The first value, true or false, should indicate whether the daughterboard successfully attained the desired frequency. It is only set to false if the desired RF frequency is above 4 GHz, where the RFIC is unable to operate. This is a flaw in the RFIC – there is no way to know, within the chip, whether the desired frequency has been successfully attained. Even if the chip fails to attain the desired frequency, the function must return a success. The second value, in this case the desired frequency, is used by GNU Radio to calculate the digital up-conversion (DUC) or down-conversion (DDC) frequency. The difference between the desired frequency and this returned value is the IF frequency that the DUC or DDC must convert from or to baseband. Since the RFIC is a direct-conversion chip, no digital up-conversion is necessary in the transmitter and no digital down-conversion is necessary in the receiver.

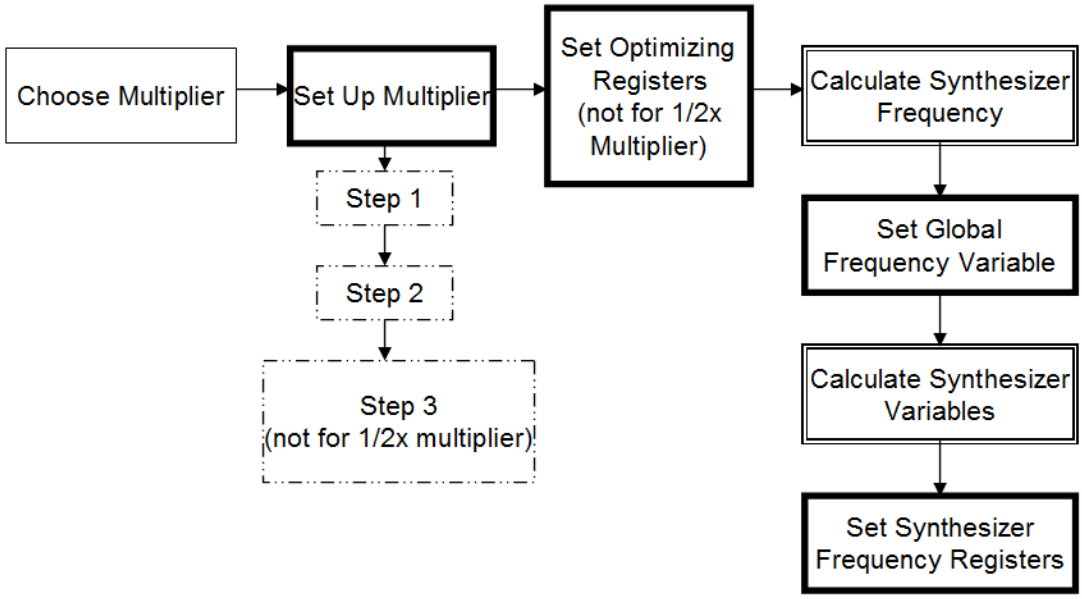

Figure 18: Set Frequency Procedure Diagram

Synthesizer phase offset in any of the three paths can be set in the RFIC with three values, stored in six variables across five registers. A function exists to calculate the values to correctly set the phase offset. Phase offset is calculated in a similar manner to frequency, and the phase offset calculation is identical for each of the three frequency synthesizers. Six values are calculated based on the current synthesizer frequency, the clock frequency, and the desired phase offset, in degrees. Again, the values are different depending on whether the synthesizer frequency is above or below 1/4th of the clock frequency. The values needed to set all six variables are returned and are used by the functions that set the transmitter, receiver and feedback phase-offset.

Each of the transmitter, receiver and feedback loop paths have a function that passes the clock frequency, the current synthesizer frequency, and the desired phase offset to the phase offset calculator function above. They calculate the current synthesizer frequency from the current LO frequency and the multiplier used. The LO frequency for each path is saved as a global variable, and is therefore readily available to the phase offset function. If the corresponding LO frequency is below 500 MHz, the 1/2x multiplier is in use and the LO frequency must be multiplied by 2 to find the synthesizer frequency. If the LO frequency is between 500 MHz and 1 GHz, the 1x multiplier is in use and the LO frequency is equal to the synthesizer frequency. If the LO frequency is between 1 GHz and 2 GHz, the 2x multiplier is in use and the LO frequency must be divided by 2 to find the synthesizer frequency. If the LO frequency is above 2 GHz, the 4x multiplier is in use and the LO frequency must be multiplied by 4 to find the synthesizer frequency. Using the above phase offset calculator function, each function determines the necessary values for all six variables. They set the corresponding variables with the six returned values, and then set the corresponding registers on the RFIC. The functions automatically return a success. The RFIC ought to be able to correctly set any desired phase offset from 0 to 360 degrees, but there is no way to determine whether the offset has been successfully set.

The RFIC object contains functions to set the bandwidth of the transmitter, receiver and feedback loop. These functions are in the RFIC object so that either a transmitter or receiver flow graph may set the bandwidth of any of the three signal paths. In each case, several bandwidth steps and the settings that correspond with them are available, as provided by the documentation from Motorola [18] [19]. The filtering occurs at baseband in every signal path. The functions take in a desired bandwidth, in Hz, set the bandwidth variables to the closest available step, and set the corresponding registers. Bandwidth adjustment functions are not available in the standard daughterboard drivers. These functions are unique to this driver and this daughterboard.

For the receiver, bandwidth is set via the adjustable resistor and capacitor settings in each of the three amplifying filters (BiQuad, VGA and PMA, as seen in Figure 10, above). In setting the bandwidth of any filter, coarse adjustments are made by changing resistor values and fine adjustments are made by changing capacitor values. Only four resistor values are available in setting the PMA filter bandwidth. Eight resistor values are available in setting the bandwidth of the other two filters. In contrast, the PMA capacitor can be set to 4,096 different values, the VGA capacitor can be set to 1,024 different values and the BiQuad capacitor can be set to 512 values. Altogether, this allows a very fine adjustment to the total system bandwidth. The maximum bandwidth is 14.46863 MHz. The minimum bandwidth is 3.532 kHz. These are baseband values – the equivalent pass band values would be 28.93726 MHz and 7.064 kHz. For the purposes of this driver, fine adjustment is not necessary. Most applications would use the full available bandwidth and, for all five receive signal paths, the default setting is maximum bandwidth. Thirteen bandwidth steps are available, though, in the receiver bandwidth set function. The resistor and capacitor settings for each bandwidth step were given by Motorola in the RFIC documentation [18] [19]. From the desired bandwidth, the function determines the closest bandwidth setting available and sets the nine variables corresponding to the resistor and capacitor settings and sets the seven corresponding SPI registers.

Both transmitter and feedback loop bandwidth are set more simply. Adjustable resistors and capacitors need not be set. Two variables on two registers control the transmitter bandwidth. One variable and one register control the feedback loop bandwidth. For the transmitter, the two variables simply set the two poles of the two baseband filters in the transmit path, as seen in Figure 12, above. For the feedback loop, a single filter controls the bandwidth. The transmitter bandwidth can be set from 6.25 kHz to 14 MHz in eleven steps. The transmitter bandwidth set function takes the desired bandwidth, determines the closest step available, and sets the corresponding variables and registers. For the feedback loop, the bandwidth can be 5 MHz, 10 MHz or 14 MHz. The feedback loop bandwidth set function takes the desired bandwidth, determines the closest step available and sets the corresponding variable and register.

Also in the RFIC object are functions to enable and disable the feedback loop. This function is unique to the RFIC daughterboard and to this driver – it is not available in any of the standard daughterboards. The feedback loop uses the RFIC baseband receive output pins to the high-speed ADCs to output a baseband representation of the signal being transmitted by the RFIC. Essentially, it takes the RF transmit output signal from the transmit output pins, amplifies it and mixes it down to baseband, amplifies and filters it at baseband and outputs it. The user can then look at this baseband representation of the transmitted signal as if it were a received signal. It can be analyzed with any software-defined radio processing tool. Armed with a good representation of exactly what is being transmitted, the user can do DC offset correction or implement a pre-distortion filter. This can significantly improve the performance of the

transmitter. Feedback can be enabled or disabled in real-time, using the functions below, allowing the user to do DC offset correction or pre-distortion at runtime, optimizing its effect. A diagram of the feedback loop can be seen in Figure 19, below.

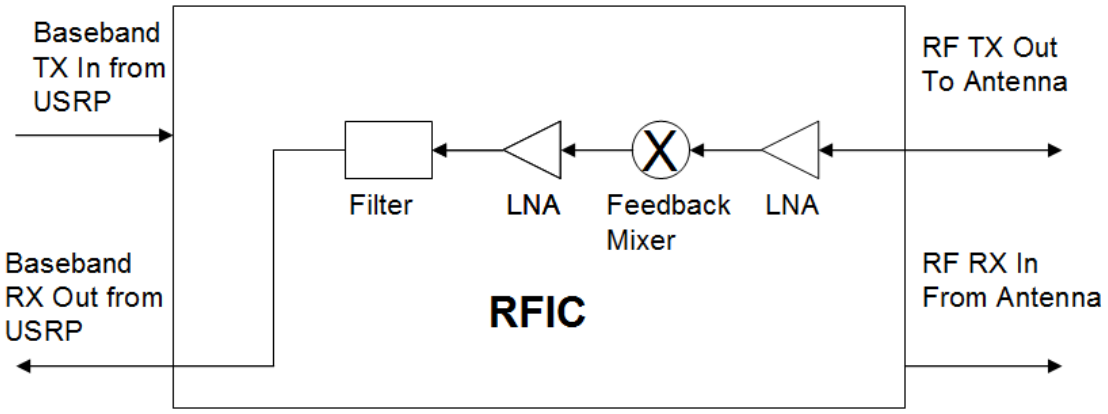

Figure 19: Feedback Loop Diagram

The function that enables the feedback loop starts by disabling the receiver filter output. The pins normally used by the receiver to send analog information to the ADCs must be re-tasked for the feedback loop. Information received by the RFIC is not relevant at this point – the transmitted waveform is. Next, the function enables baseband feedback, with the TX I and Q paths being fed back through the RX I and Q paths, respectively. This allows the user to directly analyze the transmitted waveform without having to swap I and Q or do I-and-Q mixing. Next, baseband feedback calibration is disabled. This mode shorts the baseband feedback amplifier input for the purpose of calibration, and is not desirable when feedback is output to the user. The Cartesian baseband feedback forward path is enabled. In this context, Cartesian refers to the I-and-Q paths. The Cartesian feedback path is enabled. DC offset correction is enabled. The gain of the baseband amplifiers is set. Finally, the zero of the Cartesian feedback forward path is enabled.

Another function is used to disable the feedback loop. This allows a user to return to receiving normally, after using the transmitter feedback to do DC offset correction or pre-distortion. The function enables the RX filter output, disables the baseband feedback, enables baseband feedback calibration, disables the baseband Cartesian feedback forward path, disables the Cartesian feedback path, disables Cartesian feedback, disables DC offset correction, sets the feedback gain to zero, and disables the zero of the Cartesian feedback forward path.

The last function in the RFIC object is the RSSI, or received signal-strength indicator function. This is another function not available with the standard daughterboards. The RFIC has on-channel clip and fade detectors and an off-channel clip detector. The off-channel detector operates similarly to the on-channel clip detector, except that it has no corresponding fade detector. Currently, the off-channel detector is not functional, so the RSSI function only uses the on-channel clip and fade detectors. The signal from the on-channel clip detector is a pulse-width modulated signal – it is zero volts when the amplitude of the received signal is below a certain threshold and 2.5 volts when the received signal is above a certain threshold. The on-channel fade detector returns zero volts when the amplitude of the received signal is above a certain threshold and 2.5 volts when the amplitude of the received signal is below a certain threshold. Figure 20, below, shows typical operation of the clip and fade detectors.

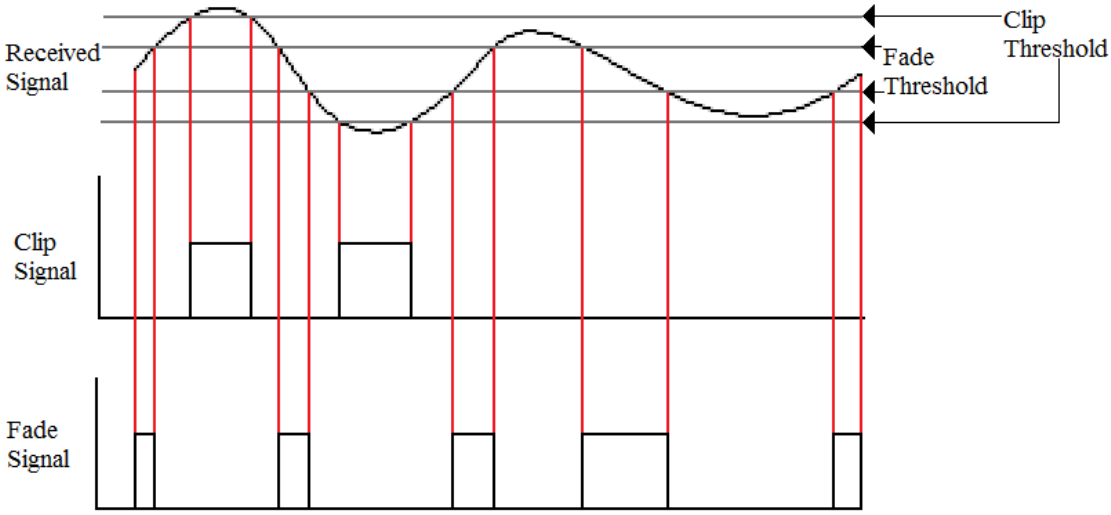

Figure 20: RSSI Graph

The clip and fade signals are output to two test pins. In the current revision of the RFIC daughterboard, these test pins may be connected directly to the two ADCs on the receive side of the USRP. Since the signals are pulse-width modulated, using the signals directly would require constantly sampling them. This information would have to be sent to GNU Radio over the USB 2.0 connection, further taxing the available data rate of the connection. A future revision of the RFIC daughterboard is planned, which will have a low-pass filter between the test pins and the ADCs. This will convert the pulse-width modulated signal to an amplitude-modulated signal. A single sampling at the output of the low-pass filter will give the user a value directly related to how often the received signal is clipping or fading at that time. A large value from the clip pin will indicate that the signal is clipping often – and that the user should reduce the gain. A large value from the fade pin will indicate that the signal is fading often – and that the user should increase the gain. This will be very useful for automatic gain control and to optimally use the full dynamic range of the high-speed ADCs in the receiver.

The RSSI function in the RFIC object first turns off the test-pin multiplexer. The test-pin multiplexer allows the test pins to be used for several purposes, but it must be disabled to use the test pins for clip-and-fade detection. Next, the on-channel clip and fade detectors are turned on. The off-channel clip detector, as mentioned above, does not work and is turned off. The function sets the clip and fade thresholds. Finally, it polls the auxiliary ADCs connected to the test pins. It returns these values to the user.

3.4.2 The Base Class

The RFIC base class is an abstract, base class for all RFIC daughterboards. Transmit and receive subclasses are derived from this base class. It consists of four functions: an initialization function; a deletion function; a function that returns whether the board is quadrature; and a function that returns the frequency range of the board. GNU Radio, when trying to perform a function on a subdevice, tries to run the function in this class first. If the function is not available

in this class, it tries to run the function in the subclass. Functions to set the transmitter and receiver frequency, amplitude, and gain range, among others, are located in the subclasses for the RFIC board, but may be located in the base class for other daughterboard types. For instance, if a transceiver daughterboard uses a single VCO for both transmitting and receiving, and is therefore half-duplex, it may have only a single frequency-set function, which would be placed in the base class of the driver. The complete base class code can be found in Appendix A: The Driver Code, under the heading **`class db_rfic_base(db_base.db_base):`**.

First, and most importantly, there is the initialization function. When GNU Radio initializes an RFIC subdevice, this function is always run. The base class for all USRP daughterboards is initialized first. This is part of GNU Radio, not part of this driver, and sets up various standard functions. Some of these functions are designed to be over-written by the daughterboard drivers. Either the base class or any subclass in the daughterboard driver can over-write the functions. Functions related to automatic transmit/receive switching, setting center frequency, frequency range, setting gain, gain range, quadrature operation and antenna selection are over-written by the RFIC daughterboard driver. These functions will be covered in the Sections 3.4.3 The TX Subclass and 3.4.4 The RX Subclass, below.

Next, the initialization function runs a function to get or make an RFIC object. This function is part of the driver, but exists outside of the RFIC object or any of the classes. If the RFIC daughterboard, which GNU Radio is trying to initialize, has not already been initialized or has been deleted, the function creates a new RFIC object. This initializes the RFIC object, as described in the Section 3.4.1 The RFIC Object, above. It associates the class being initialized (and subclass about to be initialized) with the new RFIC object. If the RFIC daughterboard has been initialized, and has not been deleted, the function associates the class being initialized, and subclass about to be initialized, with the existing RFIC object. It does not need to, and should not, initialize the RFIC object again.

The deletion function doesn't really do anything – the deletion function in the RFIC object, as described above in Section 3.4.1 The RFIC Object, turns off some power-consuming features, as do the deletion functions in the TX and RX subclasses, which will be discussed in the Sections 3.4.3 The TX Subclass and 3.4.4 The RX Subclass, below.

Another function simply returns "True" when GNU Radio asks if the daughterboard is quadrature. Both the TX and RX sections of the RFIC use separate I-and-Q paths, so both the transmitter and receiver benefit from quadrature operation with respect to GNU Radio. On the transmit side, I and Q samples from the USRP are converted to I and Q signals in the DAC. The I and Q signals are filtered, amplified and up-converted to RF by the RFIC. The signals are combined and amplified again before being sent to the antenna port on the daughterboard. Similarly, on the receive side, the RFIC receives an RF signal from the antenna port. The RF signal is mixed down to baseband with a quadrature mixer, resulting in I and Q baseband signals. These signals are amplified and filtered by the RFIC and sent to the USRP, where the ADC turns them into digital samples. This function tells GNU Radio that the board operates this way.

The last function in the RFIC base class returns the frequency range of the daughterboard. It also returns the frequency step size. The minimum frequency returned by this function is 100 MHz. The maximum is 2.5 GHz. While the receiver can receive signals above 2.5 GHz, the transmitter power level is unacceptably low above 2.5 GHz. This function needs to account for that limit. It returns a frequency step size of 1 kHz, which is higher than the step size of the RFIC, but low enough to be useful for nearly any real radio implementation. This function is rarely used by GNU Radio. Most programs simply try to set the daughterboard frequency to a

desired value, without checking whether the value is in the daughterboard's theoretical operating range. This is partly because most daughterboards are able to operate outside their theoretical range. The values returned by this function serve more as guidelines than as rules.

3.4.3 The TX Subclass

The TX subclass is used when GNU Radio is transmitting information using the RFIC daughterboard. It is a subclass of the base class, as described in Section 3.4.2 The Base Class, above. It consists of: an initialization function; a deletion function; an antenna-selection function; a gain range function; a gain setting function; a frequency setting function; a phase-offset setting function; a bandwidth setting function; and a function to invert the RF spectrum. Most of these functions call functions in, or use variables found in, the RFIC object, as described above in Section 3.4.1 The RFIC Object. They control transmission-related functions on the RFIC daughterboard. The complete TX subclass code can be found in Appendix A: The Driver Code, under the heading **`class db_rfic_tx(db_rfic_base):`**.

When GNU Radio initializes a transmitter subdevice, it initializes the TX subclass using an initialization function. If GNU Radio deletes the subdevice, the subclass is deleted but the base class is not. The base class may be in use by a RX subdevice and should only be deleted if the entire daughterboard is being deleted. The initialization function sets up the daughterboard, and specifically the RFIC, to transmit signals. This involves setting many variables and their associated registers on the RFIC along with one of the low-speed auxiliary DACs. The TX subclass initializer first initializes the base class, as described in Section 3.4.2 The Base Class, above. This sets up the RFIC object, as described in Section 3.4.1 The RFIC Object, above, if it has not already been set up. It retrieves an instance of the RFIC object if it has already been set up. The base class also sets up two functions common to the receiver and transmitter subclasses.

Next, the initialization function sets up the RFIC. First, it gets the direct digital frequency synthesizer out of reset mode, so that it can be used. Second, it turns on the forward baseband reference section of the transmitter. This section filters, level-shifts, attenuates and buffers the signals from the high-speed DACs on the USRP. A simplified diagram of this section can be seen in Figure 21, below. The filtered, shifted, buffered signals can be fed to the forward RF section of the transmitter, a simplified diagram of which can be seen below in Figure 22. Third, it disconnects the Cartesian error signal from the baseband correction feedback loop and routes the feedback signal through the baseband correction feedback loop. This allows feedback through the Cartesian loop. Fourth, it enables the baseband correction feedback path. Fifth, it enables the forward RF transmit path. This path includes the mixer and stepped attenuator along with the drivers to output the RF signal, biased by the RF bias reference and the AOC bias, which provides continuous power control. Sixth, the Cartesian feedback path is switched to send feedback data to the error generation block, which allows the feedback loop to operate. Seventh, a zero in the Cartesian feedback loop frequency response is enabled. This helps with second-order stability.

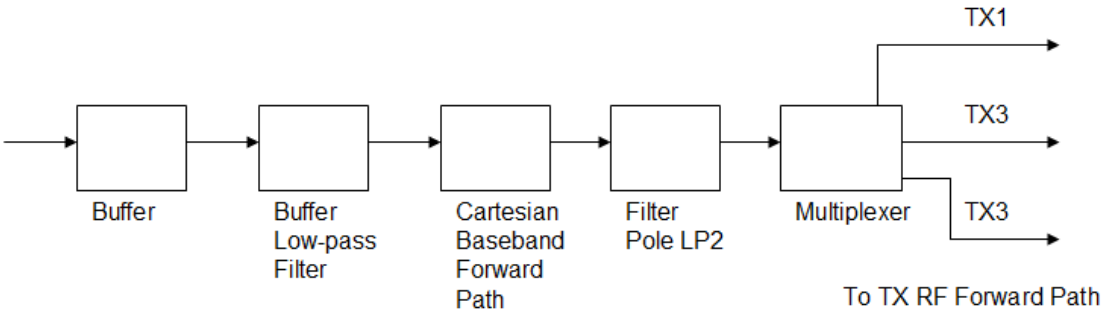

Figure 21: Transmitter Baseband Reference and Filter

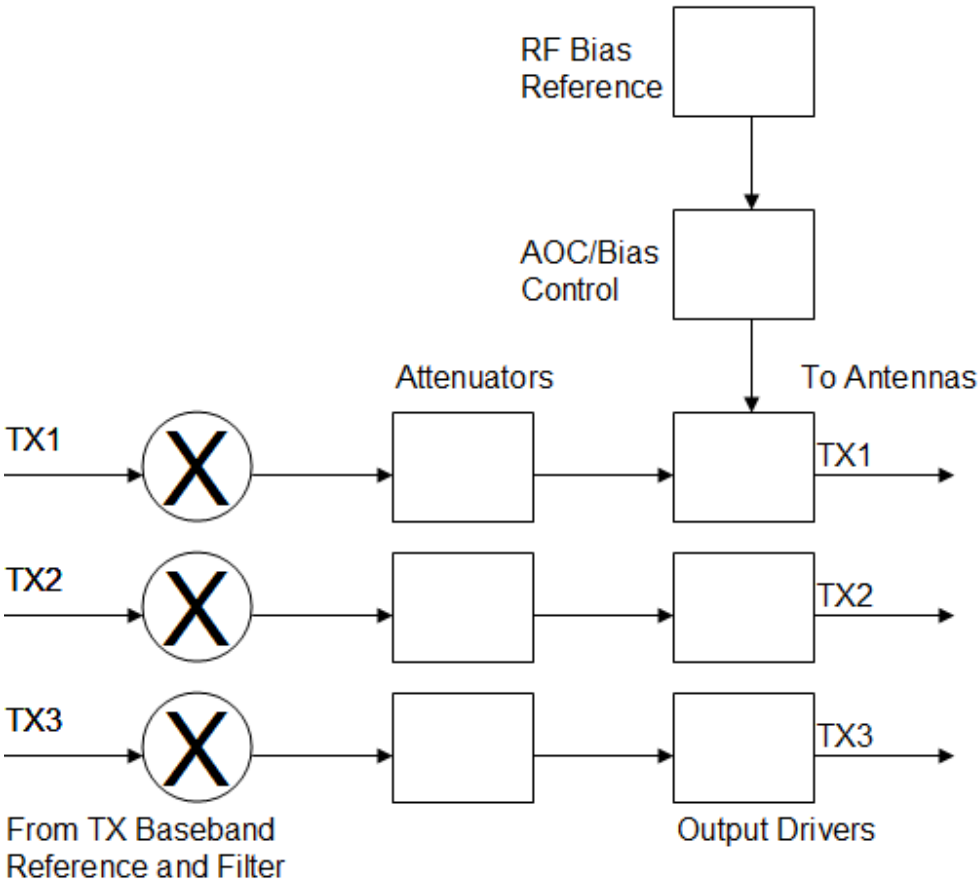

Figure 22: Transmitter RF Forward Path

Eighth, the transmit output path is selected. By default, and when the transmit subdevice is initialized, the transmit path used is TX1. A diagram above, Figure 12, shows the three TX output paths. Ninth, the continuous gain for output channel TX1 is set to the maximum. While all three transmit paths have 45 dB of stepped gain control (in 5 dB steps), transmit path TX1 has an additional 35 dB of continuous gain control. This gain is controlled through a voltage bias on one of the pins on the RFIC chip. To harness this extra gain control, one of the auxiliary DACs,

AUX_DAC_D, is connected to that pin. Maximum gain is achieved at 2.2 volts, so the value sent to the DAC, upon initialization, is 2750, which results in an output of 2.2 volts. I decided not to take advantage of the extra 35 dB of gain control in this driver because it was exclusive to this output. A user who requires the extra gain control can adjust the value of the DAC manually without too much trouble. It would have been confusing to advertise 70 dB of gain control when only 45 dB of control is available when using two of the three output paths.

Tenth, and finally, the initialization function sets up several variables related to the direct digital frequency synthesizer on the RFIC. It uses standard values recommended by Motorola [18] [19]. These values work with any of the frequency multipliers through the entire 200 MHz to 1 GHz frequency range of the synthesizer. They do not need to be adjusted. It enables the clock driver, connecting the clock to the digital frequency synthesizer block. It enables two sets of voltage regulators, which supply power to both analog and quadrature generator functions. It takes the windowing function out of reset mode, allowing the direct digital synthesizer to output a signal. Next, it disables the fine line for tuning the output of the synthesizer, in favor of the coarse taps. It disables the output of calibration signals off-chip. It allows the frequency reference, multiplied to 1 GHz, to be sent to the synthesizer. Finally, it enables control of the delay line via the DLL loop filter, which is the normal operating state.

The deletion function turns off the transmitter output entirely, selecting none of the output paths. It turns off the forward RF transmit path and the forward baseband reference section. It disconnects the Cartesian error and feedback signals from the baseband correction feedback loop. Next, it switches the Cartesian feedback path to not send feedback data to the error generation block, disabling the feedback loop. The zero in the Cartesian feedback path is disabled. The baseband feedback section is turned off. All of the functions and variables related to the frequency synthesizer are turned off. Finally, the digital synthesizer block is put into a reset state. All of these measures save power when the transmitter is not in use.

The next function in the TX subclass chooses an output path. Three output paths are available on the RFIC: TX1, TX2 and TX3. Each of these is connected to an antenna port on the RFIC daughterboard. The paths have different RF characteristics: TX1 has 80 dB of gain control, medium linearity and better low-frequency performance; TX2 has 45 dB of gain control, high linearity and good low-frequency performance; and TX3 has 45 dB of gain control, low linearity and better high-frequency performance. The user can select any of these transmit paths at will, in order to satisfy varying performance requirements.

Another function returns the gain range of the transmitter. It also returns the increment. As described above, the gain range for all three outputs is 45 dB (the lowest gain being 0 dB and the highest gain being 45 dB), while the gain is adjustable in increments of 5 dB. The four functions to set gain, frequency, phase-offset and bandwidth of the transmitter are very similar. Each of them calls the corresponding function in the RFIC object, which are described in the Section 3.4.1 The RFIC Object, above. The gain setting function takes an input in the form of the desired gain, in dB. The frequency setting function takes an input in the form of the desired operating frequency, in Hz. The phase-offset setting function takes an input in the form of the desired phase offset, in degrees. The bandwidth setting function takes an input in the form of the desired bandwidth, in Hz. These functions allow the user to specify basic operating parameters of the transmitter. The last function in the TX subclass tells GNU Radio to invert the RF spectrum. This can easily be done by swapping the I and Q samples or by negating I or Q – by using +I and –Q or –I and +Q.

3.4.4 The RX Subclass

Similar to the TX subclass, described in the Section 3.4.3 The TX Subclass, above, the RX subclass is initialized by GNU Radio when creating a receiver flow graph. It is also a subclass of the base class, described in Section 3.4.2 The Base Class, above. It consists of functions to: initialize the receiver subdevice; delete the receiver subdevice; set the receiver path; return the gain range of the receiver; set the gain of the receiver; set the frequency of the receiver; set the phase offset of the receiver; set the bandwidth of the receiver; enable the feedback loop; disable the feedback loop; return the gain range of the feedback loop; set the gain of the feedback loop; set the frequency of the feedback loop; set the phase offset of the feedback loop; set the bandwidth of the feedback loop; and return received signal-strength indicators. Most of these functions, like the TX subclass functions, call functions in the RFIC object, as described in Section 3.4.1 The RFIC Object, above. They are used to set the various operating parameters of a receiver using the RFIC daughterboard. The complete RX subclass code can be found in Appendix A: The Driver Code, under the heading **`class db_rfic_rx(db_rfic_base):`**.

When GNU Radio creates a receiver flow graph, it automatically initializes the RX subclass of the attached daughterboard. The initialization function first initializes the base class. This also retrieves an existing implementation of the RFIC object, as described in Section 3.4.1 The RFIC Object, above, if one exists. If there is no existing RFIC object, one is created by the base class. This ensures that the board is ready for standard operation. Initializing the RFIC object sets the registers on the RFIC to standard default values, sets up variables, sets up USRP operations and automatic transmit/receive switching, among other things.

Second, the initialization function takes the digital frequency synthesizer out of its reset state. Third, it sets the receive path. By default, this driver sets the receive path to be RX1, which has the lowest noise floor. It sets the bias current for the LNA. In order to allow the mixer to operate, it connects the LO to the receiver mixer. It enables the baseband receiver filters. It enables the baseband filter chopper clock, and then enables the choppers on all five receive-path mixers. The choppers improve the low-frequency response of the baseband signal, output to the user. It sets the chopper divide clock, which sets the LO frequency in the chopper. The chopper operation is described in the Section 2.5 The RFIC, above.

Next, the initialization function enables the output of the receiver filter. Without this, the user will see no signal regardless of any other settings. The initialization function sets the BiQuad Q and the BiQuad and VGA resistor values and the PMA feedback resistor. These are set to default values. It disables compensation control in the BiQuad and VGA filters, which allows higher bandwidths in the filters. A diagram of the receiver, including these filters, can be seen above in Figure 10. Next, it enables the DAC for the DC offset-correction circuitry and the DCOC comparator. The function enables RC tuning for the baseband filters and enables the ramp circuit for RC tuning. It selects the divider ratio for the DCOC and RC tuning clock. Next, it enables DC offset correction. Finally, it sets several variables related to the direct digital synthesizer. These variables and values are the same as those described in the initialization function of the TX subdevice, described in the Section 3.4.3 The TX Subclass, above. The only difference is that the variables and values are set in the RX synthesizer, rather than in the TX synthesizer.

The deletion function disables everything enabled in the initialization function above. First, it turns off all five receive signal paths. Second, it disables the LO connection to the receiver mixer. Third, it disables the receiver baseband filters. Fourth, it disables the clock to

the choppers. Fifth, it disables the output of the receiver baseband filters along with the RSSI indicators. Sixth, it disables the DC offset correction DAC and the DC offset correction comparator. Seventh, it disables the RC tuning circuit and the ramp circuit in the RC tuning circuit. Eighth, it disables the DC offset correction. Ninth, it disables all of the direct digital synthesizer-related blocks. These are described in the description of the initialization function above. Finally, the deletion function puts the direct digital synthesizer into a reset state. Turning off these blocks and devices saves power when the receiver is not in use.

The RX subclass has a function to select an input path, as the TX subclass has a function to select an output path. The RFIC has five input paths: LNA1, LNA2, LNA3, LNA4 and MIX5. Each of the input paths has different characteristics, and may be selected by the user at any time to best meet the current receiver requirements. LNA1 through LNA4 each has a different LNA and mixer, while MIX5 has a unique mixer but no LNA. The mixers in LNA2 and LNA4 use chopping mixers. Chopping is described in the Section 2.5 The RFIC, above, and improves low-frequency response of the mixer. It improves second-order harmonic, flicker noise and the DC offset. The mixers in LNA1, LNA3 and MIX5 are passive mixers – they do not have choppers. MIX5, with no LNA of its own, is designed to operate with an external LNA.

Another function in the RX subclass returns the gain range of the receiver and the step size. The receive path in the RFIC has 38 dB of gain, adjustable in 1 dB increments. Therefore, this function returns 0, 38 and 1 (the minimum gain, maximum gain and increment). This function lets GNU Radio and the user know how much gain is available on the receive side of the RFIC daughterboard. Other daughterboard drivers have similar functions returning the appropriate available gain range.

Several functions in the RX subclass call, and pass variables to, functions in the RFIC object, as described in Section 3.4.1 The RFIC Object, above. These are: set gain; set phase; set bandwidth; enable feedback; disable feedback; set feedback gain; set feedback frequency; set feedback phase; set feedback bandwidth; and get RSSI information. These functions and their operations are described in Section 3.4.1 The RFIC Object. The set gain function sets the receive path gain, from 0 to 38 dB, in increments of 1 dB. The set phase function sets the receive path phase offset, from 0 to 360 degrees. The set bandwidth function sets the bandwidth of the receive path, from 3.5 kHz to 14.4 MHz. These values represent the baseband bandwidth, set in the baseband filters in the receive path.

Feedback is available in the RX subclass. Because feedback data takes the place of received radio data, when the RFIC daughterboard is in feedback mode, the USRP acts as a data source, in GNU Radio terms. This means that the gain, frequency, phase offset, and bandwidth of the feedback loop should be set in a receiver flow graph. Hence, the feedback functions are in the RX subclass. Feedback is not available in any other standard daughterboard – it is a feature unique to the RFIC daughterboard. The enable, disable, set gain, set frequency, set phase offset, and set bandwidth functions simply call and pass variables to the functions in the RFIC object, as described in Section 3.4.1 The RFIC Object, above. The gain range function returns 0, 40, and 5. This corresponds to a minimum gain of 0 dB, a maximum gain of 40 dB, and an increment of 5 dB. The feedback loop in the RFIC has 40 dB of gain range, in increments of 5 dB. This function tells GNU Radio, and the user, what gain values are available for the feedback loop.

The last function in the RX subclass is the RSSI, or received signal-strength indicator, function. This is another function not available with the standard USRP daughterboards. Calling the RSSI function in the RFIC object, as described in Section 3.4.1 The RFIC Object, above, this function returns to the user values proportional to how often the signal is clipping (that is, high

amplitude) and how often the signal is fading (that is, low amplitude). These values are very useful when adjusting receive path gain.

3.4.5 Auto-Instantiation

One last function in the driver is not in the RFIC object, the base class or either the TX or the RX subclasses. It hooks the daughterboard subclasses into GNU Radio's automatic instantiation framework. The classes are added to the daughterboard instantiator, and associated with the unique daughterboard ID assigned to them.

3.5 Tuning and Optimization

Each of the frequency synthesizers had to be optimized separately [27]. The three synthesizers had to be optimized for three of the four multipliers (1x, 2x and 4x) connected to each synthesizer as well. This optimization procedure does not work for the 1/2x multiplier. The optimization was achieved by adjusting values of the variables in various SPI registers. I hooked up a signal generator to the RFIC daughterboard input and used usrp_fft.py [28] to view the received spectrum in order to optimize the receiver. To optimize the transmitter, I hooked up the output of the RFIC daughterboard to a spectrum analyzer, produced a tone with usrp_siggen_rfic.py, and viewed the output spectrum. Usrp_siggen_rfic.py, which may be seen in Section Appendix C: usrp_siggen_rfic.py, is a slightly modified version of usrp_siggen.py, which is a signal generator program. It is modified to force GNU Radio to recognize the RFIC daughterboard. In each case, I followed the instructions in the Motorola document, RFIC4a Evaluation Board Alignment Procedure [27].

Eight variables are adjusted in the alignment procedure: Qg00degDelay, Qg90degDelay, Qg180degDelay, Qg270degDelay, DischargeTap16, ChargeTap16, DischargeTapnn16 and ChargeTapnn16. All three frequency synthesizers (transmitter, receiver and feedback loop) have unique instances of these variables. Furthermore, the optimal settings are different for the 1x, 2x and 4x multipliers. The QgXXdegDelay variables control the quadrature phase offset in the local oscillator. They control exactly where the square-wave signals generated by the frequency synthesizer shift from low voltage to high voltage. The diagram below, Figure 23, shows the quadrature signals from the frequency synthesizer. This alignment, using the QgXXdegDelay variables, effectively controls the exact phase of the I and Q signals relative to one another. The alignment does not work for the 1/2x multiplier. It is important for the 0-degree, 90-degree, 180-degree and 270-degree phases to be aligned properly. Phase mismatch between the I and Q signals from the LO causes higher spurs, lower signal-to-noise ratio and more carrier leakage in the transmitter and lower signal-to-noise ratio and higher noise floor in the receiver and feedback loop.

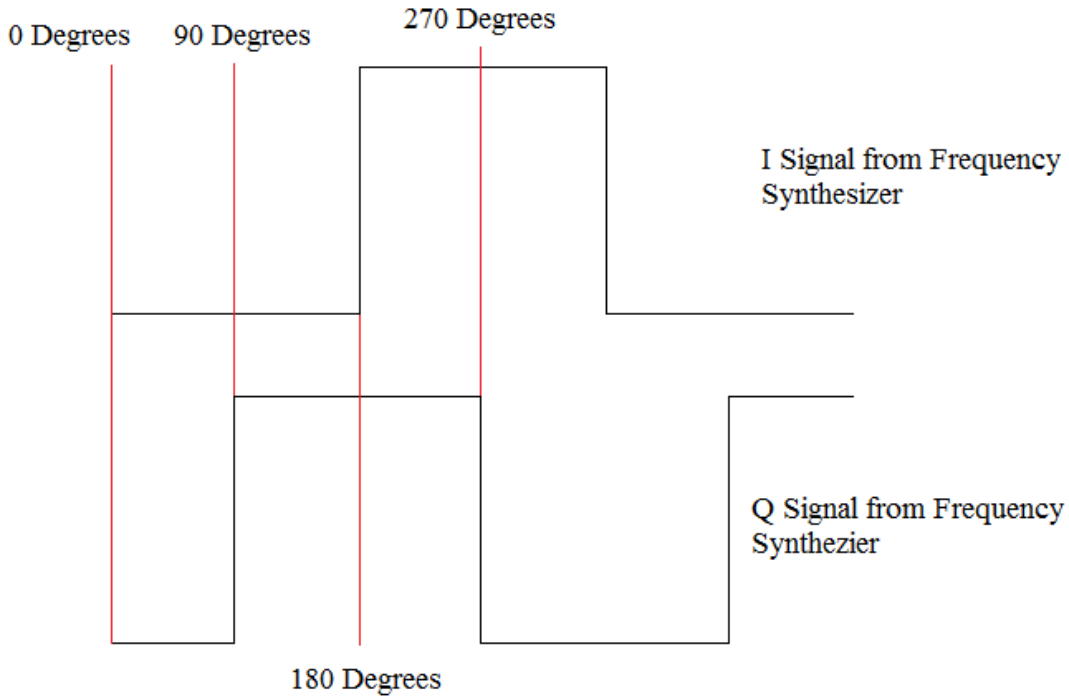

Figure 23: Graph of Phase Delay

The DischargeTap16XX and ChargeTap16XX variables adjust the delay-lock loop (DLL) offset error in the quadrature frequency synthesizers. In each of the frequency synthesizers, the DLL is part of the circuitry that controls the selection of "rising" and "falling" edges, where the signals go from low to high and vice-versa, which therefore determines the synthesized frequency. The signal from the frequency synthesizer, after being sent through one of the frequency multipliers, is the local oscillator signal, and is sent to the corresponding mixer. Any error in the delay-lock loop will cause spurs in the LO output signal. In the receiver, that will raise the noise floor. In the transmitter, that will increase the amplitude of unintentional spurs in the output. In both cases, it will lower the signal-to-noise ratio.

When optimizing the receiver, I attached one of the signal generators to the RX1 input on the RFIC daughterboard. I turned on the RF output of the signal generator and used usrp_fft.py, which is a spectrum analyzer program in GNU Radio, to view the received signal. I adjusted the QgXXdegDelay variables for the receiver frequency synthesizer and monitored the level of the received signal. When the level of the received signal was highest above the noise floor, with respect to all four QgXXdegDelay variables, I recorded the value of each variable. I repeated the same process with the DischargeTap16XX and ChargeTap16XX variables. This process I repeated for with the 1x frequency multiplier engaged, the 2x frequency multiplier engaged, and the 4x frequency multiplier engaged. The values I ended up with are the optimal values for aligning the receiver frequency synthesizer, and I used them when setting the variables in the receiver frequency-selection function, as described in Section 3.4.2 The Base Class, above.

When optimizing the transmitter, I attached the TX1 output on the RFIC daughterboard to a spectrum analyzer. I used usrp_siggen_rfic.py, which is a signal generator program modified slightly from GNU Radio (to force recognition of the RFIC daughterboard), to produce an output from the RFIC daughterboard. Usrp_siggen_rfic.py may be seen in Appendix C: usrp_siggen_rfic.py. While monitoring the spectrum and the output signal on the spectrum analyzer, I adjusted the QgXXdegDelay variables. When all four variables were such that the output power was maximized and the spurs and harmonics minimized, I recorded the value of each variable. I repeated the same process with the DischargeTap16XX and ChargeTap16XX variables. Again, I recorded the values corresponding to the optimal output spectrum with the 1x multiplier engaged, the 2x multiplier engaged and the 4x multiplier engaged. These values I used in the driver when setting the frequency of the transmitter, as described in Section 3.4.2 The Base Class, above.

4. Testing and Results

I thoroughly tested the RF characteristics of both the transmitter and receiver of the RFIC daughterboard as well as the RFX-series daughterboards. Most of the tests used a spectrum analyzer and/or a signal generator or two, in order to analyze the performance of the transmitter and receiver. I also compared the results from the RFIC daughterboard to the results from the RFX-series daughterboards. These direct comparisons allow quantitative determinations of the advantages and disadvantages of the RFIC board and those of the RFX-series boards. The testing procedures and complete, tabulated results can be seen below in Appendix B: RF Testing Procedure and Complete Results

I performed most of the tests at 400 MHz, 900 MHz, 1800 MHz and 2400 GHz. These values were chosen for the sake of convenience: 400 MHz falls in the frequency range of the RFX400 as well as the frequency range of the RFIC while using the x1/2 frequency multiplier with the direct digital synthesizer; 900 MHz falls in the frequency range of the RFX900 as well as the frequency range of the RFIC while using the x1 frequency multiplier with the DDS; 1800 MHz falls in the frequency range of the RFX1800 and the frequency range of the RFIC while using the x2 frequency multiplier with the DDS; and 2400 MHz falls in the frequency range of the RFX2400 and the frequency range of the RFIC while using the x4 frequency multiplier. Some tests were only done at lower frequencies due to the limitations of available testing equipment. One was done at a higher frequency range, using special equipment. Most of the tests were done with all three working RX input paths and both working TX output paths. RX4 is disabled in the version of the RFIC on which I did the testing. RX2 was not working, either. I do not know why. The TX2 output path was not working either. Performing each test on each available RF signal path shows the difference between the performances of the paths. Allowing a user to select any of the available signal paths allows a much wider range of operation, under a much wider range of circumstances and with a much wider range of requirements.

4.1 The Noise Floor

The noise floor is an important measurement of the quality of any receiver. Also known as “minimum detectable signal,” the noise floor is the lowest amplitude signal useable by the receiver. A receiver with a lower noise floor allows the corresponding transmitter to operate at a lower power, allows the corresponding transmitter to operate from farther away, or allows the receiver to detect a fainter signal. A lower noise floor is always desirable.

In Appendix B: RF Testing Procedure and Complete Results, Section Test 1: Noise Floor, is the test procedure I used to measure the noise floor as well as tables containing the complete test results. I used usrp_fft.py [28], a program that comes with GNU Radio by default, along with a signal generator to measure noise floor. Usrp_fft.py sets up a USRP and attached daughterboard to the user’s specifications: center frequency, gain and decimation rate are the most important specs. The program receives baseband samples from the USRP, runs an FFT on the samples, and graphs the result in real-time. It is essentially a spectrum analyzer program and allows the user to see any received signal, centered about a specified center frequency, with a bandwidth determined by the decimation rate, and using a specified gain, in dB, in the daughterboard. An example of an FFT graph created by usrp_fft.py can be seen in the figure below, Figure 24. It is also easy to see the noise floor of the receiver. In order to see the signal of the lowest possible amplitude, I set the receiver gain to the maximum value for each daughterboard I tested. Further, I used the maximum decimation rate (causing the real-time

spectrum graph to show the smallest possible frequency range) in order to see the spectrum with the highest possible resolution. Connecting the signal generator to the daughterboard being tested, I turned on the RF output and saw the signal in the usrp_fft.py plot. A diagram of this setup can be seen in the figure below, Figure 25. I reduced the amplitude of the signal from the signal generator until it disappeared into the noise floor. The amplitude, in dBm, of the signal where it disappeared is the noise floor value.

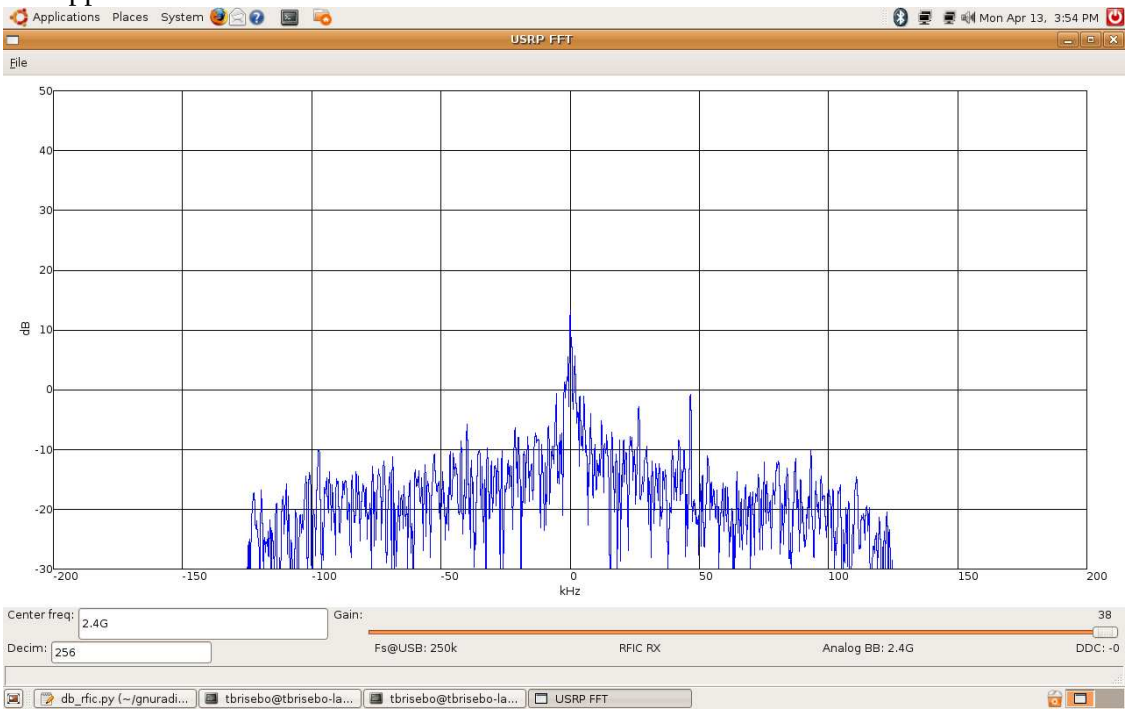

Figure 24: Usrp_fft.py Output Window

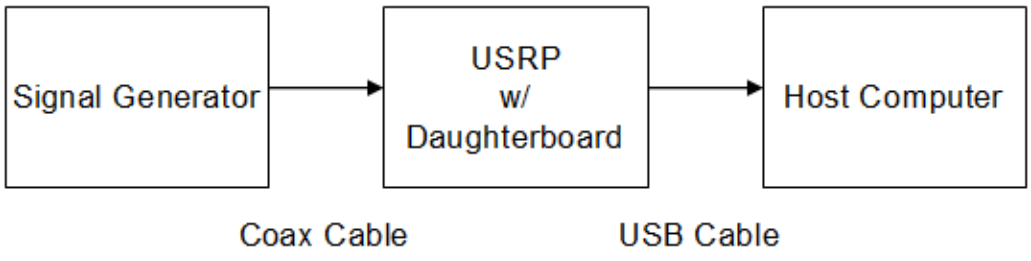

Figure 25: Noise Floor Test Setup

I ran this test at 400 MHz, 900 MHz, 1800 MHz and 2400 MHz on the RFIC daughterboard using each of the three working receive paths: RX1, RX3 and MIX5. RX1 had the lowest noise floor by a significant margin. At lower frequencies, the noise floor was better.

At 400 and 900 MHz, the noise floor was -132 dBm. At 1800 MHz, it was -130 dBm and at 2400 MHz it was -116 dBm. I repeated the tests at 400 MHz on the RFX400 daughterboard, at 900 MHz on the RFX900, at 1800 MHz on the RFX1800 and at 2400 MHz on the RFX2400: the RFX400 at 400 MHz recorded a noise floor of -135 dBm; the RFX900 at 900 MHz recorded a noise floor of -126 dBm; the RFX1800 at 1800 MHz recorded a noise floor of -116 dBm; and the RFX2400 at 2400 MHz recorded a noise floor of -105 dBm. In the noise floor test, the RX1 input of the RFIC is comparable to – or better than – the RFX-series daughterboards. The other inputs have significantly lower noise floors. These results are in Table 3: Noise Floor Test Results, below.

I ran the test again on the RFIC at higher frequencies. No RFX-series daughterboard is able to cover the 3 GHz to 4 GHz frequency range, but the RFIC can. I had to use a different signal generator – the one I used for the other tests could not reach 4 GHz. I only ran this test on the RX1 input, because it has the lowest noise floor. At 3 GHz, the RFIC daughterboard had a noise floor of -109 dBm. At 3.5 GHz, the noise floor was -112 dBm. At 4 GHz, the noise floor was -101 dBm. These low noise floor values show that the RFIC daughterboard is definitely a viable receiver from 3 to 4 GHz, but with a somewhat lower noise floor. The results are in Table 3: Noise Floor Test Results, below.

Table 3: Noise Floor Test Results

	Noise Floor (dBm)	
Frequency (MHz)	RFX-Series	RFIC (Input RX1)
400	-135	-132
900	-126	-132
1800	-116	-130
2400	-105	-116
3000	N/A	-109
3500	N/A	-112
4000	N/A	-101

4.2 The IIP3

The second test I ran was a third-order intercept test. This test indicates the linearity of a receiver. In particular, it indicates the effects of third-order harmonics on the received signal. Third-order harmonics are particularly insidious in radio receivers – they occur when two signals mix with one another, producing a third, spurious signal – because the spurious signal frequently falls within in the receiver pass band and can therefore interfere with the intended received signal. Let's say that there are two received signals mixing together to produce a third-order harmonic at frequencies f_1 and f_2. The interfering signals would be at frequencies $(2 * f_1 – f_2)$ and $(2 * f_2 – f_1)$. If f_1 and f_2 are close together, and within the received bandwidth, then the spurious signals may also be in the received bandwidth. These spurious signals increase in amplitude three times as fast as the two original signals – meaning that they may be strong interferers. The IIP3, or input-referenced third-order intercept point, is the amplitude at which two equal-amplitude signals in the receiver will produce a spurious third-order harmonic signal that, in the receiver, appears to have amplitude equal to that of the original signals. A higher IIP3 point is always desirable – it indicates that the third-order harmonics are weaker and will cause less interference.

In Appendix B: RF Testing Procedure and Complete Results, under Test 2: IIP3, is the test procedure I used to measure the IIP3. Again, I used usrp_fft.py [28] as a spectrum analyzer. Usrp_fft.py allows the user to view a graph of the spectrum in real-time. This time, I used 0 dB of gain in the receivers, to minimize the non-linearities. However, I used the maximum decimation again, in order to see the spectrum with the highest possible resolution. I used two signal generators to create the tones, or equal-amplitude input signals. I used a simple T-connector to combine the signals from the two signal generators (a diagram of this setup can be seen in the figure below, Figure 26), which adds significant loss to both signals. For that reason, I used a spectrum analyzer to verify the actual amplitude of each input signal. Another diagram, showing the setup when testing the actual amplitude of the signals into the daughterboards, can be seen in the figure below, Figure 27. This test was performed with the two tones 20 kHz apart. The third-order harmonic signals would appear 20 kHz above the higher tone and 20 kHz below the lower tone in the usrp_fft.py plot. For instance, if I were testing at 400 MHz, I set one signal generator to 400.1 MHz and the other to 400.12 MHz. The third-order product would show up at 400.08 MHz and at 400.14 MHz. I increased the amplitudes of the tones until the third harmonic was clearly visible. I made sure the signal I was seeing was, in fact, the third harmonic. Then, I checked the amplitude of the harmonic signal. Based on the input amplitudes and the amplitude of the harmonic, I was able to calculate the IIP3.

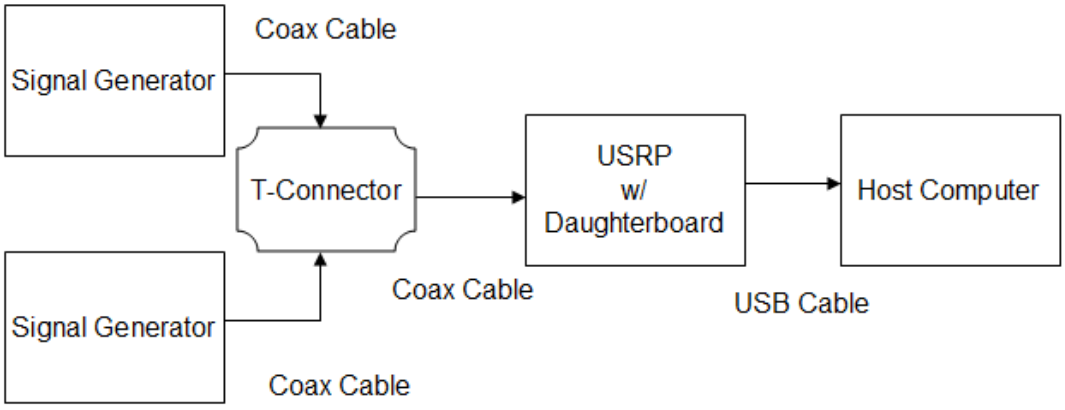

Figure 26: IIP3 Test Setup

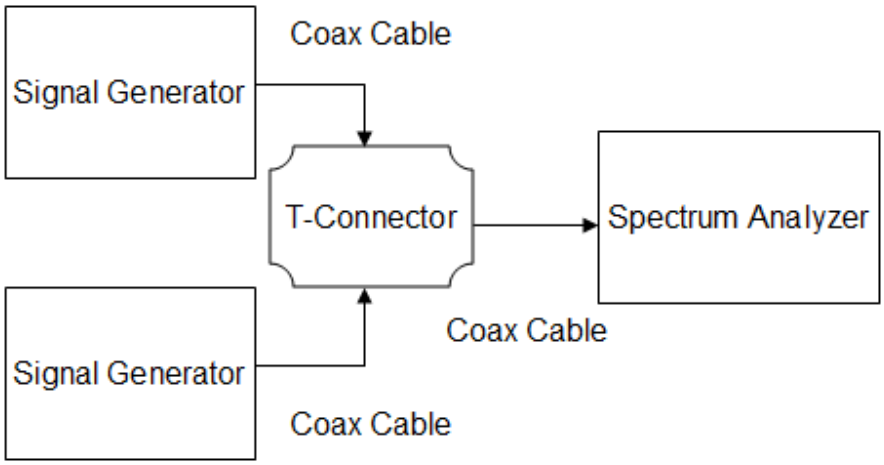

Figure 27: IIP3 Test Setup to Check Amplitude

I used this test on the RX1, RX3 and MIX5 input paths of the RFIC as well as on the RFX400, RFX900, RFX1800 and RFX2400. I ran the tests at 400 MHz, 900 MHz, 1800 MHz and 2400 MHz. The RX1 input on the RFIC daughterboard had the highest IIP3 values. At 400 MHz, the IIP3 was -2.4 dBm; at 900 MHz, it was -4.8 dBm; at 1800 MHz, it was -2.7 dBm; and at 2400, it was 1.3 dBm. With the RFX-series boards, the results were similar: at 400 MHz, the IIP3 of the RFX400 was 0.8 dBm; at 900 MHz, the IIP3 of the RFX900 was 0.5 dBm; at 1800 MHz, the IIP3 of the RFX1800 was -4.6 dBm; and at 2400 MHz, the IIP3 of the RFX2400 was 1.0 dBm. The RFIC daughterboard is comparable to the RFX-series daughterboards in third-order harmonic performance. The results can be seen in Table 4: IIP3 Test Results, below.

Table 4: IIP3 Test Results

	IIP3 (dBm)	
Frequency (MHz)	RFX-Series	RFIC (Input RX1)
400	0.8	-2.4
900	0.5	-4.8
1800	-4.6	-2.7
2400	1.0	1.3

4.3 The IIP2

The IIP2, or input-referenced second-order intercept point, is similar to the IIP3. It displays the linearity of a receiver with respect to second-order harmonics. The second-order harmonics occur either when two signals mix with one another or when one signal mixes with itself in the receiver. Either way, a spurious signal is produced in the receiver. Assuming the two signals are at frequencies f_1 and f_2, the spurious signal will be at frequency $(f_1 + f_2)$ or at frequency $(f_1 – f_2)$ (the frequencies f_1 and f_2 may be the same, if a signal is mixing with itself). These second-order harmonic signals can cause problems in two ways: if the frequencies of the two signals add, then the two signals may be out of the received band, but the harmonic may be within the received band; if the frequencies of the two signals subtract, then the two signals may be in-band and the harmonic may be in the IF or at baseband. Also, the second-order harmonic

signals rise in amplitude twice as fast as the signals that produce them. In particular, if a signal is mixing with itself, its own frequency may be subtracted from its own frequency, resulting in a spurious signal at DC. This is a significant problem for direct-conversion receivers, such as the Motorola RFIC. The IIP2 is the point at which the spurious, second-order harmonic signal appears to be equal in amplitude to the received signal or signals that produce it. A higher IIP2 point indicates that the second-order harmonics are lower in amplitude, and less likely to cause interference, and is always desirable.

I used a two-tone test to determine the IIP2. The procedure and full results can be seen in Appendix B: RF Testing Procedure and Complete Results, Test 3: IIP2. Again, I used two signal generators connected together and to the receiver with a T-connector. The test was set up identically to that in the IIP3 test, as shown in the figure above, Figure 26. This produces losses in the signals from the signal generators, so I used a spectrum analyzer to verify the amplitudes of the signals. This is identical to the amplitude verification setup used in the IIP3 test, as shown in the figure above, Figure 27. I used usrp_fft.py [28] to view the spectrum at the frequency in question and to find the second-order harmonic in the received signal. I set the signal generator tones 1 MHz apart, and to roughly half of the frequency in question – e.g. if I were testing at 400 MHz, I set one signal generator to 199.55 MHz and the other to 200.55 MHz. The two signals mix together to produce a single second-order harmonic at 400.1 MHz. I increased the amplitudes of the signal generators until the second-order harmonic was clearly visible on the usrp_fft.py plot. Then, I verified that the signal I was seeing was, in fact, the second-order harmonic. I recorded the actual received amplitudes of the two tones and the apparent amplitude of the second-order harmonic signal. From this, I calculated the IIP2 values.

Again, I ran this test on the RX1, RX3 and MIX5 input paths of the RFIC and on the RFX400, RFX900, RFX1800 and RFX2400. In this case, the RX3 receive path had by far the highest IIP2 values of the three RFIC daughterboard input paths. This shows that the various input paths can, and should, be used to meet different requirements. At 400 MHz, 900 MHz, 1800 MHz and 2400 MHz, respectively, the RX3 receive path had IIP2 values of 60.9 dBm, 47.2 dBm, 45.6 dBm, and 29.4 dBm. These values were much higher than those of the other two input paths. The RFX-series daughterboards performed as follows: at 400 MHz, the IIP2 of the RFX400 was 8.6 dBm; at 900 MHz, the IIP2 of the RFX900 was 57.8 dBm; at 1800 MHz, the IIP2 of the RFX1800 was 16.8 dBm; and at 2400 MHz, the IIP2 of the RFX2400 was 62.0 dBm. At 400 MHz and 1800 MHz, the RFIC daughterboards had much higher IIP2 values. At 900 MHz and 2400 MHz, the RFX boards had much higher IIP2 values. This is because the RFX900 and RFX2400 have narrow-band RF filters. The RFX900 has a filter around 902-928 MHz and the RFX2400 has a filter around 2400-2483 MHz. These are unlicensed bands. Ettus Research puts filters on the RFX900 and RFX2400 to prevent them from causing harmful interference outside of those unlicensed bands. The filters also effectively block the half-frequency tones in the two-tone test I ran. The RFX400, RFX1800 have only low-pass filters, so the tones were not blocked. The RFIC daughterboards I used had no filters installed whatsoever. Standard sized filters can be installed on the RFIC daughterboard, though, so in IIP2 performance, the RFIC board can achieve comparable, or better, performance compared to the RFX-series boards. The results can be seen in Table 5: IIP2 Test Results, below.

Table 5: IIP2 Test Results

	IIP2 (dBm)	
Frequency (MHz)	RFX-Series	RFIC (Input RX3)
400	8.6	60.9
900	57.8	47.2
1800	16.8	45.6
2400	62.0	29.4

4.4 Transmitter Power

Maximum transmit power is an important measure of the quality of a transmitter. A more powerful transmitter can transmit signals over longer distances or to receivers with higher noise floors. Except in portable devices with limited battery life, a higher maximum transmit power is always desirable.

The full test procedure I used and results I got can be seen in Appendix B: RF Testing Procedure and Complete Results, Test 4: Transmitter Output Power. I measured transmit power using usrp_siggen.py [29] and usrp_siggen_rfic.py, which I wrote for testing purposes. They are identical except that usrp_siggen_rfic.py forces GNU Radio to see an RFIC daughterboard on both side A and B of the attached USRP. This was necessary because the prototype daughterboard I tested did not have an EEPROM chip on the transmit side. Usrp_siggen_rfic.py may be seen in Appendix C: usrp_siggen_rfic.py As discussed in Section 2.3 The Daughterboards, above, GNU Radio uses separate EEPROM chips to recognize transmit and receive subdevices. Because the board I was using had no EEPROM chip on the transmit side, GNU Radio was unable to automatically recognize it as an RFIC daughterboard and I had to force GNU Radio to use the RFIC driver. Usrp_siggen.py is a signal generator program. It produces a waveform, chosen from a list, with a specified frequency and amplitude and mixed up to a specified RF frequency. By default, for instance, it produces a complex sinusoid with a frequency of 100 kHz. If this signal is mixed up to 400 MHz, the result will be a complex sinusoid at 400.1 MHz. Also by default, it has amplitude of 16000, which is near the maximum amplitude. The maximum amplitude corresponds to the highest level signal that the DACs can produce, and therefore the highest amplitude signal the USRP can send to a daughterboard. I used the default amplitude of 16000 rather than the absolute maximum of 16384 because some transmitter daughterboards can be damaged by the maximum signal from the USRP. The transmitters on each daughterboard were set to maximum gain. I hooked the transmitter up to a spectrum analyzer to determine the amplitude of the transmitted signal. A diagram of the setup can be seen in the figure below, Figure 28.

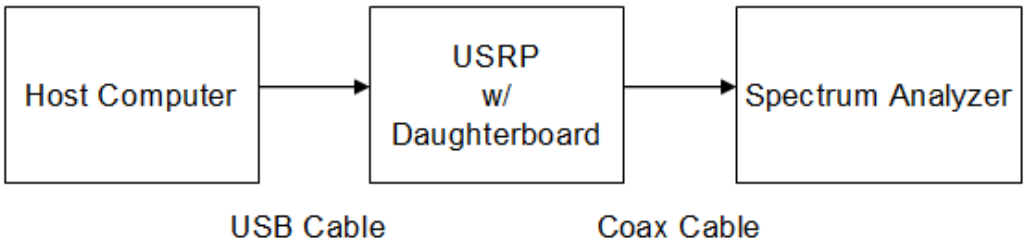

Figure 28: Transmit Test Power Setup

I tested the two working output paths of the RFIC daughterboard, TX1 and TX2, along with the RFX400, RFX900, RFX1800 and RFX2400. On the RFIC daughterboard, TX2 had slightly higher maximum output power. At 400 MHz, the RFIC put out a maximum of 7.9 dBm while the RFX400 put out 22.6 dBm; at 900 MHz, the RFIC put out 3.2 dBm while the RFX900 put out 22.0 dBm; at 1800 MHz, the RFIC put out -3.0 dBm while the RFX1800 put out 20.8 dBm; and at 2400 MHz, the RFIC put out -15.0 dBm while the RFX2400 put out 12.3 dBm. Clearly, in every case, the RFX-series boards could put out significantly more power than the RFIC daughterboard. There is an external power amplifier on the RFIC daughterboard, which I did not use in this test. It should be able to put out close to 20 dBm from 100 MHz to 2.4 GHz. Using that power amp, the power gap between the RFIC and the RFX-series boards should be narrowed or erased. The test results can be seen in Table 6: Transmitter Power Test Results, below.

Table 6: Transmitter Power Test Results

	Transmit Power (dBm)	
Frequency (MHz)	RFX-Series	RFIC (Output TX2)
400	22.6	7.9
900	22.0	3.2
1800	20.8	-3.0
2400	12.3	-15.0

4.5. Local Oscillator Suppression

Local oscillator (LO) suppression is another important measure of the quality of a transmitter. Every mixer leaks some energy from the local oscillator into the transmitted RF signal. The LO may interfere with other radio devices, if it is far away from the transmitted signal, or it may interfere with the transmitted signal itself. The latter is especially true with direct-conversion transmitters, like the RFIC, because the LO signal is at the same frequency as the intended transmitted signal. LO suppression is the difference, in dB, between the amplitude of the intended transmitted signal and the amplitude of the LO signal. Higher LO suppression is always desirable.

The full test procedure I used can be seen in Appendix B: RF Testing Procedure and Complete Results, Test 5: Transmitter LO Suppression. Again, I used usrp_siggen.py [29] with the RFX-series daughterboards and usrp_siggen_rfic.py with the RFIC daughterboard. The only difference between the two programs is that the latter forces GNU Radio to use the RFIC daughterboard driver, as GNU Radio was unable to automatically recognize the prototype board I was using. Usrp_siggen_rfic.py may be seen in Appendix C: usrp_siggen_rfic.py. Usrp_siggen.py and usrp_siggen_rfic.py produce specified types of signals of a specified power at a specified frequency. Again, I had the programs create a complex sinusoid of power 16000, close to the maximum transmit power. The transmitters on the daughterboards were set to maximum gain. This time, I set the frequency of the complex sinusoid to 200 kHz. This would separate the transmitted sinusoid signal from the RFIC LO sufficiently to measure the amplitude of each individually. For instance, when testing at 400 MHz, the RFIC LO signal would be at 400 MHz and the complex sinusoid would be at 400.2 MHz. The RFX-series boards use a low-IF, several megahertz away from the transmitted signal. For instance, when testing at 400 MHz, the RFX400 LO would be at 404 MHz and the complex sinusoid would be at 400.2 MHz. This means that the LO in the RFX-series daughterboards causes less interference with the transmitted signal. I hooked up the transmitter being tested to a spectrum analyzer. This setup is identical to that in the transmit power test, as seen in the figure above, Figure 28. With the spectrum analyzer, I could measure the amplitude of the complex sinusoid and the amplitude of the LO, both in dB. The difference between the two is the LO suppression. A positive-value LO suppression indicates that the amplitude of the complex sinusoid is higher than the amplitude of the LO, which should always be the case.

I tested the RFIC output paths TX1 and TX2 as well as the RFX-series daughterboards at 400 MHz, 900 MHz, 1800 MHz and 2400 MHz. The TX1 port on the RFIC had higher LO suppression than the TX2 port. The RFX-series daughterboards had higher LO suppression than the TX1 port of the RFIC: at 400 MHz, the RFX400 had 41.8 dB of LO suppression and the RFIC had 34.1 dB; at 900 MHz, the RFX900 had 50.3 dB of LO suppression and the RFIC had 27.0 dB; at 1800 MHz, the RFX1800 had 43.2 dB of LO suppression and the RFIC had 27.9; at 2400 MHz, the RFX2400 had 36.1 dB of LO suppression and the RFIC had 24.9. In every instance, the RFX-series daughterboards had better LO suppression than the RFIC daughterboard. Furthermore, because the RFX-series LO was further removed from the transmitted signal, the LO would cause less interference with the transmitted signal. In LO suppression, the RFX-series daughterboards are significantly better than the RFIC daughterboard. The results can be seen in Table 7: LO Suppression Test Results, below.

Table 7: LO Suppression Test Results

	LO Suppression (dBc)	
Frequency (MHz)	RFX-Series	RFIC (Output TX1)
400	41.8	34.1
900	50.3	27.0
1800	43.2	27.9
2400	36.1	24.9

4.6. 2nd-Harmonic Suppression

2nd-harmonic suppression also measures the quality of a transmitter. Rather than measuring leakage, though, it measures linearity. Second-order harmonics can stem from the transmitted signal mixing with itself, or the transmitted signal mixing with the local oscillator on the other side (e.g. high-side versus low-side – if the IF signal is at 4 MHz and the RF transmitted signal is at 400 MHz, the LO may be at 404 MHz, but there would also be an unintended harmonic signal at 408 MHz). These second-order harmonic signals can be out-of-band – and possibly cause interference with other radio users – or they can be in-band and cause interference with the intended transmitted signal. 2nd-harmonic suppression measures the difference, in dB, between the amplitude of the intended transmitted signal and the amplitude of the second harmonic signal. Higher 2nd-harmonic suppression is always desirable.

In Appendix B: RF Testing Procedure and Complete Results, under Test 6: Transmitter 2nd-Order Harmonic Suppression is the full testing procedure I used as well as tables containing the full results. I used usrp_siggen.py [29] with the RFX-series boards and usrp_siggen_rfic.py with the RFIC board. Usrp_siggen_rfic.py may be seen in Appendix C: usrp_siggen_rfic.py. I used those programs to produce a high-amplitude complex sinusoid of frequency 200 kHz, then mix that up to RF, for example to 400 MHz, resulting in a signal at 400.2 MHz. I measured the amplitude of the original signal, with a spectrum analyzer, in a setup identical to that in the figure above, Figure 28, then found the second harmonic around twice the RF frequency and measured the amplitude of that signal. The difference between the two amplitudes, in dB, is the 2nd-harmonic suppression. I measured this 2nd harmonic signal because it was easy to distinguish from other harmonics or distortions. Positive 2nd-harmonic suppression indicates that the intended signal has higher amplitude than the 2nd-order harmonic. A higher 2nd-harmonic suppression value is always desirable.

Due to equipment limitations, I only tested the daughterboards at 400 MHz and 900 MHz. The second harmonics were, therefore, around 800 and 1800 MHz, respectively. While I tested the RFIC with both the TX1 and TX2 output paths, the TX1 output path demonstrated higher 2nd-harmonic suppression. I also tested the RFX400 and RFX900. At 400 MHz, the TX1 path of the RFIC demonstrated 22.8 dB of 2nd-harmonic suppression and the RFX400 demonstrated 34.7 dB of suppression. At 900 MHz, the TX1 path of the RFIC demonstrated 23.2 dB of 2nd-harmonic suppression and the RFX900 demonstrated 38.8 dB of suppression. This is another area where the RFX-series daughterboards are significantly better than the RFIC daughterboard. The results can be seen in Table 8: 2nd-Harmonic Suppression Test Results, below.

Table 8: 2nd-Harmonic Suppression Test Results

	2nd-Harmonic Suppression (dBc)	
Frequency (MHz)	RFX-Series	RFIC (Output TX1)
400	34.7	22.8
900	38.8	23.2

4.7. 3rd-Harmonic Suppression

Like 2nd-harmonic suppression, 3rd-harmonic suppression measures the quality of a transmitter with respect to linearity. Also like the 2nd-order harmonic, the 3rd-order harmonic

may have several sources. Any frequency at a combination of the local oscillator frequency, intended RF signal frequency and original baseband or IF signal frequency, with three elements, will have a 3rd-order harmonic. For example, if the LO is at f_{LO}, the IF signal is at f_{IF} and the RF signal is at f_{RF}, there may be 3rd-order harmonic signals at frequencies: $(2 * f_{LO}) - f_{RF}$; $(2 * f_{RF}) - f_{LO}$; $3 * f_{RF}$; or many more. Some of these harmonics may be within the desired transmitted spectrum, and may cause interference with the intended transmitted signal. Others may be out of the desired transmitted spectrum and may cause interference with other radio users. The difference in amplitude, in dB, between the desired transmitted signal and the 3rd-order harmonic signal is the 3rd-harmonic suppression. Positive 3rd-harmonic suppression indicates that the intended signal has higher amplitude than the 3rd-order harmonic. A higher 3rd-order harmonic suppression value is always desirable.

The complete testing procedure and results can be seen in Appendix B: RF Testing Procedure and Complete Results, Test 7: Transmitter 3rd-Order Harmonic Suppression. I measured the amplitudes of the intended signals as well as the amplitudes of the harmonics with a spectrum analyzer. This setup is identical to that in the transmit power test, as seen in the figure above, Figure 28. I produced the signals with usrp_siggen.py [29] when testing the RFX-series daughterboards and with usrp_siggen_rfic.py when testing the RFIC daughterboard. Usrp_siggen_rfic.py may be seen in Appendix C: usrp_siggen_rfic.py. These programs allowed me to create a complex sinusoid, with frequency of 200 kHz, and mix it up to RF in the daughterboard. The amplitude of the sinusoid was close to the maximum and the transmitter gain in the daughterboard was set to maximum.

Again, due to limitations in the available equipment, I only tested the daughterboards at 400 MHz and 900 MHz. I looked for the third harmonics around 1200 and 2700 MHz, respectively. I looked for these harmonics in particular because they were easy to distinguish from other harmonics or other possible sources of interference. Again, I tested the TX1 and TX2 transmit paths on the RFIC. This time, the TX2 path showed higher 3rd-order harmonic suppression. I tested these against the RFX400 and RFX900 daughterboards. At 400 MHz, the RFX400 demonstrated a 3rd-harmonic suppression of 48.4 dB and the RFIC output path TX2 demonstrated 19.0 dB. At 900 MHz, the RFX900 demonstrated 41.7 dB of 3rd-harmonic suppression and the TX2 path of the RFIC demonstrated 26.7 dB. Again, the RFX-series daughterboards are significantly better than the RFIC daughterboard in harmonic suppression. The results can be seen in Table 9: 3rd-Harmonic Suppression, below.

Table 9: 3rd-Harmonic Suppression

	3rd-Harmonic Suppression (dBc)	
Frequency (MHz)	RFX-Series	RFIC (Output TX2)
400	48.4	19.0
900	41.7	26.7

5. Further Work and Conclusions

5.1 Further Work

A great deal is left to do. The performance needs to be improved, either through software or hardware tweaks. Several tests still need to be run. The driver itself needs to be translated. This process will continue until the daughterboard hardware has been finalized, and possibly further if GNU Radio changes the daughterboard driver format.

The LO suppression, 2nd-harmonic suppression, and 3rd-harmonic suppression in the transmitter of the RFIC daughterboard are not high enough. The RFX-series daughterboards clearly out-perform the RFIC daughterboard in all of these areas. They will, therefore, have better transmitter performance until these areas are addressed. The transmit power deficit, on the other hand, can be solved with the addition of power amplifiers on the daughterboard itself. Since power amplifiers are already implemented in the current version of the daughterboard, it will not be necessary to make any significant changes. The amplifiers must simply be used. I chose not to employ the power amplifiers on the prototype daughterboards when running my tests because I intended to evaluate the RFIC and my software driver, not an off-the-shelf power amplifier. The LO- and harmonic-suppression problems, however, remain. The results I achieved with my driver did not meet the specifications from Motorola [19]. In those specifications, the LO- and sideband-suppression figures were at least 35 dB. The highest I achieved was 34 dB. The lowest was 14 dB. Clearly, better performance may be attained. I need to run additional optimization, as described in Section 3.5 Tuning and Optimization. I also need to optimize the DC offset correction, which will improve both LO suppression in the transmitter and low-frequency noise in the receiver. Furthermore, subsequent revisions of the daughterboard hardware may provide better performance. In particular, the next revisions will include higher-performance transformers, which should improve high-frequency performance. Additionally, new revisions of the RFIC itself may offer improved performance. The chip is not a regular production model yet, and may see significant improvements in its life cycle.

More testing is required as well. For instance, I devised and ran a test of frequency-switching speed. I was not happy with either the testing procedure or the results, so I chose not to include it in this thesis. The RFIC daughterboard consistently took about 5 to 6 ms to switch from one frequency to another, either in transmit or receive mode. It also took 15 ms to set the initial frequency. This is far too slow to do any kind of frequency-hopping. The RFX-series daughterboards took about 15 ms to set the initial frequency, but subsequently only took about 1.5 to 2 ms to change frequencies. RFX-series daughterboards can change frequency almost fast enough to do frequency hopping. The RFIC daughterboard took longer to change frequency for two reasons, one related to limitations in the driver itself, the other related to limitations in the USRP.

When GNU Radio sends the daughterboard driver a frequency to set the receiver, transmitter, or feedback loop, the driver first decides which frequency multiplier to use. This entire process is described in more detail in the Section 3.4.1 The RFIC Object. It then sets up that frequency multiplier, which is a two-to-four step process. Next, it sets the alignment variables, as described in Section 3.5 Tuning and Optimization. Finally, it calculates the frequency variables and sets the corresponding registers. Each register set in each step is set individually. That's up to 19 individual SPI write operations. Because up to 64 registers can be set with the same write operation, and all of the frequency-, multiplier- and optimization-related

registers are within 63 registers of one another for all three frequency synthesizers, that procedure could be cut down to two to four write operations. Setting the frequency multiplier still requires two to four passes. Furthermore, if the driver were able to detect if a frequency multiplier were in use, and if so, which one, that could be cut down to one write operation. Upon startup, no frequency multiplier would be set up, so the first time a frequency is set, the driver would have to set SPI registers two or three times. However, when changing frequency subsequently, the multiplier may already be set correctly. This is especially true for frequency-hopping applications, which normally only use a narrow range of frequencies – therefore the entire frequency range of the frequency-hopping protocol may be covered by a single frequency multiplier on the RFIC. That solution would require a re-tooling of the frequency-set functions within the RFIC object, along with an SPI write function that could handle multiple registers in a single pass. It could cut the frequency-switching time down by a factor of four or more – potentially down below 1 ms and possibly fast enough for some frequency-hopping protocols. The other limitation on the speed of frequency hopping is the speed of the USB 2.0 interface between the host computer and the USRP. The host computer must send SPI writes over the USB 2.0 connection, which has uncertain timing. This may improve with the Gigabit Ethernet interface of the USRP2.

I also need to run real-world tests. I used benchmark_tx.py and benchmark_tx_rfic.py (essentially the same as benchmark_tx.py, except that, like usrp_siggen_rfic.py, it is forced to recognize the RFIC daughterboard on the USRP) to transmit digital signals and benchmark_rx.py to receive digital signals. Benchmark_tx.py and benchmark_rx.py are standard components of GNU Radio, which allow the user to create a real-world digital radio link with a variety of bit-rates, a range of transmit power gain and receive gain, and a variety of modulation schemes. When transmitting with an RFX-series daughterboard, the RFIC was able to receive the signal consistently and correctly. When transmitting with the RFIC, an RFX-series daughterboard could only intermittently receive the signal. The RFX-series daughterboard almost never received the signal correctly. Clearly, I need to do more real-world testing. The RFIC transmitter also clearly needs work.

Finally, the entire RFIC daughterboard driver must be translated to C++. It is currently written in Python, and is therefore compatible with GNU Radio versions 3.0 and 3.1. GNU Radio 3.2, the latest version, uses daughterboard drivers written in C++. It is important for this daughterboard to work with all versions of GNU Radio, so the driver must be translated. I have had neither the time nor the C++ coding skill to attempt this yet. It is especially important because the USRP2 is only compatible with GNU Radio 3.2 and higher. If, in the future, GNU Radio uses different daughterboard drivers for the USRP, USRP2 or some new piece of hardware compatible with the RFIC daughterboard, it may have to be translated or rewritten again.

5.2 Conclusions

The RFIC daughterboard, with my driver, has the potential to revolutionize the GNU Radio and USRP world. With it, GNU Radio users will have access to a far broader range of frequencies than was ever available before. The Virginia Tech CWT lab, for instance, will be able to operate in every public safety band simultaneously with a single daughterboard. This will facilitate public safety interoperability.

Much of this work would be applicable for anyone who wants to build a daughterboard for the USRP. For instance, if one wanted to build a daughterboard for 700 MHz to 6 GHz based

on the AsicAhead AA 1001 chip, the structure of the code would be very similar. The AA 1001 is controlled through SPI, and is a CMOS direct-conversion transceiver with tunable bandwidth and multiple receiver and transmitter paths [30], just like the Motorola RFIC. While the SPI format and location and function of the registers would be different, most of the functions in my driver would exist in a similar form in an AA 1001-based daughterboard driver.

The transmitter still needs work, but the receiver functions very well. More optimization and tuning may improve the performance of the transmitter. Also, new revisions of the daughterboard hardware and of the RFIC itself may improve performance. Finally, in order to work with GNU Radio 3.2 and the USRP2, the driver must be translated from Python to C++.

More testing is also needed. Real-world analog and digital communication systems must be tested on the RFIC daughterboard. If they do not perform well, the board will have very limited usefulness.

I hope that the daughterboard driver code will be incorporated into GNU Radio in the future. Every GNU Radio user will be able to access and run it. Randall Nealy and Virginia Tech are working with Motorola and Ettus Research to come up with a deal to distribute the RFIC daughterboard commercially. This is the ideal scenario – any radio user, researcher or university will be able to buy a USRP and an RFIC daughterboard, download GNU Radio with the RFIC driver, and be able to access, operate on, and research radio waves across nearly the entire commonly-used spectrum.

Appendix A: The Driver Code

```
from gnuradio import usrp1, gru, eng_notation
import time, math, weakref
from math import floor

from usrpm import usrp_dbid
import db_base
import db_instantiator
from usrpm.usrp_fpga_regs import *

# Convenience function
n2s = eng_notation.num_to_str

# TX/RX Switch IO Pin (on the RX side, pin IO_RX_06)
TX_EN = (1 << 6)          # 1 = TX on, 0 = RX on

# ----------------------------------------------------------------------
# A few comments about the RFIC:
#
# The board is full duplex, meaning that the transmitter and receiver ca
# be used simultaneously.  There are seperate LOs for TX and RX as well
# as a third LO for the feedback from TX to RX, which can be used to
# offset non-linearity or DC offset.
# The feedback can be enabled from the receiver.  Receiver, transmitter
# and feedback can be set independently.  Gain and frequency can be
# controlled in all three modes, as well as phase offset in the LO
# and bandwidth.
# The board is a direct-conversion transciever, so bandwidth is measured
# at baseband and any received signal will come into the host computer
# at baseband.
#
# Each board is uniquely identified by the *USRP hardware* instance and side
# This dictionary holds a weak reference to existing board controller so it
# can be created or retrieved as needed.

_rfic_inst = weakref.WeakValueDictionary()
def _get_or_make_rfic(usrp, which):
      key = (usrp.serial_number(), which)
      if not _rfic_inst.has_key(key):
            print "Creating new RFIC instance"
            inst = rfic(usrp, which)
            _rfic_inst[key] = inst
      else:
            print "Using existing RFIC instance"
            inst = _rfic_inst[key]
      return inst

# ----------------------------------------------------------------------
# Common, shared object for RFIC board.  Transmit and receive classes
# operate on an instance of this; one instance is created per physical
# daughterboard.
```

```
class rfic(object):
      def __init__(self, usrp, which):
            print "RFIC: __init__ with %s: %d" % (usrp.serial_number(),
which)
            self.u = usrp
            self.which = which

            # For SPI interface, use MSB with two-byte header
            # Use RX side for SPI interface
            self.spi_format = usrp1.SPI_FMT_MSB | usrp1.SPI_FMT_HDR_2
            self.spi_format_no_header = usrp1.SPI_FMT_MSB |
usrp1.SPI_FMT_HDR_0
            self.spi_enable = (usrp1.SPI_ENABLE_RX_A,
usrp1.SPI_ENABLE_RX_B)[which]

            # Sane defaults:
            # For more information about setting each variable and SPI
register, see RFIC4 SPI Default Variables.xls


            #-----------------------------------------------------------------
----------------------------------------
            # TRANSMIT SIDE QuIET Frequency Generator
            #-----------------------------------------------------------------
----------------------------------------
            self.Ngt3 = 0 #Output frequency control bit.  Calculated.#
            self.NorNdiv4 = 1 #Output frequency control word.  Calculated.#
            self.RorFrNpRdiv4_25to18 = 0 #Output frequency control word.
Calculated#
            self.RorFrNpRdiv4_17to10 = 0 ##
            self.RorFrNpRdiv4_9to2 = 0 ##
            self.RorFrNpRdiv4_1to0 = 0 ##
            self.Qu_tx_Ngt3 = 0 #Enables divide-by-4 freq divider - Phase
shift control bit.  Calculated#
            self.NorNdiv4_phsh = 1 #Phase shift control word.  Calculated.#
            self.RorFrNpRdiv4_phsh_25to18 = 0 #Phase shift control word.
Calculated.#
            self.RorFrNpRdiv4_phsh_17to10 = 0 ##
            self.RorFrNpRdiv4_phsh_9to2 = 0 ##
            self.RorFrNpRdiv4_phsh_1to0 = 0 ##
            self.Passthru_ref_clk = 0 #A test mode where the 1 GHz input
reference is passed directly to the output#
            self.Byp_ram = 1 #Bypass the SRAMs#
            self.Dis_adr_dith = 1 #Disable the dither generator in the ca2adr
block#
            self.Dis_p5G_dith = 1 #Disable the dither generator in the
lup2decod block#
            self.Byp_fine = 1 #Bypass fine delay line control bit#
            self.Exclude32 = 0 #Bypass fine delay line control bit (exclude
32)#
            self.Dis_risedge = 0 #Disable the rising edges decoders#
            self.Dis_faledge = 0 #Disable the falling edges decoders#
            self.Spr_puls_en = 0 #enable spur pulsing#
            self.Spr_puls_val_a_9to3 = 0 #spur pulsing control word#
            self.Spr_pulse_val_2to0 = 0 ##
            self.Spr_puls_val_b_9to2 = 8 #spur pulsing control word#
```

```
            self.Spr_puls_val_b_1to0 = 0 ##
            self.Thru_ris_en = 0 #Put rising edges decoders into through-tap
mode#
            self.Thru_ris_tap_11to6 = 32 #Through-tap control word#
            self.Thru_ris_tap_5to0 = 0 ##
            self.Thru_fal_en = 0 #Put falling edges decoders into through-tap
mode#
            self.Thru_fal_tap_11to6 = 32 #Through-tap control word#
            self.Thru_fal_tap_5to0 = 0 ##
            self.Dig_delay = 0 #This bit provides delay to the clock going
into the digital block. It is a remnant of past designs and should always be
left off because the digClkPhase setting in address 23 provides much finer
control.#
            self.Clk_driver_en = 0 #This allows the clock to reach the
digital block. It first passes through the digital/analog clock
synchronization mux, which means that dlEn must be on (dlEn=1) and
Clk_driver=1 for the digital block to receive a clock.  See Byp_fine, address
10, bit 6#
            self.qu_reg_en = 0 #This bit enables the voltage regulators that
supply 1.2 V to all the analog block functions. There are 6 separate
regulators that are simultaneously enabled by this bit.#
            self.qq_reg_en = 0 #This bit enables the voltage regulators that
supply 1.2 V to all the Quad Gen functions. There are 3 separate regulators
that are simultaneously enabled by this bit.#
            self.win_rst = 0 #When this bit is high, the windowing function
is in a reset state, which means that no taps will be passed to the DDS
output regardless of the tap select signals coming from the digital block.#
            self.fineEn = 0 #This bit, when high, routes the coarse taps
through the fine line before reaching the output RS Flip Flop of the DDS.
When low, the coarse tap is routed directly to the output RS Flip Flop.#
            self.fineEnb = 0 #Opposite of  fineEn#
            self.rsffEn = 0 #This bit must be high to send the QuIET 0 and
180 degree calibration signals off chip. It does not control the RS Flip Flop
outputs of the DDS, though it may have some second order (coupling) effect.#
            self.dl_en = 1 #Allows the PLL reference to enter the QuIET delay
line when enabled.#
            self.cp_en = 1 #This bit, when enables, activates the charge pump
that controls the delay line via the single pole (one capacitor) DLL loop
filter.#
            self.forceCpUpb = 0 #This bit only matters when pdEn=0 (address
22, bit 1). When low, the pmos device connected to the DLL loop filter cap
turns on and sources current into the cap, thereby increasing the delay line
control voltage. #
            self.forceCpDn = 0 #This bit only matters when pdEn=0 (address
22, bit 1). When low, the nmos device connected to the DLL loop filter cap
turns off and allows the pmos device to charge up the loop cap as described
above.#
            self.pdUpTune_1to0 = 3 #These bits control the pulse width from
the phase detector into the charge up port of the charge pump. 00 turns the
charge up signal off. 01 is the minimum pulse width setting and 11 is the
maximum pulse width setting.#
            self.pdDnTune_1to0 = 0 #These bits control the pulse width from
the phase detector into the charge down port of the charge pump. 00 turns the
charge down signal off. 01 is the minimum pulse width setting and 11 is the
maximum pulse width setting.#
```

```
            self.cpUpTune_2to0 = 7 #These bits control amount of current that
is sourced while the charge up signal from the phase detector is high. 000 is
minimum current and 111 is maximum current.#
            self.cpDnTune_2to0 = 2 #These bits control amount of current that
is sinked while the charge down signal from the phase detector is high. 000
is minimum current and 111 is maximum current.#
            self.pdEn = 1 #When enables, the phase detector will send charge
up and down signals to the charge pump and over ride the forceCpUp and
forceCpDn settings in address 21. When disabled, the forceCpUp and forceCpDn
settings will control the charge pump.#
            self.digClkPhase_7to0 = 4 #Only one bit in this field should be
active at one time. This signal drives a mux that selects one of eight clock
phases from the delay line to drive the digital block. This is needed to
control the windowing function of the DDS.#
            self.Rst_n_async = 0 #Digital reset#
            self.L1_lup00_15to8 = [] #Read-only#
            self.L1_lup90_15to8 = [] #Read-only#
            self.Merg_ris_fin = [] #Read-only#
            self.Merg_fal_fin = [] #Read-only#
            self.Qg00degDelay_0to4 = 31 #Adjusts series delay in the 0 degree
path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.Qg90degDelay_0to4 = 7 #Adjusts series delay in the 90 degree
path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.Qg180degDelay_0to4 = 31 #Adjusts series delay in the 180
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.Qg270degDelay_0to4 = 7 #Adjusts series delay in the 270
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.DischargeTap16_0to3 = 15 #Adjusts DLL offset error in the
Quad Gen delay line by controlling down currents in one of the parallel
charge pumps.#
            self.ChargeTap16_0to3 = 4 #Adjusts DLL offset error in the Quad
Gen delay line by controlling up currents in one of the parallel charge
pumps.#
            self.DischargeTapn_0to3 = 15 #Adjusts DLL offset error in the
Quad Gen delay line by controlling down currents in one of the parallel
charge pumps.#
            self.ChargeTapn16_0to3 = 2 #Adjusts DLL offset error in the Quad
Gen delay line by controlling up currents in one of the parallel charge
pumps.#
            self.X1sel_32to39 = 0 #Control for the divide-by-two and x1
functions.#
            self.X1sel_40to47 = 0 #Control for the divide-by-two and x1
functions.#
            self.X2sel_32to36 = 0 #Control for the x2 function.#
            self.X2sel_37to41 = 0 #Control for the x2 function.#
            self.X4sel_32to36 = 0 #Control for the x4 function.#
            self.X4sel_37to41 = 0 #Control for the x4 function.#
            self.X8sel_32to36 = 0 #Bit 41 is used for a fine line windowing
control bit. If the fine line is needed, this bit needs to be set high if
Fout is close to Fref (greater than ~ 950 MHz) or for some testing modes like
pass through or thru_rise_en.#
            self.X8sel_41 = 0 #hiFout - set for passthrough and Fout close to
Fref#
            self.X8sel_37to40 = 0 ##
            self.qutx_fwd_180Cal_en = 0 #Enables the pad driver that sends
the falling edge signal off chip. This falling edge signal is used internally
to trigger the 'Reset' pin of the output RS Flip Flop.#
```

```
            self.qutx_fwd_0Cal_en = 0 #Enables the pad driver that sends the
rising edge signal off chip. This rising edge signal is used internally to
trigger the 'Set' pin of the output RS Flip Flop.#
            #-----------------------------------------------------------------
---------------------------------------
            # TRANSMIT FEEDBACK QuIET FREQUENCY GENERATOR
            #-----------------------------------------------------------------
---------------------------------------
            self.Ngt3_2 = 0 #Output frequency control bit.  Calculated.#
            self.NorNdiv4_2 = 1 #Output frequency control word.  Calculated.#
            self.RorFrNpRdiv4_25to18_2 = 0 #Output frequency control word.
Calculated.#
            self.RorFrNpRdiv4_17to10_2 = 0 ##
            self.RorFrNpRdiv4_9to2_2 = 0 ##
            self.RorFrNpRdiv4_1to0_2 = 0 ##
            self.Qu_tx_Ngt3_2 = 0 #Enables divide-by-4 freq divider - Phase
shift control bit.  Calculated#
            self.NorNdiv4_phsh_2 = 1 #Phase shift control word.  Calculated#
            self.RorFrNpRdiv4_phsh_25to18_2 = 0 #Phase shift control word.
Calculated#
            self.RorFrNpRdiv4_phsh_17to10_2 = 0 ##
            self.RorFrNpRdiv4_phsh_9to2_2 = 0 ##
            self.RorFrNpRdiv4_phsh_1to0_2 = 0 ##
            self.Passthru_ref_clk_2 = 0 #Enable reference clock pass-through
mode#
            self.Byp_ram_2 = 1 #Bypass the SRAMs#
            self.Dis_adr_dith_2 = 1 #Disable the dither generator in the
ca2adr block#
            self.Dis_p5G_dith_2 = 1 #Disable the dither generator in the
lup2decod block#
            self.Byp_fine_2 = 1 #Bypass fine delay line control bit#
            self.Exclude32_2 = 0 #Bypass fine delay line control bit (exclude
32)#
            self.Dis_risedge_2 = 0 #Disable the rising edges decoders#
            self.Dis_faledge_2 = 0 #Disable the falling edges decoders#
            self.Spr_puls_en_2 = 0 #Enable spur pulsing mode#
            self.Spr_puls_val_a_9to3_2 = 0 #Spur pulsing mode control word#
            self.Spr_pulse_val_2to0_2 = 0 ##
            self.Spr_puls_val_b_9to2_2 = 8 #Spur pulsing mode control word#
            self.Spr_puls_val_b_1to0_2 = 0 ##
            self.Thru_ris_en_2 = 0 #Put rising edges decoders into through-
tap mode#
            self.Thru_ris_tap_11to6_2 = 32 #Through-tap mode control word#
            self.Thru_ris_tap_5to0_2 = 0 #Through-tap mode control word#
            self.Thru_fal_en_2 = 0 #Put falling edges decoders into through-
tap mode#
            self.Thru_fal_tap_11to6_2 = 32 #Through-tap mode control word#
            self.Thru_fal_tap_5to0_2 = 0 #Through-tap mode control word#
            self.Dig_delay_2 = 0 #This bit provides delay to the clock going
into the digital block. It is a remnant of past designs and should always be
left off because the digClkPhase setting in address 23 provides much finer
control.#
            self.Clk_driver_en_2 = 0 #This bit provides delay to the clock
going into the digital block. It is a remnant of past designs and should
always be left off because the digClkPhase setting in address 23 provides
much finer control.  See Byp_fine, address 10, bit 6#
```

```
            self.qu_reg_en_2 = 0 #This bit enables the voltage regulators
that supply 1.2 V to all the analog block functions. There are 6 separate
regulators that are simultaneously enabled by this bit.#
            self.qq_reg_en_2 = 0 #This bit enables the voltage regulators
that supply 1.2 V to all the Quad Gen functions. There are 3 separate
regulators that are simultaneously enabled by this bit.#
            self.win_rst_2 = 0 #When this bit is high, the windowing function
is in a reset state, which means that no taps will be passed to the DDS
output regardless of the tap select signals coming from the digital block.#
            self.fineEn_2 = 0 #This bit, when high, routes the coarse taps
through the fine line before reaching the output RS Flip Flop of the DDS.
When low, the coarse tap is routed directly to the output RS Flip Flop.#
            self.fineEnb_2 = 0 #Opposite of  fineEn.#
            self.rsffEn_2 = 0 #This bit must be high to send the QuIET 0 and
180 degree calibration signals off chip. It does not control the RS Flip Flop
outputs of the DDS, though it may have some second order (coupling) effect.#
            self.dl_en_2 = 1 #Allows the PLL reference to enter the QuIET
delay line when enabled.#
            self.cp_en_2 = 1 #This bit, when enables, activates the charge
pump that controls the delay line via the single pole (one capacitor) DLL
loop filter.#
            self.forceCpUpb_2 = 0 #This bit only matters when pdEn=0 (address
22, bit 1). When low, the pmos device connected to the DLL loop filter cap
turns on and sources current into the cap, thereby increasing the delay line
control voltage. #
            self.forceCpDn_2 = 0 #This bit only matters when pdEn=0 (address
22, bit 1). When low, the nmos device connected to the DLL loop filter cap
turns off and allows the pmos device to charge up the loop cap as described
above.#
            self.pdUpTune_1to0_2 = 3 #These bits control the pulse width from
the phase detector into the charge up port of the charge pump. 00 turns the
charge up signal off. 01 is the minimum pulse width setting and 11 is the
maximum pulse width setting.#
            self.pdDnTune_1to0_2 = 0 #These bits control the pulse width from
the phase detector into the charge down port of the charge pump. 00 turns the
charge down signal off. 01 is the minimum pulse width setting and 11 is the
maximum pulse width setting.#
            self.cpUpTune_2to0_2 = 7 #These bits control amount of current
that is sourced while the charge up signal from the phase detector is high.
000 is minimum current and 111 is maximum current.#
            self.cpDnTune_2to0_2 = 2 #These bits control amount of current
that is sinked while the charge down signal from the phase detector is high.
000 is minimum current and 111 is maximum current.#
            self.pdEn_2 = 1 #When enables, the phase detector will send
charge up and down signals to the charge pump and over ride the forceCpUp and
forceCpDn settings in address 21. When disabled, the forceCpUp and forceCpDn
settings will control the charge pump.#
            self.digClkPhase_7to0_2 = 4 #Only one bit in this field should be
active at one time. This signal drives a mux that selects one of eight clock
phases from the delay line to drive the digital block. This is needed to
control the windowing function of the DDS.#
            self.Rst_n_async_2 = 0 #Digital reset#
            self.L1_lup00_15to8_2 = [] #Read-only#
            self.L1_lup90_15to8_2 = [] #Read-only#
            self.Merg_ris_fin_2 = [] #Read-only#
            self.Merg_fal_fin_2 = [] #Read-only#
```

```
            self.Qg00degDelay_0to4_2 = 31 #Adjusts series delay in the 0
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.Qg90degDelay_0to4_2 = 7 ##Adjusts series delay in the 90
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.
            self.Qg180degDelay_0to4_2 = 31 #Adjusts series delay in the 180
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.Qg270degDelay_0to4_2 = 7 #Adjusts series delay in the 270
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.DischargeTap16_3to0 = 15 #Adjusts DLL offset error in the
Quad Gen delay line by controlling down currents in one of the parallel
charge pumps.#
            self.ChargeTap16_3to0 = 4 #Adjusts DLL offset error in the Quad
Gen delay line by controlling up currents in one of the parallel charge
pumps.#
            self.DischargeTapn_3to0 = 15 #Adjusts DLL offset error in the
Quad Gen delay line by controlling down currents in one of the parallel
charge pumps.#
            self.ChargeTapn16_3to0 = 2 #Adjusts DLL offset error in the Quad
Gen delay line by controlling up currents in one of the parallel charge
pumps.#
            self.X1sel_32to39_2 = 0 #Control for the divide-by-two and x1
functions.#
            self.X1sel_40to47_2 = 0 #Control for the divide-by-two and x1
functions.#
            self.X2sel_32to36_2 = 0 #Control for the x2 function.#
            self.X2sel_37to41_2 = 0 #Control for the x2 function.#
            self.X4sel_32to36_2 = 0 #Control for the x4 function.#
            self.X4sel_37to41_2 = 0 #Control for the x4 function.#
            self.X8sel_32to36_2 = 0 #Bit 41 is used for a fine line windowing
control bit. If the fine line is needed, this bit needs to be set high if
Fout is close to Fref (greater than ~ 950 MHz) or for some testing modes like
pass through or thru_rise_en.#
            self.X8sel_41_2 = 0 #hiFout - set for passthrough and Fout close
to Fref#
            self.X8sel_37to40_2 = 0 ##
            self.qutx_fb_180Cal_en = 0 #Enables the pad driver that sends the
falling edge signal off chip. This falling edge signal is used internally to
trigger the 'Reset' pin of the output RS Flip Flop.#
            self.qutx_fb_0Cal_en = 0 #Enables the pad driver that sends the
rising edge signal off chip. This rising edge signal is used internally to
trigger the 'Set' pin of the output RS Flip Flop.#
            self.qutx_fb_180Rsff_en = 0 #off#
            self.qutx_fb_0Rsff_en = 0 #off#
            #-------------------------------------------------------------
----------------------------------------
            # QuIET Dm
            #-------------------------------------------------------------
----------------------------------------
            self.N = 4 ##
            self.R_11to8 = 13 ##
            self.R_7to0 = 172 ##
            self.Asyncrst_n = 0 #off#
            self.Cp_sel_6to0 = 63 ##
            self.Cp_sel_8to7 = 0 ##
            self.ForceFout = 0 #off#
            self.ForceFoutb = 0 #off#
            self.Out_en = 0 #off#
```

```
            self.Dll_en = 1 #on#
            self.Ana_en = 1 #off#
            self.Decod_in_0deg = [] #Read Only#
            #-------------------------------------------------------------------
-------------------------------------
            # RECEIVE QuIET FREQUENCY GENERATOR
            #-------------------------------------------------------------------
-------------------------------------
            self.Ngt3_3 = 0 #Output frequency control bit.  Calculated.#
            self.NorNdiv4_3 = 0 #Output frequency control word.  Calculated.#
            self.RorFrNpRdiv4_25to18_3 = 0 #Output frequency control word.
Calculated.#
            self.RorFrNpRdiv4_17to10_3 = 0 ##
            self.RorFrNpRdiv4_9to2_3 = 0 ##
            self.RorFrNpRdiv4_1to0_3 = 0 ##
            self.Qu_tx_Ngt3_3 = 0 #Enables divide-by-4 freq divider - Phase
shift control bit.  Calculated.#
            self.NorNdiv4_phsh_3 = 1 #Phase shift control word.  Calculated#
            self.RorFrNpRdiv4_phsh_25to18_3 = 0 #Phase shift control word.
Calculated.#
            self.RorFrNpRdiv4_phsh_17to10_3 = 0 ##
            self.RorFrNpRdiv4_phsh_9to2_3 = 0 ##
            self.RorFrNpRdiv4_phsh_1to0_3 = 0 ##
            self.Passthru_ref_clk_3 = 0 #Enable reference clock pass-through
mode#
            self.Byp_ram_3 = 1 #Bypass the SRAMs#
            self.Dis_adr_dith_3 = 1 #Disable the dither generator in the
ca2adr block#
            self.Dis_p5G_dith_3 = 1 #Disable the dither generator in the
lup2decod block#
            self.Byp_fine_3 = 1 #Bypass fine delay line control bit#
            self.Exclude32_3 = 0 #Bypass fine delay line control bit (exclude
32)#
            self.Dis_risedge_3 = 0 #Disable the rising edges decoders#
            self.Dis_faledge_3 = 0 #Disable the falling edges decoders#
            self.Spr_puls_en_3 = 0 #Enable spur pulsing mode#
            self.Spr_puls_val_a_9to3_3 = 0 #Spur pulsing mode control word#
            self.Spr_pulse_val_2to0_3 = 0 ##
            self.Spr_puls_val_b_9to2_3 = 8 #Spur pulsing mode control word#
            self.Spr_puls_val_b_1to0_3 = 0 ##
            self.Thru_ris_en_3 = 0 #Put rising edges decoders into through-
tap mode#
            self.Thru_ris_tap_11to6_3 = 32 #Through-tap mode control word#
            self.Thru_ris_tap_5to0_3 = 0 #Through-tap mode control word#
            self.Thru_fal_en_3 = 0 #Put falling edges decoders into through-
tap mode#
            self.Thru_fal_tap_11to6_3 = 0 #Through-tap mode control word#
            self.Thru_fal_tap_5to0_3 = 0 #Through-tap mode control word#
            self.Dig_delay_3 = 0 #This bit provides delay to the clock going
into the digital block. It is a remnant of past designs and should always be
left off because the digClkPhase setting in address 23 provides much finer
control.#
            self.Clk_driver_en_3 = 0 #This allows the clock to reach the
digital block. It first passes through the digital/analog clock
synchronization mux, which means that dlEn must be on (dlEn=1) and
Clk_driver=1 for the digital block to receive a clock.  See Byp_fine, address
10, bit 6#
```

```
            self.qu_reg_en_3 = 0 #This bit enables the voltage regulators
that supply 1.2 V to all the analog block functions. There are 6 separate
regulators that are simultaneously enabled by this bit.#
            self.qq_reg_en_3 = 0 #This bit enables the voltage regulators
that supply 1.2 V to all the Quad Gen functions. There are 3 separate
regulators that are simultaneously enabled by this bit.#
            self.win_rst_3 = 0 #When this bit is high, the windowing function
is in a reset state, which means that no taps will be passed to the DDS
output regardless of the tap select signals coming from the digital block.#
            self.fineEn_3 = 0 #This bit, when high, routes the coarse taps
through the fine line before reaching the output RS Flip Flop of the DDS.
When low, the coarse tap is routed directly to the output RS Flip Flop.#
            self.fineEnb_3 = 0 #Opposite of  fineEn.#
            self.rsffEn_3 = 0 #This bit must be high to send the QuIET 0 and
180 degree calibration signals off chip. It does not control the RS Flip Flop
outputs of the DDS, though it may have some second order (coupling) effect.#
            self.dl_en_3 = 1 #Allows the PLL reference to enter the QuIET
delay line when enabled.#
            self.cp_en_3 = 1 #This bit, when enables, activates the charge
pump that controls the delay line via the single pole (one capacitor) DLL
loop filter.#
            self.forceCpUpb_3 = 0 #This bit only matters when pdEn=0 (address
22, bit 1). When low, the pmos device connected to the DLL loop filter cap
turns on and sources current into the cap, thereby increasing the delay line
control voltage. #
            self.forceCpDn_3 = 0 #This bit only matters when pdEn=0 (address
22, bit 1). When low, the nmos device connected to the DLL loop filter cap
turns off and allows the pmos device to charge up the loop cap as described
above.#
            self.pdUpTune_1to0_3 = 3 #These bits control the pulse width from
the phase detector into the charge up port of the charge pump. 00 turns the
charge up signal off. 01 is the minimum pulse width setting and 11 is the
maximum pulse width setting.#
            self.pdDnTune_1to0_3 = 1 #These bits control the pulse width from
the phase detector into the charge down port of the charge pump. 00 turns the
charge down signal off. 01 is the minimum pulse width setting and 11 is the
maximum pulse width setting.#
            self.cpUpTune_2to0_3 = 7 #These bits control amount of current
that is sourced while the charge up signal from the phase detector is high.
000 is minimum current and 111 is maximum current.#
            self.cpDnTune_2to0_3 = 2 #These bits control amount of current
that is sinked while the charge down signal from the phase detector is high.
000 is minimum current and 111 is maximum current.#
            self.pdEn_3 = 1 #When enables, the phase detector will send
charge up and down signals to the charge pump and over ride the forceCpUp and
forceCpDn settings in address 21. When disabled, the forceCpUp and forceCpDn
settings will control the charge pump.#
            self.digClkPhase_7to0_3 = 4 #Only one bit in this field should be
active at one time. This signal drives a mux that selects one of eight clock
phases from the delay line to drive the digital block. This is needed to
control the windowing function of the DDS.#
            self.Rst_n_async_3 = 0 #Digital reset.#
            self.L1_lup00_15to8_3 = [] #Read-only#
            self.L1_lup90_15to8_3 = [] #Read-onnly#
            self.Merg_ris_fin_3 = [] #Read-only#
            self.Merg_fal_fin_3 = [] #Read-only#
```

```
            self.Qg00degDelay_0to4_3 = 31 #Adjusts series delay in the 0
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.Qg90degDelay_0to4_3 = 31 #Adjusts series delay in the 90
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.Qg180degDelay_0to4_3 = 31 #Adjusts series delay in the 180
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.Qg270degDelay_0to4_3 = 31 #Adjusts series delay in the 270
degree path for the divide-by-two, x1, x2, and x4 quadrature generators.#
            self.DischargeTap16_0to3_3 = 15 #Adjusts DLL offset error in the
Quad Gen delay line by controlling down currents in one of the parallel
charge pumps.#
            self.ChargeTap16_0to3_3 = 15 #Adjusts DLL offset error in the
Quad Gen delay line by controlling up currents in one of the parallel charge
pumps.#
            self.DischargeTapn_0to3_3 = 15 #Adjusts DLL offset error in the
Quad Gen delay line by controlling down currents in one of the parallel
charge pumps.#
            self.ChargeTapn16_0to3_3 = 15 #Adjusts DLL offset error in the
Quad Gen delay line by controlling up currents in one of the parallel charge
pumps.#
            self.X1sel_32to39_3 = 0 #Control for the divide-by-two and x1
functions.#
            self.X1sel_40to47_3 = 0 #Control for the divide-by-two and x1
functions.#
            self.X2sel_32to36_3 = 0 #Control for the x2 function.#
            self.X2sel_37to41_3 = 0 #Control for the x2 function.#
            self.X4sel_32to36_3 = 0 #Control for the x4 function.#
            self.X4sel_37to41_3 = 0 #Control for the x4 function.#
            self.X8sel_32to36_3 = 0 #Bit 41 is used for a fine line windowing
control bit. If the fine line is needed, this bit needs to be set high if
Fout is close to Fref (greater than ~ 950 MHz) or for some testing modes like
pass through or thru_rise_en.#
            self.X8sel_41_3 = 0 #hiFout - set for passthrough and Fout close
to Fref#
            self.X8sel_37to40_3 = 0 ##
            self.qurx_180Cal_en = 0 #Enables the pad driver that sends the
falling edge signal off chip. This falling edge signal is used internally to
trigger the 'Reset' pin of the output RS Flip Flop.#
            self.qurx_0Cal_en = 0 #Enables the pad driver that sends the
rising edge signal off chip. This rising edge signal is used internally to
trigger the 'Set' pin of the output RS Flip Flop.#
            #-------------------------------------------------------------
------------------------------------
            # PLL
            #-------------------------------------------------------------
------------------------------------
            self.extClkEn = 0 #PLL Reg 0#
            self.extClkEnBNOTD7 = 1 #on#
            self.div2_rst = 1 #on#
            self.TxChClkSel = 0 ##
            self.TxChClkEn = 0 #PLL Reg 1#
            #-------------------------------------------------------------
------------------------------------
            # TRANSMITTER
            #-------------------------------------------------------------
------------------------------------
```

```
            self.tx_bb_en = 0 #BB Fdbk Mux Buffer BW Control.  Enables the
Forward BB Reference Section of TX#
            self.tx_bb_fdbk_bw = 0 #Sets the BW of the BB Correction feedback
amp#
            self.tx_bb_fdbk_cal_en = 0 #BB Feedback Mux path Routing.  Shorts
the BB Correction feedback Amp input for self-calibration#
            self.tx_bb_fdbk_cart_err_en = 0 #Routes the Cartesian error
signal through the BB Correction feedback#
            self.tx_bb_fdbk_cart_fb_en = 0 #Routes the Cartesian feedback
signal through the BB Correction feedback#
            self.tx_bb_fdbk_cart_fwd_en = 0 #Routes the Cartesian reference
signal through the BB Correction feedback#
            self.tx_bb_fdbk_en = 0 #BB Feedback Mux path Routing.  Enables
the BB Correction feedback path via the RX pins#
            self.tx_bb_fdbk_1q_sel = 0 #Chooses between I or Q channel for
the BB Correction feedback path#
            self.tx_bb_fdbk_lp = 0 #BB Fdbk Mux Buffer current.  Sets the
current drive capability for BB Correction feedback Amp#
            self.tx_bb_fdbk_statt = 3 #BB Fdbk Mux Buffer Gain Control.  BB
Feedback Attenuator.  Sets the voltage gain for BB Correction feedback Amp#
            self.tx_bb_fdbk_swapi = 0 #Baseband Feedback Swap I & Ix.  Swaps
the I and Ix BB signals through the BB Correction feedback path#
            self.tx_bb_fdbk_swapq = 0 #Baseband feedback Swap Q & Qx.  Swaps
the Q and Qx BB signal through the BB Correction feedback path#
            self.tx_bb_gain_cmp = 1 #Baseband Gain 1 dB Compensation.  Adds
and extra 1.3 dB of Forward Baseband Reference Gain#
            self.tx_bb_lp = 0 #BB ref. stage current.  BB Amp Stage Current.
Sets the current drive capability for Forward BB Reference Amps#
            self.tx_bb_swapi = 1 #Baseband Swap I & Ix.  Swaps the I and Ix
BB signals through the Forward BB Reference Path#
            self.tx_bb_swapq = 0 #Baseband Swap Q & Qx.  Swaps the Q and Qx
BB signals through the Forward BB Reference Path#
            self.tx_butt_bw = 0 #BB ref. Butterworth filter BW control.  Sets
the BW of the Forward BB Reference 4-pole Butterworth Filters#
            self.tx_bw_trck = 5 #TX MIM cap tracking filter BW.  Bandwidth
Tracking.  Sets tracking BW of all the MIM cap based TX Filters (16 states)#
            self.tx_cart_en = 0 #Cartesian FB path Enable.  Enables the
Cartesian Baseband Section of Tx#
            self.tx_cart_fb_bb_statt = 15 #Cartesian down-mix path BB gain.
Cartesian FB path BB gain.  Sets the voltage gain for Cartesian BB down
converter PMA#
            self.tx_cart_fb_dcoc_dac_I1 = 32 #Sets Cartesian BB down
converter PMA Dc offset correction DAC I1#
            self.tx_cart_fb_dcoc_dac_I2 = 32 #Sets Cartesian BB down
converter PMA Dc offset correction DAC I2#
            self.tx_cart_fb_dcoc_dac_Q1 = 32 #Sets Cartesian BB down
converter PMA Dc offset correction DAC Q1#
            self.tx_cart_fb_dcoc_dac_Q2 = 32 #Sets Cartesian BB down
converter PMA Dc offset correction DAC Q2#
            self.CartesianFeedbackpathDCOCenable = 0 #Cartesian down-mix path
BB BW#
            self.CartesianFeedbackpathenable = 0 #off#
            self.CartesianFBpathHiResolutionDCOCenable = 0 #off#
            self.CartesianFBpathBW = 15 ##
            self.CartesianFBRFGain = 0 #Cartesian down conv. path RF Gain#
            self.CartesianFBpathSwapIandIx = 0 #Swap I & Ix BB in Down
Converter#
```

```
            self.CartesianFBpathSwapQandQx = 0 #off#
            self.CartesianFBpathSwitchtoforwardSummer = 0 #off#
            self.tx_cart_fb_lo_select = 0 #Cart. down conv LO curr. (tied to
Gain)#
            self.CartesianFBpathAmp1Gain = 3 ##
            self.CartesianFBpathAmp2Gain = 3 ##
            self.CartesianFBpathAmp3Gain = 3 ##
            self.CartesianFBpathAmp4Gain = 3 ##
            self.CartesianFBpathAmpCurrentSelect = 7 ##
            self.CartesianFBpathZeroEnable = 0 #off#
            self.tx_cart_zero_statt = 1 #Cartesian FB path Zero Gain.  Sets
the voltage gain for Cartesian Forward BB Zero Amp#
            self.tx_inbuf_bw = 0 #Sets the BW of the Forward BB Reference
Input Buffers#
            self.tx_inbuf_statt = 0 #Sets the attenuation of the Forward BB
Ref. Buffers#
            self.tx_output_channel_sel = 0 #Selects from the 3 RF Forward TX
output paths, 000 is full power down#
            self.tx_p1_bw = 0 #Sets the BW of the Cartesian Forward BB Loop
Pole 1#
            self.tx_pw_bw1 = 0 #Cartesian FB path Pole 2 Bandwidth.  Sets the
BW of the Cartesian Forward BB Loop Pole 2#
            self.tx_p2_bw2 = 0 #Cartesian FB path Pole 2 Bandwidth.  Sets the
BW of the Cartesian Forward BB Loop Pole 2#
            self.PushPullBufferCurrent = 7 ##
            self.tx_rf_aoc_bw = 0 #Sets the BW of the AOC control line#
            self.RFForwardPathEnable_toMUX = 0 #off#
            self.RFForwardPathEnable_ExternalPinenable = 1 #on#
            self.tx_rf_fwd_lp = 0 #RF Forward Bias Reference Control.  RF
Forward Path Current Drain Select.  Sets the current drive capability for
Forward RF Output Drivers#
            self.tx_rf_fwd_statt1 = 0 #RF Passive Step Attenuator control.
RF Forward Path Step Attn1.  Sets the attenuation level for the RF Step
attenuators#
            self.tx_rf_fwd_statt2 = 0 #RF Output Driver Step Attn. Control.
RF Forward Path Step Attn2.  Sets the attenuation level for the RF Output
Drivers#
            self.BBQDivideby2or4Select = 0 #BBQ Quad Gen Divide by 2 or 4
(High=1/4)#
            self.BBQQuadGenEnable = 0 #Bypass Quiet LO with external LO#
            self.BBQPolyphaseQuadGenEnable = 0 #off#
            self.lofb_tun_s = 8 ##
            self.lofb_tun_sx = 8 ##
            self.lofw_tun_s2 = 8 ##
            self.lofw_tun_sx2 = 8 ##
            self.reserve_tx26 = 0 ##
            self.reserve_tx27 = 0 ##
            #-----------------------------------------------------------------
---------------------------------------
            # RECEIVER
            #-----------------------------------------------------------------
---------------------------------------
            self.rx_Idac = 16 #I path DCOC DAC setting.  Digital values for
the DC offset adjustment.  11111 represents the maximum positive offset
adjust and 00000 represents the maximum negative offset adjust.  By design,
codes 10000 and 01111 cause no change in the offset voltage.#
```

```
            self.rx_dcs = 0 #DCOC step size select.  Selects the proper
current reference in the DAC to maintain constant step size at ouptut of
baseband filters.  This value works in tandem with the BiQuad Gain Select
(address 198, bits 4:3) to maintain a constant step size of 24 mV at filter
output.#
            self.rx_den = 0 #Enables the DC offset correction DACs in the I
and Q path.#
            self.rx_Qdac = 12 #Q path DCOC DAC setting.  Digital values for
the DC offset adjustment.  11111 represents the maximum positive offset
adjust and 00000 represents the maximum negative offset adjust.  By design,
codes 10000 and 01111 cause no change in the offset voltage.#
            self.rx_cmpen = 0 #Enables the DC offset correction comparator
used in the DCOC circuitry.#
            self.rx_dcoc = 0 #Enables the DC offset correction circuitry for
automatic correction of the DC offset in the baseband filters.#
            self.rx_ten = 0 #Enables the RC tuning circuit to tune RX and TX
baseband filters.#
            self.rx_ren = 0 #Enables the ramp circuit used in the RC tuning
circuitry to tune the RX and TX baseband filters.#
            self.rx_dven = 0 ##
            self.rx_dv = 0 #DCOC/tune clock divider select.  Selects the
clock rate used for clocking the DCOC and RC tuning circuit.  Bits 3 and 2
set the divider setting used for both the DCOC circuitry and RC Tune
circuitry.  Bits 1 and 0 set the divider setting for the dedicated divider in
the DCOC circuitry.  Table below shows the mapping of divider settings.  The
DCOC clock divider setting is the total divide ratio of both dividers.  The
maximum divide ratio is 8*8 = 64.#
            self.rx_extc = 0 #Enables the external capacitor pins to allow
for external low-frequency pole to be placed in the signal path between the
mixer and baseband filter.#
            self.rx_cen = 0 #Chopper enable for filter stages.  Settings to
enable which amplifier the clock is being applied#
            self.rx_chck = 0 #Divider setting for the chopper clock#
            self.rx_chcken = 0 #Enables the baseband filter chopper clock.#
            self.rx_fen = 0 #Enables baseband filters.  0 puts filter in
power save mode.#
            self.rx_onchen = 0 #Enables on-channel detector.#
            self.rx_offchen = 0 #Enables off-channel detector#
            self.rx_foe = 0 #Enables the output of the baseband filters.
Otherwise the baseband filter outputs are in a Hi-Z state to allow
transmitter to use filter output pins.  When Filter Enable is set LOW,
outputs are disabled (Hi-Z)#
            self.rx_offch = 1 #Sets the Clip Threshold for the Off-channel
Detector#
            self.rx_onchf = 0 #Sets the Fade Threshold for the On-channel
Detector relative to the On-channel clip point.#
            self.rx_onchc = 2 #Sets the Clip Threshold for the On-channel
Detector#
            self.rx_qs = 0 #Sets the BiQuad filter Q#
            self.rx_bqg = 0 #Set BiQuad filter gain#
            #FIXME Maybe set rx_rq to 0
            self.rx_rq = 1 #Sets the BiQuad filter resistor value.  The
natural frequency of the BiQuad (wo) is this resistor value multiplied by the
BiQuad Capacitor value.#
            self.rx_rv = 1 #Sets the VGA filter (passive filter) resistor
value.  The pole frequency of the passive filter is this resistor value
multiplied by the VGA capacitor value.#
```

```
            self.rx_rip = 0 #Sets the MPA input resistor value that sets the
gain of the PMA.  Gain of the PMA is Rf/Rin where Rf is the PMA feedback
resistor and Rin is the input resistor.  Note that the input resistance
remains at 2 kohm differential for all settings.  An R2R ladder is used to
accomplish this while changing the Rin value.#
            self.rx_rfp = 2 #Sets the PMA feedback resistor value that sets
the gain of the PMA as well as the pole frequency (along with PMA capacitor
value).  Gain of the PMA is Rf/Rin where Rf is the PMA feedback resistor and
Rin is the input resistor.#
            self.rx_cp_12to8 = 0 #Sets the PMA filter capacitor value.  The
pole frequency of the PMA filter is the PMA feedback resistor value
multiplied by this Capacitor value.  PMA Capacitor (in pF) = (PMAC) * 0.0625
+ 1#
            self.rx_gs = 0 #Sets the gain of the VGA in the baseband filter#
            self.rx_cp_7to0 = 0 #PMA cap select LSBs.  Sets the PMA filter
capacitor value.  The pole frequency of the PMA filter is the PMA feedback
resistor value multiplied by this Capacitor value.  PMA Capacitor (in pF) =
(PMAC) * 0.0625 + 1#
            self.rx_cv_10to3 = 0 #VGA cap select MSBs.  Sets the VGA
(passive) filter capacitor value.  This pole frequency of the passive filter
is the VGA resistor value multiplied by this Capacitor value.  VGA Capacitor
(in pF) = (VGAC) * 0.0625 + 1#
            self.rx_cv_2to0 = 0 #VGA cap select LSBs.  Sets the VGA (passive)
filter capacitor value.  This pole frequency of the passive filter is the VGA
resistor value multiplied by this Capacitor value.  VGA Capacitor (in pF) =
(VGAC) * 0.0625 + 1#
            self.rx_cc_2to0 = 0 #Compensation control.  Disables additional
compensation capacitance in the VGA and BiQuad op-amps to allow for higher
bandwidths.  Also increases the op-ampdominate pole-frequency which improves
filter response.  Bit 4 controls the VGA amplifier, Bit 3 controls the
feedback amplifier in the BiQuad, and Bit 2 controls the output buffer in the
BiQuad.#
            self.rx_cq_9to8 = 0 #BiQuad cap select MSBs.  Sets the BiQuad
filter capacitor value.  The natural frequency of the BiQuad (wo) is the
BiQuad resistor value multiplied by this Capacitor value.  BiQuad Capacitor
(in pF) = (BiQuadC) * 0.125 + 2#
            self.rx_cq_7to0 = 0 #BiQuad cap select LSBs.  Sets the BiQuad
filter capacitor value.  The natural frequency of the BiQuad (wo) is the
BiQuad resistor value multiplied by this Capacitor value.  BiQuad Capacitor
(in pF) = (BiQuadC) * 0.125 + 2#
            self.rx_lna = 1 #LNA select#
            self.rx_lnab = 0 #LNA bias select#
            self.rx_rxchen = 0 #RX mixer enable.  Must be set to 1 to enable
Mixer operation#
            self.rx_bbq_div2or4 = 0 #Selects divide ratio of RX Quad Gen when
using external LO.  0->DIV2, 1 ->DIV1#
            self.rx_Loselect = 0 #RX external LO select.  Enables external LO
clock source#
            self.poly_en = 0 #off#
            self.lorx_tun_s = 8 ##
            self.lorx_tun_sx = 8 ##
            self.rx_Icmpo = [] #I path DCOC comparator output.  Output of the
DCOC comparator - used for test purposes.  Output only.#
            self.rx_Iodac = [] #I path DCOC DAC output.  Output of the DCOC
DACs - used to read result of DCOC correction circuitry.  Output only.#
            self.rx_Qcmpo = [] #Q path DCOC comparator output.  Output of the
DCOC comparator - used for test purposes.  Output only.#
```

```
            self.rx_Qodac = [] #Q path DCOC DAC output.  Output of the DCOC
DACs - used to read result of DCOC correction circuitry.  Output only.#
            self.rx_rc = [] #Output word from RC Tune circuit that is used to
calculate adjustment needed to TX and RX filter bandwidths for correct
tuning.  Output only.#
            #-------------------------------------------------------------------
-------------------------------------
            # VAG Generator
            #-------------------------------------------------------------------
-------------------------------------
            self.shft_cml_in = 0 #Enable - 150mV level shift of Ref. BB VAG#
            self.vagenable1 = 1 #Enable VAG Gen into Sleep Mode (slow ramp
up)#
            self.vagenable2 = 1 #Enable VAG Gen in Full On Mode (Fast ramp
from sleep)#
            #-------------------------------------------------------------------
-------------------------------------
            # TEST MULTIPLEXER
            #-------------------------------------------------------------------
-------------------------------------
            self.TestMuxBufferEnable = 0 #Enable Test Mux Buffer#
            self.TestMuxEnable = 0 #Enable Test Mux#
            self.TestMuxSetting = 0 #Four Output Description (Test1, Test2,
Test3, Test4)#

            self.txgain = 0   #Set Transmit Gain#

            self.Fclk = 1000e6 #Default clock frequency, in Hz#

            self.Fouttx = 0   # Default tx frequency is zero#
            self.Foutrx = 0   # Default rx frequency is zero#
            self.Foutfb = 0 # Default feedback frequency is zero#

            # Initialize GPIO and ATR
            # GPIO are the general-purpose IO pins on the daughterboard
            # IO_RX_06 must be used for ATR (1 = TX, 0 = RX)
            # ATR is the automatic transmit/receive switching, done in the
FPGA
            # FIXME
            self.rx_write_io(0, TX_EN)
            self.rx_write_oe(TX_EN, TX_EN)
            self.rx_set_atr_rxval(0)
            self.rx_set_atr_txval(TX_EN)
            self.rx_set_atr_mask(TX_EN)

            # Initialize Chipset
            # Set initial SPI values
            # Neither transmit nor receive currently on

            self.set_reg_0()
            self.set_reg_1()
            self.set_reg_2()
            self.set_reg_3()
            self.set_reg_4()
            self.set_reg_5()
            self.set_reg_6()
            self.set_reg_7()
```

```
self.set_reg_8()
self.set_reg_9()
self.set_reg_10()
self.set_reg_12()
self.set_reg_13()
self.set_reg_14()
self.set_reg_15()
self.set_reg_16()
self.set_reg_17()
self.set_reg_18()
self.set_reg_19()
self.set_reg_20()
self.set_reg_21()
self.set_reg_22()
self.set_reg_23()
self.set_reg_24()
self.set_reg_29()
self.set_reg_30()
self.set_reg_31()
self.set_reg_32()
self.set_reg_33()
self.set_reg_34()
self.set_reg_35()
self.set_reg_36()
self.set_reg_37()
self.set_reg_38()
self.set_reg_39()
self.set_reg_40()
self.set_reg_41()
self.set_reg_42()
self.set_reg_43()
self.set_reg_48()
self.set_reg_49()
self.set_reg_50()
self.set_reg_51()
self.set_reg_52()
self.set_reg_53()
self.set_reg_54()
self.set_reg_55()
self.set_reg_56()
self.set_reg_57()
self.set_reg_58()
self.set_reg_60()
self.set_reg_61()
self.set_reg_62()
self.set_reg_63()
self.set_reg_64()
self.set_reg_65()
self.set_reg_66()
self.set_reg_67()
self.set_reg_68()
self.set_reg_69()
self.set_reg_70()
self.set_reg_71()
self.set_reg_72()
self.set_reg_77()
self.set_reg_78()
```

```
self.set_reg_79()
self.set_reg_80()
self.set_reg_81()
self.set_reg_82()
self.set_reg_83()
self.set_reg_84()
self.set_reg_85()
self.set_reg_86()
self.set_reg_87()
self.set_reg_88()
self.set_reg_89()
self.set_reg_90()
self.set_reg_91()
self.set_reg_96()
self.set_reg_97()
self.set_reg_98()
self.set_reg_99()
self.set_reg_104()
self.set_reg_105()
self.set_reg_106()
self.set_reg_107()
self.set_reg_108()
self.set_reg_109()
self.set_reg_110()
self.set_reg_111()
self.set_reg_112()
self.set_reg_113()
self.set_reg_114()
self.set_reg_116()
self.set_reg_117()
self.set_reg_118()
self.set_reg_119()
self.set_reg_120()
self.set_reg_121()
self.set_reg_122()
self.set_reg_123()
self.set_reg_124()
self.set_reg_125()
self.set_reg_126()
self.set_reg_127()
self.set_reg_128()
self.set_reg_133()
self.set_reg_134()
self.set_reg_135()
self.set_reg_136()
self.set_reg_137()
self.set_reg_138()
self.set_reg_139()
self.set_reg_140()
self.set_reg_141()
self.set_reg_142()
self.set_reg_143()
self.set_reg_144()
self.set_reg_145()
self.set_reg_146()
self.set_reg_147()
self.set_reg_152()
```

```
        self.set_reg_153()
        self.set_reg_156()
        self.set_reg_157()
        self.set_reg_158()
        self.set_reg_159()
        self.set_reg_160()
        self.set_reg_161()
        self.set_reg_162()
        self.set_reg_163()
        self.set_reg_164()
        self.set_reg_165()
        self.set_reg_166()
        self.set_reg_167()
        self.set_reg_168()
        self.set_reg_169()
        self.set_reg_170()
        self.set_reg_171()
        self.set_reg_172()
        self.set_reg_173()
        self.set_reg_174()
        self.set_reg_175()
        self.set_reg_176()
        self.set_reg_177()
        self.set_reg_178()
        self.set_reg_179()
        self.set_reg_180()
        self.set_reg_181()
        self.set_reg_192()
        self.set_reg_193()
        self.set_reg_194()
        self.set_reg_195()
        self.set_reg_196()
        self.set_reg_197()
        self.set_reg_198()
        self.set_reg_199()
        self.set_reg_200()
        self.set_reg_201()
        self.set_reg_202()
        self.set_reg_203()
        self.set_reg_204()
        self.set_reg_205()
        self.set_reg_206()
        self.set_reg_207()
        self.set_reg_220()
        self.set_reg_222()

        #self.set_reg_220()
        #self.set_reg_222()

    def __del__(self):
        # Delete instance, shut down
        # FIXME
        print "RFIC: __del__"

        # Reset all three QuIET synthesizers
        self.Rst_n_async = 0
        self.set_reg_24()
```

```
            self.Rst_n_async2 = 0
            self.set_reg_72()
            self.Rst_n_async3 = 0
            self.set_reg_128()

            self.X1sel_32to39_3 = 0
            self.X1sel_40to47_3 = 0
            self.X2sel_32to36_3 = 0
            self.X2sel_37to41_3 = 0
            self.X4sel_32to36_3 = 0
            self.X4sel_37to41_3 = 0

            self.set_reg_139()
            self.set_reg_140()
            self.set_reg_141()
            self.set_reg_142()
            self.set_reg_143()
            self.set_reg_144()

            self.X1sel_32to39 = 0
            self.X1sel_40to47 = 0
            self.X2sel_32to36 = 0
            self.X2sel_37to41 = 0
            self.X4sel_32to36 = 0
            self.X4sel_37to41 = 0

            self.set_reg_35()
            self.set_reg_36()
            self.set_reg_37()
            self.set_reg_38()
            self.set_reg_39()
            self.set_reg_40()

            self.X1sel_32to39_2 = 0
            self.X1sel_40to47_2 = 0
            self.X2sel_32to36_2 = 0
            self.X2sel_37to41_2 = 0
            self.X4sel_32to36_2 = 0
            self.X4sel_37to41_2 = 0

            self.set_reg_83()
            self.set_reg_84()
            self.set_reg_85()
            self.set_reg_86()
            self.set_reg_87()
            self.set_reg_88()

      # ------------------------------------------------------------------
      # These methods set the RFIC onboard registers over the SPI bus.
      # Thus, the shift values here are the 0-7 values from the data sheet

      # For more information about setting each variable and SPI register,
see RFIC4 SPI Default Variables.xls

      def set_reg_0(self):
            reg_0 = (
            self.Ngt3 << 7 |
```

```
      self.NorNdiv4 << 0 )
      self.send_reg(0, reg_0)
def set_reg_1(self):
      reg_1 = (
      self.RorFrNpRdiv4_25to18 << 0 )
      self.send_reg(1, reg_1)
def set_reg_2(self):
      reg_2 = (
      self.RorFrNpRdiv4_17to10 << 0 )
      self.send_reg(2, reg_2)
def set_reg_3(self):
      reg_3 = (
      self.RorFrNpRdiv4_9to2 << 0 )
      self.send_reg(3, reg_3)
def set_reg_4(self):
      reg_4 = (
      self.RorFrNpRdiv4_1to0 << 6 )
      self.send_reg(4, reg_4)
def set_reg_5(self):
      reg_5 = (
      self.Qu_tx_Ngt3 << 7 |
      self.NorNdiv4_phsh << 0 )
      self.send_reg(5, reg_5)
def set_reg_6(self):
      reg_6 = (
      self.RorFrNpRdiv4_phsh_25to18 << 0 )
      self.send_reg(6, reg_6)
def set_reg_7(self):
      reg_7 = (
      self.RorFrNpRdiv4_phsh_17to10 << 0 )
      self.send_reg(7, reg_7)
def set_reg_8(self):
      reg_8 = (
      self.RorFrNpRdiv4_phsh_9to2 << 0 )
      self.send_reg(8, reg_8)
def set_reg_9(self):
      reg_9 = (
      self.RorFrNpRdiv4_phsh_1to0 << 6 )
      self.send_reg(9, reg_9)
def set_reg_10(self):
      reg_10 = (
      self.Passthru_ref_clk << 7 |
      self.Byp_ram << 6 |
      self.Dis_adr_dith << 5 |
      self.Dis_p5G_dith << 4 |
      self.Byp_fine << 3 |
      self.Exclude32 << 2 |
      self.Dis_risedge << 1 |
      self.Dis_faledge << 0 )
      self.send_reg(10, reg_10)
def set_reg_12(self):
      reg_12 = (
      self.Spr_puls_en << 7 |
      self.Spr_puls_val_a_9to3 << 0 )
      self.send_reg(12, reg_12)
def set_reg_13(self):
      reg_13 = (
```

```
      self.Spr_pulse_val_2to0 << 5 )
      self.send_reg(13, reg_13)
def set_reg_14(self):
      reg_14 = (
      self.Spr_puls_val_b_9to2 << 0 )
      self.send_reg(14, reg_14)
def set_reg_15(self):
      reg_15 = (
      self.Spr_puls_val_b_1to0 << 6 )
      self.send_reg(15, reg_15)
def set_reg_16(self):
      reg_16 = (
      self.Thru_ris_en << 7 |
      self.Thru_ris_tap_11to6 << 1 )
      self.send_reg(16, reg_16)
def set_reg_17(self):
      reg_17 = (
      self.Thru_ris_tap_5to0 << 2 )
      self.send_reg(17, reg_17)
def set_reg_18(self):
      reg_18 = (
      self.Thru_fal_en << 7 |
      self.Thru_fal_tap_11to6 << 1 )
      self.send_reg(18, reg_18)
def set_reg_19(self):
      reg_19 = (
      self.Thru_fal_tap_5to0 << 2 )
      self.send_reg(19, reg_19)
def set_reg_20(self):
      reg_20 = (
      self.Dig_delay << 7 |
      self.Clk_driver_en << 6 |
      self.qu_reg_en << 5 |
      self.qq_reg_en << 4 |
      self.win_rst << 3 |
      self.fineEn << 2 |
      self.fineEnb << 1 |
      self.rsffEn << 0 )
      self.send_reg(20, reg_20)
def set_reg_21(self):
      reg_21 = (
      self.dl_en << 7 |
      self.cp_en << 6 |
      self.forceCpUpb << 5 |
      self.forceCpDn << 4 |
      self.pdUpTune_1to0 << 2 |
      self.pdDnTune_1to0 << 0 )
      self.send_reg(21, reg_21)
def set_reg_22(self):
      reg_22 = (
      self.cpUpTune_2to0 << 5 |
      self.cpDnTune_2to0 << 2 |
      self.pdEn << 1 )
      self.send_reg(22, reg_22)
def set_reg_23(self):
      reg_23 = (
      self.digClkPhase_7to0 << 0 )
```

```
      self.send_reg(23, reg_23)
def set_reg_24(self):
      reg_24 = (
      self.Rst_n_async << 7 )
      self.send_reg(24, reg_24)
def read_reg_25(self):
      reg_25 = self.get_reg(25)
      self.L1_lup00_15to8 = reg_25

def read_reg_26(self):
      reg_26 = self.get_reg(26)
      self.L1_lup90_15to8 = reg_26

def read_reg_27(self):
      reg_27 = self.get_reg(27)
      self.Merg_ris_fin = reg_27 >> 2

def read_reg_28(self):
      reg_28 = self.get_reg(28)
      self.Merg_fal_fin = reg_28 >> 2

def set_reg_29(self):
      reg_29 = (
      self.Qg00degDelay_0to4 << 3 )
      self.send_reg(29, reg_29)
def set_reg_30(self):
      reg_30 = (
      self.Qg90degDelay_0to4 << 3 )
      self.send_reg(30, reg_30)
def set_reg_31(self):
      reg_31 = (
      self.Qg180degDelay_0to4 << 3 )
      self.send_reg(31, reg_31)
def set_reg_32(self):
      reg_32 = (
      self.Qg270degDelay_0to4 << 3 )
      self.send_reg(32, reg_32)
def set_reg_33(self):
      reg_33 = (
      self.DischargeTap16_0to3 << 4 |
      self.ChargeTap16_0to3 << 0 )
      self.send_reg(33, reg_33)
def set_reg_34(self):
      reg_34 = (
      self.DischargeTapn_0to3 << 4 |
      self.ChargeTapn16_0to3 << 0 )
      self.send_reg(34, reg_34)
def set_reg_35(self):
      reg_35 = (
      self.X1sel_32to39 << 0 )
      self.send_reg(35, reg_35)
def set_reg_36(self):
      reg_36 = (
      self.X1sel_40to47 << 0 )
      self.send_reg(36, reg_36)
def set_reg_37(self):
      reg_37 = (
```

```
      self.X2sel_32to36 << 3 )
      self.send_reg(37, reg_37)
def set_reg_38(self):
      reg_38 = (
      self.X2sel_37to41 << 3 )
      self.send_reg(38, reg_38)
def set_reg_39(self):
      reg_39 = (
      self.X4sel_32to36 << 3 )
      self.send_reg(39, reg_39)
def set_reg_40(self):
      reg_40 = (
      self.X4sel_37to41 << 3 )
      self.send_reg(40, reg_40)
def set_reg_41(self):
      reg_41 = (
      self.X8sel_32to36 << 3 )
      self.send_reg(41, reg_41)
def set_reg_42(self):
      reg_42 = (
      self.X8sel_41 << 7 |
      self.X8sel_37to40 << 3 )
      self.send_reg(42, reg_42)
def set_reg_43(self):
      reg_43 = (
      self.qutx_fwd_180Cal_en << 7 |
      self.qutx_fwd_0Cal_en << 6 )
      self.send_reg(43, reg_43)
def set_reg_48(self):
      reg_48 = (
      self.Ngt3_2 << 7 |
      self.NorNdiv4_2 << 0 )
      self.send_reg(48, reg_48)
def set_reg_49(self):
      reg_49 = (
      self.RorFrNpRdiv4_25to18_2 << 0 )
      self.send_reg(49, reg_49)
def set_reg_50(self):
      reg_50 = (
      self.RorFrNpRdiv4_17to10_2 << 0 )
      self.send_reg(50, reg_50)
def set_reg_51(self):
      reg_51 = (
      self.RorFrNpRdiv4_9to2_2 << 0 )
      self.send_reg(51, reg_51)
def set_reg_52(self):
      reg_52 = (
      self.RorFrNpRdiv4_1to0_2 << 6 )
      self.send_reg(52, reg_52)
def set_reg_53(self):
      reg_53 = (
      self.Qu_tx_Ngt3_2 << 7 |
      self.NorNdiv4_phsh_2 << 0 )
      self.send_reg(52, reg_53)
def set_reg_54(self):
      reg_54 = (
      self.RorFrNpRdiv4_phsh_25to18_2 << 0 )
```

```
      self.send_reg(54, reg_54)
def set_reg_55(self):
      reg_55 = (
      self.RorFrNpRdiv4_phsh_17to10_2 << 0 )
      self.send_reg(55, reg_55)
def set_reg_56(self):
      reg_56 = (
      self.RorFrNpRdiv4_phsh_9to2_2 << 0 )
      self.send_reg(56, reg_56)
def set_reg_57(self):
      reg_57 = (
      self.RorFrNpRdiv4_phsh_1to0_2 << 6 )
      self.send_reg(57, reg_57)
def set_reg_58(self):
      reg_58 = (
      self.Passthru_ref_clk_2 << 7 |
      self.Byp_ram_2 << 6 |
      self.Dis_adr_dith_2 << 5 |
      self.Dis_p5G_dith_2 << 4 |
      self.Byp_fine_2 << 3 |
      self.Exclude32_2 << 2 |
      self.Dis_risedge_2 << 1 |
      self.Dis_faledge_2 << 0 )
      self.send_reg(58, reg_58)
def set_reg_60(self):
      reg_60 = (
      self.Spr_puls_en_2 << 7 |
      self.Spr_puls_val_a_9to3_2 << 0 )
      self.send_reg(60, reg_60)
def set_reg_61(self):
      reg_61 = (
      self.Spr_pulse_val_2to0_2 << 5 )
      self.send_reg(61, reg_61)
def set_reg_62(self):
      reg_62 = (
      self.Spr_puls_val_b_9to2_2 << 0 )
      self.send_reg(62, reg_62)
def set_reg_63(self):
      reg_63 = (
      self.Spr_puls_val_b_1to0_2 << 6 )
      self.send_reg(63, reg_63)
def set_reg_64(self):
      reg_64 = (
      self.Thru_ris_en_2 << 7 |
      self.Thru_ris_tap_11to6_2 << 1 )
      self.send_reg(64, reg_64)
def set_reg_65(self):
      reg_65 = (
      self.Thru_ris_tap_5to0_2 << 2 )
      self.send_reg(65, reg_65)
def set_reg_66(self):
      reg_66 = (
      self.Thru_fal_en_2 << 7 |
      self.Thru_fal_tap_11to6_2 << 1 )
      self.send_reg(66, reg_66)
def set_reg_67(self):
      reg_67 = (
```

```
      self.Thru_fal_tap_5to0_2 << 2 )
      self.send_reg(67, reg_67)
def set_reg_68(self):
      reg_68 = (
      self.Dig_delay_2 << 7 |
      self.Clk_driver_en_2 << 6 |
      self.qu_reg_en_2 << 5 |
      self.qq_reg_en_2 << 4 |
      self.win_rst_2 << 3 |
      self.fineEn_2 << 2 |
      self.fineEnb_2 << 1 |
      self.rsffEn_2 << 0 )
      self.send_reg(68, reg_68)
def set_reg_69(self):
      reg_69 = (
      self.dl_en_2 << 7 |
      self.cp_en_2 << 6 |
      self.forceCpUpb_2 << 5 |
      self.forceCpDn_2 << 4 |
      self.pdUpTune_1to0_2 << 2 |
      self.pdDnTune_1to0_2 << 0 )
      self.send_reg(69, reg_69)
def set_reg_70(self):
      reg_70 = (
      self.cpUpTune_2to0_2 << 5 |
      self.cpDnTune_2to0_2 << 2 |
      self.pdEn_2 << 1 )
      self.send_reg(70, reg_70)
def set_reg_71(self):
      reg_71 = (
      self.digClkPhase_7to0_2 << 0 )
      self.send_reg(71, reg_71)
def set_reg_72(self):
      reg_72 = (
      self.Rst_n_async_2 << 7 )
      self.send_reg(72, reg_72)
def read_reg_73(self):
      reg_73 = self.get_reg(73)
      self.L1_lup00_15to8_2 = reg_73

def read_reg_74(self):
      reg_74 = self.get_reg(74)
      self.L1_lup90_15to8_2 = reg_74

def read_reg_75(self):
      reg_75 = self.get_reg(75)
      self.Merg_ris_fin_2 = reg_75 >> 2

def read_reg_76(self):
      reg_76 = self.get_reg(76)
      self.Merg_fal_fin_2 = reg_76 >> 2

def set_reg_77(self):
      reg_77 = (
      self.Qg00degDelay_0to4_2 << 3 )
      self.send_reg(77, reg_77)
def set_reg_78(self):
```

```
      reg_78 = (
      self.Qg90degDelay_0to4_2 << 3 )
      self.send_reg(78, reg_78)
def set_reg_79(self):
      reg_79 = (
      self.Qg180degDelay_0to4_2 << 3 )
      self.send_reg(79, reg_79)
def set_reg_80(self):
      reg_80 = (
      self.Qg270degDelay_0to4_2 << 3 )
      self.send_reg(80, reg_80)
def set_reg_81(self):
      reg_81 = (
      self.DischargeTap16_3to0 << 4 |
      self.ChargeTap16_3to0 << 0 )
      self.send_reg(81, reg_81)
def set_reg_82(self):
      reg_82 = (
      self.DischargeTapn_3to0 << 4 |
      self.ChargeTapn16_3to0 << 0 )
      self.send_reg(82, reg_82)
def set_reg_83(self):
      reg_83 = (
      self.X1sel_32to39_2 << 0 )
      self.send_reg(83, reg_83)
def set_reg_84(self):
      reg_84 = (
      self.X1sel_40to47_2 << 0 )
      self.send_reg(84, reg_84)
def set_reg_85(self):
      reg_85 = (
      self.X2sel_32to36_2 << 3 )
      self.send_reg(85, reg_85)
def set_reg_86(self):
      reg_86 = (
      self.X2sel_37to41_2 << 3 )
      self.send_reg(86, reg_86)
def set_reg_87(self):
      reg_87 = (
      self.X4sel_32to36_2 << 3 )
      self.send_reg(87, reg_87)
def set_reg_88(self):
      reg_88 = (
      self.X4sel_37to41_2 << 3 )
      self.send_reg(88, reg_88)
def set_reg_89(self):
      reg_89 = (
      self.X8sel_32to36_2 << 3 )
      self.send_reg(89, reg_89)
def set_reg_90(self):
      reg_90 = (
      self.X8sel_41_2 << 7 |
      self.X8sel_37to40_2 << 3 )
      self.send_reg(90, reg_90)
def set_reg_91(self):
      reg_91 = (
      self.qutx_fb_180Cal_en << 7 |
```

```
      self.qutx_fb_0Cal_en << 6 |
      self.qutx_fb_180Rsff_en << 5 |
      self.qutx_fb_0Rsff_en << 4 )
      self.send_reg(91, reg_91)
def set_reg_96(self):
      reg_96 = (
      self.N << 4 |
      self.R_11to8 << 0 )
      self.send_reg(96, reg_96)
def set_reg_97(self):
      reg_97 = (
      self.R_7to0 << 0 )
      self.send_reg(97, reg_97)
def set_reg_98(self):
      reg_98 = (
      self.Asyncrst_n << 7 |
      self.Cp_sel_6to0 << 0 )
      self.send_reg(98, reg_98)
def set_reg_99(self):
      reg_99 = (
      self.Cp_sel_8to7 << 6 |
      self.ForceFout << 5 |
      self.ForceFoutb << 4 |
      self.Out_en << 3 |
      self.Dll_en << 2 |
      self.Ana_en << 1 )
      self.send_reg(99, reg_99)
def read_reg_100(self):
      reg_100 = self.get_reg(100)
      self.Decod_in_0deg = reg_100 >> 3

def set_reg_104(self):
      reg_104 = (
      self.Ngt3_3 << 7 |
      self.NorNdiv4_3 << 0 )
      self.send_reg(104, reg_104)
def set_reg_105(self):
      reg_105 = (
      self.RorFrNpRdiv4_25to18_3 << 0 )
      self.send_reg(105, reg_105)
def set_reg_106(self):
      reg_106 = (
      self.RorFrNpRdiv4_17to10_3 << 0 )
      self.send_reg(106, reg_106)
def set_reg_107(self):
      reg_107 = (
      self.RorFrNpRdiv4_9to2_3 << 0 )
      self.send_reg(107, reg_107)
def set_reg_108(self):
      reg_108 = (
      self.RorFrNpRdiv4_1to0_3 << 6 )
      self.send_reg(108, reg_108)
def set_reg_109(self):
      reg_109 = (
      self.Qu_tx_Ngt3_3 << 7 |
      self.NorNdiv4_phsh_3 << 0 )
      self.send_reg(109, reg_109)
```

```
def set_reg_110(self):
      reg_110 = (
      self.RorFrNpRdiv4_phsh_25to18_3 << 0 )
      self.send_reg(110, reg_110)
def set_reg_111(self):
      reg_111 = (
      self.RorFrNpRdiv4_phsh_17to10_3 << 0 )
      self.send_reg(111, reg_111)
def set_reg_112(self):
      reg_112 = (
      self.RorFrNpRdiv4_phsh_9to2_3 << 0 )
      self.send_reg(112, reg_112)
def set_reg_113(self):
      reg_113 = (
      self.RorFrNpRdiv4_phsh_1to0_3 << 6 )
      self.send_reg(113, reg_113)
def set_reg_114(self):
      reg_114 = (
      self.Passthru_ref_clk_3 << 7 |
      self.Byp_ram_3 << 6 |
      self.Dis_adr_dith_3 << 5 |
      self.Dis_p5G_dith_3 << 4 |
      self.Byp_fine_3 << 3 |
      self.Exclude32_3 << 2 |
      self.Dis_risedge_3 << 1 |
      self.Dis_faledge_3 << 0 )
      self.send_reg(114, reg_114)
def set_reg_116(self):
      reg_116 = (
      self.Spr_puls_en_3 << 7 |
      self.Spr_puls_val_a_9to3_3 << 0 )
      self.send_reg(116, reg_116)
def set_reg_117(self):
      reg_117 = (
      self.Spr_pulse_val_2to0_3 << 5 )
      self.send_reg(117, reg_117)
def set_reg_118(self):
      reg_118 = (
      self.Spr_puls_val_b_9to2_3 << 0 )
      self.send_reg(118, reg_118)
def set_reg_119(self):
      reg_119 = (
      self.Spr_puls_val_b_1to0_3 << 6 )
      self.send_reg(119, reg_119)
def set_reg_120(self):
      reg_120 = (
      self.Thru_ris_en_3 << 7 |
      self.Thru_ris_tap_11to6_3 << 1 )
      self.send_reg(120, reg_120)
def set_reg_121(self):
      reg_121 = (
      self.Thru_ris_tap_5to0_3 << 2 )
      self.send_reg(121, reg_121)
def set_reg_122(self):
      reg_122 = (
      self.Thru_fal_en_3 << 7 |
      self.Thru_fal_tap_11to6_3 << 1 )
```

```
      self.send_reg(122, reg_122)
def set_reg_123(self):
      reg_123 = (
      self.Thru_fal_tap_5to0_3 << 2 )
      self.send_reg(123, reg_123)
def set_reg_124(self):
      reg_124 = (
      self.Dig_delay_3 << 7 |
      self.Clk_driver_en_3 << 6 |
      self.qu_reg_en_3 << 5 |
      self.qq_reg_en_3 << 4 |
      self.win_rst_3 << 3 |
      self.fineEn_3 << 2 |
      self.fineEnb_3 << 1 |
      self.rsffEn_3 << 0 )
      self.send_reg(124, reg_124)
def set_reg_125(self):
      reg_125 = (
      self.dl_en_3 << 7 |
      self.cp_en_3 << 6 |
      self.forceCpUpb_3 << 5 |
      self.forceCpDn_3 << 4 |
      self.pdUpTune_1to0_3 << 2 |
      self.pdDnTune_1to0_3 << 0 )
      self.send_reg(125, reg_125)
def set_reg_126(self):
      reg_126 = (
      self.cpUpTune_2to0_3 << 5 |
      self.cpDnTune_2to0_3 << 2 |
      self.pdEn_3 << 1 )
      self.send_reg(126, reg_126)
def set_reg_127(self):
      reg_127 = (
      self.digClkPhase_7to0_3 << 0 )
      self.send_reg(127, reg_127)
def set_reg_128(self):
      reg_128 = (
      self.Rst_n_async_3 << 7 )
      self.send_reg(128, reg_128)
def read_reg_129(self):
      reg_129 = self.get_reg(129)
      self.L1_lup00_15to8_3 = reg_129

def read_reg_130(self):
      reg_130 = self.get_reg(130)
      self.L1_lup90_15to8_3 = reg_130

def read_reg_131(self):
      reg_131 = self.get_reg(131)
      self.Merg_ris_fin_3 = reg_131 >> 2

def read_reg_132(self):
      reg_132 = self.get_reg(132)
      self.Merg_fal_fin_3 = reg_132 >> 2

def set_reg_133(self):
      reg_133 = (
```

```
      self.Qg00degDelay_0to4_3 << 3 )
      self.send_reg(133, reg_133)
def set_reg_134(self):
      reg_134 = (
      self.Qg90degDelay_0to4_3 << 3 )
      self.send_reg(134, reg_134)
def set_reg_135(self):
      reg_135 = (
      self.Qg180degDelay_0to4_3 << 3 )
      self.send_reg(135, reg_135)
def set_reg_136(self):
      reg_136 = (
      self.Qg270degDelay_0to4_3 << 3 )
      self.send_reg(136, reg_136)
def set_reg_137(self):
      reg_137 = (
      self.DischargeTap16_0to3_3 << 4 |
      self.ChargeTap16_0to3_3 << 0 )
      self.send_reg(137, reg_137)
def set_reg_138(self):
      reg_138 = (
      self.DischargeTapn_0to3_3 << 4 |
      self.ChargeTapn16_0to3_3 << 0 )
      self.send_reg(138, reg_138)
def set_reg_139(self):
      reg_139 = (
      self.X1sel_32to39_3 << 0 )
      self.send_reg(139, reg_139)
def set_reg_140(self):
      reg_140 = (
      self.X1sel_40to47_3 << 0 )
      self.send_reg(140, reg_140)
def set_reg_141(self):
      reg_141 = (
      self.X2sel_32to36_3 << 3 )
      self.send_reg(141, reg_141)
def set_reg_142(self):
      reg_142 = (
      self.X2sel_37to41_3 << 3 )
      self.send_reg(142, reg_142)
def set_reg_143(self):
      reg_143 = (
      self.X4sel_32to36_3 << 3 )
      self.send_reg(143, reg_143)
def set_reg_144(self):
      reg_144 = (
      self.X4sel_37to41_3 << 3 )
      self.send_reg(144, reg_144)
def set_reg_145(self):
      reg_145 = (
      self.X8sel_32to36_3 << 3 )
      self.send_reg(145, reg_145)
def set_reg_146(self):
      reg_146 = (
      self.X8sel_41_3 << 7 |
      self.X8sel_37to40_3 << 3 )
      self.send_reg(146, reg_146)
```

```
def set_reg_147(self):
      reg_147 = (
      self.qurx_180Cal_en << 7 |
      self.qurx_0Cal_en << 6 )
      self.send_reg(147, reg_147)
def set_reg_152(self):
      reg_152 = (
      self.extClkEn << 7 |
      self.extClkEnBNOTD7 << 6 |
      self.div2_rst << 5 |
      self.TxChClkSel << 3 )
      self.send_reg(152, reg_152)
def set_reg_153(self):
      reg_153 = (
      self.TxChClkEn << 5 )
      self.send_reg(153, reg_153)
def set_reg_156(self):
      reg_156 = (
      self.tx_bb_en << 7 |
      self.tx_bb_fdbk_bw << 5 |
      self.tx_bb_fdbk_cal_en << 4 |
      self.tx_bb_fdbk_cart_err_en << 3 |
      self.tx_bb_fdbk_cart_fb_en << 2 |
      self.tx_bb_fdbk_cart_fwd_en << 1 )
      self.send_reg(156, reg_156)
def set_reg_157(self):
      reg_157 = (
      self.tx_bb_fdbk_en << 6 |
      self.tx_bb_fdbk_1q_sel << 5 |
      self.tx_bb_fdbk_lp << 2 )
      self.send_reg(157, reg_157)
def set_reg_158(self):
      reg_158 = (
      self.tx_bb_fdbk_statt << 5 |
      self.tx_bb_fdbk_swapi << 4 |
      self.tx_bb_fdbk_swapq << 3 |
      self.tx_bb_gain_cmp << 2 )
      self.send_reg(158, reg_158)
def set_reg_159(self):
      reg_159 = (
      self.tx_bb_lp << 5 |
      self.tx_bb_swapi << 4 |
      self.tx_bb_swapq << 3 |
      self.tx_butt_bw << 0 )
      self.send_reg(159, reg_159)
def set_reg_160(self):
      reg_160 = (
      self.tx_bw_trck << 4 |
      self.tx_cart_en << 3 )
      self.send_reg(160, reg_160)
def set_reg_161(self):
      reg_161 = (
      self.tx_cart_fb_bb_statt << 3 )
      self.send_reg(161, reg_161)
def set_reg_162(self):
      reg_162 = (
      self.tx_cart_fb_dcoc_dac_I1 << 2 )
```

```
      self.send_reg(162, reg_162)
def set_reg_163(self):
      reg_163 = (
      self.tx_cart_fb_dcoc_dac_I2 << 2 )
      self.send_reg(163, reg_163)
def set_reg_164(self):
      reg_164 = (
      self.tx_cart_fb_dcoc_dac_Q1 << 2 )
      self.send_reg(164, reg_164)
def set_reg_165(self):
      reg_165 = (
      self.tx_cart_fb_dcoc_dac_Q2 << 2 )
      self.send_reg(165, reg_165)
def set_reg_166(self):
      reg_166 = (
      self.CartesianFeedbackpathDCOCenable << 7 |
      self.CartesianFeedbackpathenable << 6 |
      self.CartesianFBpathHiResolutionDCOCenable << 5 |
      self.CartesianFBpathBW << 1 )
      self.send_reg(166, reg_166)
def set_reg_167(self):
      reg_167 = (
      self.CartesianFBRFGain << 2 )
      self.send_reg(167, reg_167)
def set_reg_168(self):
      reg_168 = (
      self.CartesianFBpathSwapIandIx << 7 |
      self.CartesianFBpathSwapQandQx << 6 |
      self.CartesianFBpathSwitchtoforwardSummer << 5 |
      self.tx_cart_fb_lo_select << 0 )
      self.send_reg(168, reg_168)
def set_reg_169(self):
      reg_169 = (
      self.CartesianFBpathAmp1Gain << 6 |
      self.CartesianFBpathAmp2Gain << 4 |
      self.CartesianFBpathAmp3Gain << 2 |
      self.CartesianFBpathAmp4Gain << 0 )
      self.send_reg(169, reg_169)
def set_reg_170(self):
      reg_170 = (
      self.CartesianFBpathAmpCurrentSelect << 5 |
      self.CartesianFBpathZeroEnable << 4 |
      self.tx_cart_zero_statt << 0 )
      self.send_reg(170, reg_170)
def set_reg_171(self):
      reg_171 = (
      self.tx_inbuf_bw << 6 |
      self.tx_inbuf_statt << 3 )
      self.send_reg(171, reg_171)
def set_reg_172(self):
      reg_172 = (
      self.tx_output_channel_sel << 5 )
      self.send_reg(172, reg_172)
def set_reg_173(self):
      reg_173 = (
      self.tx_p1_bw << 4 |
      self.tx_pw_bw1 << 2 )
```

```
      self.send_reg(173, reg_173)
def set_reg_174(self):
      reg_174 = (
      self.tx_p2_bw2 << 4 |
      self.PushPullBufferCurrent << 1 )
      self.send_reg(174, reg_174)
def set_reg_175(self):
      reg_175 = (
      self.tx_rf_aoc_bw << 6 |
      self.RFForwardPathEnable_toMUX << 5 |
      self.RFForwardPathEnable_ExternalPinenable << 4 |
      self.tx_rf_fwd_lp << 1 )
      self.send_reg(175, reg_175)
def set_reg_176(self):
      reg_176 = (
      self.tx_rf_fwd_statt1 << 5 |
      self.tx_rf_fwd_statt2 << 2 )
      self.send_reg(176, reg_176)
def set_reg_177(self):
      reg_177 = (
      self.BBQDivideby2or4Select << 7 |
      self.BBQQuadGenEnable << 6 |
      self.BBQPolyphaseQuadGenEnable << 5 )
      self.send_reg(177, reg_177)
def set_reg_178(self):
      reg_178 = (
      self.lofb_tun_s << 4 |
      self.lofb_tun_sx << 0 )
      self.send_reg(178, reg_178)
def set_reg_179(self):
      reg_179 = (
      self.lofw_tun_s2 << 4 |
      self.lofw_tun_sx2 << 0 )
      self.send_reg(179, reg_179)
def set_reg_180(self):
      reg_180 = (
      self.reserve_tx26 << 0 )
      self.send_reg(180, reg_180)
def set_reg_181(self):
      reg_181 = (
      self.reserve_tx27 << 0 )
      self.send_reg(181, reg_181)
def set_reg_192(self):
      reg_192 = (
      self.rx_Idac << 3 |
      self.rx_dcs << 1 |
      self.rx_den << 0 )
      self.send_reg(192, reg_192)
def set_reg_193(self):
      reg_193 = (
      self.rx_Qdac << 3 |
      self.rx_cmpen << 1 |
      self.rx_dcoc << 0 )
      self.send_reg(193, reg_193)
def set_reg_194(self):
      reg_194 = (
      self.rx_ten << 7 |
```

```
      self.rx_ren << 6 |
      self.rx_dven << 4 |
      self.rx_dv << 0 )
      self.send_reg(194, reg_194)
def set_reg_195(self):
      reg_195 = (
      self.rx_extc << 7 |
      self.rx_cen << 4 |
      self.rx_chck << 2 |
      self.rx_chcken << 1 |
      self.rx_fen << 0 )
      self.send_reg(195, reg_195)
def set_reg_196(self):
      reg_196 = (
      self.rx_onchen << 7 |
      self.rx_offchen << 6 |
      self.rx_foe << 0 )
      self.send_reg(196, reg_196)
def set_reg_197(self):
      reg_197 = (
      self.rx_offch << 5 |
      self.rx_onchf << 3 |
      self.rx_onchc << 1 )
      self.send_reg(197, reg_197)
def set_reg_198(self):
      reg_198 = (
      self.rx_qs << 5 |
      self.rx_bqg << 3 |
      self.rx_rq << 0 )
      self.send_reg(198, reg_198)
def set_reg_199(self):
      reg_199 = (
      self.rx_rv << 5 |
      self.rx_rip << 2 |
      self.rx_rfp << 0 )
      self.send_reg(199, reg_199)
def set_reg_200(self):
      reg_200 = (
      self.rx_cp_12to8 << 3 |
      self.rx_gs << 0 )
      self.send_reg(200, reg_200)
def set_reg_201(self):
      reg_201 = (
      self.rx_cp_7to0 << 0 )
      self.send_reg(201, reg_201)
def set_reg_202(self):
      reg_202 = (
      self.rx_cv_10to3 << 0 )
      self.send_reg(202, reg_202)
def set_reg_203(self):
      reg_203 = (
      self.rx_cv_2to0 << 5 |
      self.rx_cc_2to0 << 2 |
      self.rx_cq_9to8 << 0 )
      self.send_reg(203, reg_203)
def set_reg_204(self):
      reg_204 = (
```

```
            self.rx_cq_7to0 << 0 )
            self.send_reg(204, reg_204)
      def set_reg_205(self):
            reg_205 = (
            self.rx_lna << 5 |
            self.rx_lnab << 3 |
            self.rx_rxchen << 2 |
            self.rx_bbq_div2or4 << 1 |
            self.rx_Loselect << 0 )
            self.send_reg(205, reg_205)
      def set_reg_206(self):
            reg_206 = (
            self.poly_en << 7 )
            self.send_reg(206, reg_206)
      def set_reg_207(self):
            reg_207 = (
            self.lorx_tun_s << 4 |
            self.lorx_tun_sx << 0 )
            self.send_reg(207, reg_207)
      def read_reg_208(self):
            reg_208 = self.get_reg(208)
            self.rx_Icmpo = reg_208 >> 5
            self.rx_Iodac = reg_208 % 64

      def read_reg_209(self):
            reg_209 = self.get_reg(209)
            self.rx_Qcmpo = reg_209 >> 5
            self.rx_Qodac = reg_209 % 64

      def read_reg_210(self):
            reg_210 = self.get_reg(210)
            self.rx_rc = reg_210

      def set_reg_220(self):
            reg_220 = (
            self.shft_cml_in << 7 |
            self.vagenable1 << 6 |
            self.vagenable2 << 5 )
            self.send_reg(220, reg_220)
      def set_reg_222(self):
            reg_222 = (
            self.TestMuxBufferEnable << 7 |
            self.TestMuxEnable << 6 |
            self.TestMuxSetting << 0 )
            self.send_reg(222, reg_222)

      #The SPI format is 8 bits, plus a two-byte header
      # The format is:
      # Byte sent on MOSI     Bit   Description
      # ---------------------------------------------------------------------
---------
      # 1               7     Not W - Read/write indicator, where 0 indicates
      #                       a write and 1 indicates a read
      #                 6-0   Upper 7 bits of the register address
      # ---------------------------------------------------------------------
---------
      # 2               7-1   Lower 7 bits of the register address
```

```
      #                 0     If 1, will disable the auto-increment of the
      #                       register address.
      # ------------------------------------------------------------------
---------
      # 3, ..., n+3           7-0   Optionally n words of write data byte
      #
      # Byte sent on MISO     Bit   Description
      # ------------------------------------------------------------------
---------
      # 1               7-0   Read data returned that was read during
      #                       the last transfer
      # ------------------------------------------------------------------
---------
      # 2               7-0   0s will be forced
      # ------------------------------------------------------------------
---------
      # 3, ..., n+3           7-0   Optionally n words of read data byte

      #Send register read to SPI, get result
      #         _read_spi()
      #Type           Sub Function
      #Description    Read data from SPI bus peripheral.
      #            Return the data read if successful, else a zero length
string.
      #Usage          usrp.source_x._read_spi(optional_header, enables,
format, len)
      #            optional_header : 0,1 or 2 bytes to write before buf.
      #Parameters
      #            enables : bitmask of peripherals to write.
      #             format : transaction format. SPI_FMT_*
      #            len : number of bytes to read.#

      #Write register to SPI
      #Type        Sub Function
      #Description Write data to SPI bus peripheral.
      #         SPI == "Serial Port Interface". SPI is a 3 wire bus plus a
separate enable for each
      #         peripheral. The common lines are SCLK,SDI and SDO. The FX2
always drives SCLK
      #         and SDI, the clock and data lines from the FX2 to the
peripheral. When enabled, a
      #         peripheral may drive SDO, the data line from the peripheral
to the FX2.
      #         The SPI_READ and SPI_WRITE commands are formatted
identically.
      #         Each specifies which peripherals to enable, whether the bits
should be transmistted Most
      #         Significant Bit first or Least Significant Bit first, the
number of bytes in the optional
      #         header, and the number of bytes to read or write in the body.
      #         The body is limited to 64 bytes. The optional header may
contain 0, 1 or 2 bytes. For an
      #         SPI_WRITE, the header bytes are transmitted to the peripheral
followed by the the body
      #         bytes. For an SPI_READ, the header bytes are transmitted to
the peripheral, then len
```

```
      #         bytes are read back from the peripheral.(see :
usrp_spi_defs.h file). If format specifies
      #         that optional_header bytes are present, they are written to
the peripheral immediately
      #         prior to writing buf.
      #         Return true if successful. Writes are limited to a maximum of
64 bytes.
      #Usage       usrp.source_x._write_spi(optional_header, enables, format,
buf)
      #Parameters  optional_header: 0,1 or 2 bytes to write before buf.
      #         enables: bitmask of peripherals to write.
      #          format: transaction format. SPI_FMT_*
      #         buf : the data to write#

      def send_reg(self, regnum, dat):
            #Send 16 bit header over SPI to send register number
            #Write 8 bit register
            #Set first byte of header
            #hdr_hi = int( (regnum >> 7) & 0x7f)
            #Set second byte of header
            #hdr_lo = int( (regnum << 1) & 0xff)
            #Set full two-byte header
            #hdr = ((hdr_hi << 8) + hdr_lo) & 0x7fff

            hdr = int( (regnum << 1) & 0x7ffe)

            #Set byte of write data
            s = chr(dat & 0xff)
            #Send data over SPI
            self.u._write_spi(hdr, self.spi_enable, self.spi_format, s)
            print 'RFIC4: Writing register %d with %d' % (regnum, dat)

      def get_reg(self, regnum):
            #Send 16 bit header over SPI to send register number
            #Read 8 bit register
            #Set first byte of header
            #hdr_hi = chr( ( (regnum >> 7) + (1 << 7) ) & 0xff)
            #Set second byte of header
            #hdr_lo = chr( (regnum << 1) & 0xff)
            #Set full two-byte header
            #hdr = ((hdr_hi << 8) + hdr_lo) & 0xffff

            #Send data over SPI, get register contents
            #r = self.u._read_spi(hdr, self.spi_enable, self.spi_format, 1)

            # First set register zero, to set the SPI register number to
zero, then get all registers, then return desired register as integer

            # Get data to set register zero
            dat = self.Ngt3 << 7 | self.NorNdiv4 << 0

            r = self.u._write_spi(0, self.spi_enable, self.spi_format,
chr(dat & 0xff))

            # Get all registers, no header required
            read = self.u._read_spi(0, self.spi_enable,
self.spi_format_no_header, 64)
```

```
            read = read + self.u._read_spi(0, self.spi_enable,
self.spi_format_no_header, 64)
            read = read + self.u._read_spi(0, self.spi_enable,
self.spi_format_no_header, 64)
            read = read + self.u._read_spi(0, self.spi_enable,
self.spi_format_no_header, 64)
            read = read + self.u._read_spi(0, self.spi_enable,
self.spi_format_no_header, 64)

            # Return desired register as integer
            r = ord(read[regnum])

            print 'RFIC4: Reading register %d' % (regnum)
            return r


      # ----------------------------------------------------------------
      # These methods control the GPIO bus.  Since the board has to access
      # both the io_rx_* and io_tx_* pins, we define our own methods to do
so.
      # This bypasses any code in db_base.
      #
      # The board operates in ATR mode, always.  Thus, when the board is
first
      # initialized, it is in receive mode, until bits show up in the TX
FIFO.
      #
      def rx_write_oe(self, value, mask):
            return self.u._write_fpga_reg((FR_OE_1, FR_OE_3)[self.which],
gru.hexint((mask << 16) | value))

      def rx_write_io(self, value, mask):
            return self.u._write_fpga_reg((FR_IO_1, FR_IO_3)[self.which],
gru.hexint((mask << 16) | value))

      def rx_read_io(self):
            t = self.u._read_fpga_reg((FR_RB_IO_RX_A_IO_TX_A,
FR_RB_IO_RX_B_IO_TX_B)[self.which])
            return (t >> 16) & 0xffff

      def rx_set_atr_mask(self, v):
            #print 'Set mask to %s' % (v)
            return
self.u._write_fpga_reg((FR_ATR_MASK_1,FR_ATR_MASK_3)[self.which],
gru.hexint(v))

      def rx_set_atr_txval(self, v):
            #print 'Set TX value to %s' % (v)
            return
self.u._write_fpga_reg((FR_ATR_TXVAL_1,FR_ATR_TXVAL_3)[self.which],
gru.hexint(v))

      def rx_set_atr_rxval(self, v):
            #print 'Set RX value to %s' % (v)
            return
self.u._write_fpga_reg((FR_ATR_RXVAL_1,FR_ATR_RXVAL_3)[self.which],
gru.hexint(v))
```

```
      # ----------------------------------------------------------------------
      # These methods set control the high-level operating parameters.

      def set_rx_gain(self, gain):
            # Set RX gain
            # @param gain: gain in dB
            # Four parameters: self.rx_bqg, self.rx_dcs, self.rx_gs,
self.rx_rip
            # 1 to 39 dB of gain (0 to 38)
            # Not all steps available
            if gain < 0.0: gain = 0.0
            if gain > 38.0: gain = 38.0

            if gain <= 3:
                  self.rx_bqg = 3   #reg 198
                  self.rx_dcs = 0   #reg 192
                  self.rx_gs = 4    #reg 200
                  self.rx_rip = 4   #reg 199

            elif gain >= 3 and gain < 4:
                  self.rx_bqg = 3
                  self.rx_dcs = 0
                  self.rx_gs = 4
                  self.rx_rip = 3

            elif gain >= 4 and gain < 5:
                  self.rx_bqg = 3
                  self.rx_dcs = 0
                  self.rx_gs = 2
                  self.rx_rip = 4

            elif gain >=5  and gain < 6:
                  self.rx_bqg = 3
                  self.rx_dcs = 0
                  self.rx_gs = 3
                  self.rx_rip = 3

            elif gain >= 6 and gain < 7:
                  self.rx_bqg = 3
                  self.rx_dcs = 0
                  self.rx_gs = 4
                  self.rx_rip = 2

            elif gain >= 7 and gain < 8:
                  self.rx_bqg = 3
                  self.rx_dcs = 0
                  self.rx_gs = 2
                  self.rx_rip = 3

            elif gain >= 8 and gain < 9:
                  self.rx_bqg = 3
                  self.rx_dcs = 0
                  self.rx_gs = 3
                  self.rx_rip = 2

            elif gain >= 9 and gain < 10:
```

```
            self.rx_bqg = 3
            self.rx_dcs = 0
            self.rx_gs = 1
            self.rx_rip = 3

        elif gain >= 10 and gain < 11:
            self.rx_bqg = 3
            self.rx_dcs = 0
            self.rx_gs = 2
            self.rx_rip = 2

        elif gain >= 11 and gain < 12:
            self.rx_bqg = 3
            self.rx_dcs = 0
            self.rx_gs = 0
            self.rx_rip = 3

        elif gain >= 12 and gain < 13:
            self.rx_bqg = 2
            self.rx_dcs = 0
            self.rx_gs = 4
            self.rx_rip = 2

        elif gain >= 13 and gain < 14:
            self.rx_bqg = 3
            self.rx_dcs = 0
            self.rx_gs = 2
            self.rx_rip = 1

        elif gain >= 14 and gain < 15:
            self.rx_bqg = 3
            self.rx_dcs = 0
            self.rx_gs = 0
            self.rx_rip = 2

        elif gain >= 15 and gain < 16:
            self.rx_bqg = 2
            self.rx_dcs = 0
            self.rx_gs = 1
            self.rx_rip = 3

        elif gain >= 16 and gain < 17:
            self.rx_bqg = 2
            self.rx_dcs = 0
            self.rx_gs = 2
            self.rx_rip = 2

        elif gain >= 17 and gain < 18:
            self.rx_bqg = 3
            self.rx_dcs = 0
            self.rx_gs = 0
            self.rx_rip = 2

        elif gain >= 18 and gain < 19:
            self.rx_bqg = 3
            self.rx_dcs = 0
            self.rx_gs = 1
```

```
      self.rx_rip = 0

elif gain >= 19 and gain < 20:
      self.rx_bqg = 2
      self.rx_dcs = 0
      self.rx_gs = 2
      self.rx_rip = 1

elif gain >= 20 and gain < 21:
      self.rx_bqg = 3
      self.rx_dcs = 0
      self.rx_gs = 0
      self.rx_rip = 0

elif gain >= 21 and gain < 22:
      self.rx_bqg = 2
      self.rx_dcs = 0
      self.rx_gs = 1
      self.rx_rip = 1

elif gain >= 22 and gain < 23:
      self.rx_bqg = 1
      self.rx_dcs = 1
      self.rx_gs = 2
      self.rx_rip = 2

elif gain >= 23 and gain < 24:
      self.rx_bqg = 2
      self.rx_dcs = 0
      self.rx_gs = 0
      self.rx_rip = 1

elif gain >= 24 and gain < 25:
      self.rx_bqg = 1
      self.rx_dcs = 1
      self.rx_gs = 1
      self.rx_rip = 2

elif gain >= 25 and gain < 26:
      self.rx_bqg = 1
      self.rx_dcs = 1
      self.rx_gs = 2
      self.rx_rip = 1

elif gain >= 26 and gain < 27:
      self.rx_bqg = 1
      self.rx_dcs = 1
      self.rx_gs = 3
      self.rx_rip = 0

elif gain >= 27 and gain < 28:
      self.rx_bqg = 1
      self.rx_dcs = 1
      self.rx_gs = 1
      self.rx_rip = 1

elif gain >= 28 and gain < 29:
```

```
            self.rx_bqg = 1
            self.rx_dcs = 1
            self.rx_gs = 2
            self.rx_rip = 0

        elif gain >= 29 and gain < 30:
            self.rx_bqg = 1
            self.rx_dcs = 1
            self.rx_gs = 0
            self.rx_rip = 1

        elif gain >= 30 and gain < 31:
            self.rx_bqg = 1
            self.rx_dcs = 1
            self.rx_gs = 1
            self.rx_rip = 0

        elif gain >= 31 and gain < 32:
            self.rx_bqg = 0
            self.rx_dcs = 3
            self.rx_gs = 2
            self.rx_rip = 1

        elif gain >= 32 and gain < 33:
            self.rx_bqg = 1
            self.rx_dcs = 1
            self.rx_gs = 0
            self.rx_rip = 0

        elif gain >= 33 and gain < 34:
            self.rx_bqg = 0
            self.rx_dcs = 3
            self.rx_gs = 1
            self.rx_rip = 1

        elif gain >= 34 and gain < 35:
            self.rx_bqg = 0
            self.rx_dcs = 3
            self.rx_gs = 2
            self.rx_rip = 0

        elif gain >= 35 and gain < 36:
            self.rx_bqg = 0
            self.rx_dcs = 3
            self.rx_gs = 0
            self.rx_rip = 1

        elif gain >= 36 and gain < 38:
            self.rx_bqg = 0
            self.rx_dcs = 3
            self.rx_gs = 1
            self.rx_rip = 0

        elif gain >= 38:
            self.rx_bqg = 0
            self.rx_dcs = 3
            self.rx_gs = 0
```

```
            self.rx_rip = 0

      self.set_reg_198()
      self.set_reg_192()
      self.set_reg_200()
      self.set_reg_199()


def set_tx_gain(self, gain):
      # Set TX gain
      # @param gain: output gain in dB
      # Two parameters:
      # self.tx_rf_fwd_statt1, self.tx_rf_fwd_statt2
      # (45 dB of range)
      # 5 dB steps
      if gain < 0.0: gain = 0.0
      if gain > 45.0: gain = 45.0

      if gain <= 2.5:
            self.tx_rf_fwd_statt1 = 7
            self.tx_rf_fwd_statt2 = 7

      elif gain > 2.5 and gain <= 7.5:
            self.tx_rf_fwd_statt1 = 3
            self.tx_rf_fwd_statt2 = 7

      elif gain > 7.5 and gain <= 12.5:
            self.tx_rf_fwd_statt1 = 1
            self.tx_rf_fwd_statt2 = 7

      elif gain > 12.5 and gain <= 17.5:
            self.tx_rf_fwd_statt1 = 3
            self.tx_rf_fwd_statt2 = 3

      elif gain > 17.5 and gain <= 22.5:
            self.tx_rf_fwd_statt1 = 1
            self.tx_rf_fwd_statt2 = 3

      elif gain > 22.5 and gain <= 27.5:
            self.tx_rf_fwd_statt1 = 0
            self.tx_rf_fwd_statt2 = 3

      elif gain > 27.5 and gain <= 32.5:
            self.tx_rf_fwd_statt1 = 1
            self.tx_rf_fwd_statt2 = 1

      elif gain > 32.5 and gain <= 37.5:
            self.tx_rf_fwd_statt1 = 0
            self.tx_rf_fwd_statt2 = 1

      elif gain > 37.5 and gain <= 42.5:
            self.tx_rf_fwd_statt1 = 1
            self.tx_rf_fwd_statt2 = 0

      elif gain > 42.5:
            self.tx_rf_fwd_statt1 = 0
            self.tx_rf_fwd_statt2 = 0
```

```
      self.set_reg_176()

def set_fb_gain(self, gain):
      # Set Feedback path gain
      # @param gain: output gain in dB
      #  parameters:
      # self.CartesianFBpathAmp1Gain, self.CartesianFBpathAmp2Gain,
      # self.CartesianFBpathAmp3Gain, self.CartesianFBpathAmp4Gain
      # (40 dB of range)
      # 5 dB steps
      # FIXME
      if gain < 0.0: gain = 0.0
      if gain > 40.0: gain = 40.0

      if gain <= 2.5:
            self.CartesianFBpathAmp1Gain = 3
            self.CartesianFBpathAmp2Gain = 3
            self.CartesianFBpathAmp3Gain = 3
            self.CartesianFBpathAmp4Gain = 3

      elif gain > 2.5 and gain <= 7.5:
            self.CartesianFBpathAmp1Gain = 3
            self.CartesianFBpathAmp2Gain = 3
            self.CartesianFBpathAmp3Gain = 3
            self.CartesianFBpathAmp4Gain = 1

      elif gain > 7.5 and gain <= 12.5:
            self.CartesianFBpathAmp1Gain = 3
            self.CartesianFBpathAmp2Gain = 3
            self.CartesianFBpathAmp3Gain = 1
            self.CartesianFBpathAmp4Gain = 1

      elif gain > 12.5 and gain <= 17.5:
            self.CartesianFBpathAmp1Gain = 3
            self.CartesianFBpathAmp2Gain = 1
            self.CartesianFBpathAmp3Gain = 1
            self.CartesianFBpathAmp4Gain = 1

      elif gain > 17.5 and gain <= 22.5:
            self.CartesianFBpathAmp1Gain = 1
            self.CartesianFBpathAmp2Gain = 1
            self.CartesianFBpathAmp3Gain = 1
            self.CartesianFBpathAmp4Gain = 1

      elif gain > 22.5 and gain <= 27.5:
            self.CartesianFBpathAmp1Gain = 1
            self.CartesianFBpathAmp2Gain = 1
            self.CartesianFBpathAmp3Gain = 1
            self.CartesianFBpathAmp4Gain = 0

      elif gain > 27.5 and gain <= 32.5:
            self.CartesianFBpathAmp1Gain = 1
            self.CartesianFBpathAmp2Gain = 1
            self.CartesianFBpathAmp3Gain = 0
            self.CartesianFBpathAmp4Gain = 0
```

```
            elif gain > 32.5 and gain <= 37.5:
                  self.CartesianFBpathAmp1Gain = 1
                  self.CartesianFBpathAmp2Gain = 0
                  self.CartesianFBpathAmp3Gain = 0
                  self.CartesianFBpathAmp4Gain = 0

            elif gain > 37.5:
                  self.CartesianFBpathAmp1Gain = 0
                  self.CartesianFBpathAmp2Gain = 0
                  self.CartesianFBpathAmp3Gain = 0
                  self.CartesianFBpathAmp4Gain = 0

            self.set_reg_169()

      def calc_freq_vars(self, Fclk, Fout):
            #
            #@param Fclk: Clock frequency of board (Hz)
            #@type Fclk: float
            #@param Fout: Desired clock frequency for one of three frequency
synthesizers (Hz)
            #@type Fout: float
            #
            # Calculate RFIC register variables to set frequency of frequency
synthesizers
            # data1 corresponds to Ngt, D7, a single bit
            # data2 corresponds to NorNdiv4, D6-D0, up to seven bits
            # data3 corresponds to RorFrNpRdiv4, up to 26 bits
            # D7-D0, D7-D0, D7-D0, D7-D6
            # Returns Ngt, NorNdiv4, RorFrNpRdiv4_25to18,
RorFrNpRdiv4_17to10,
            # RorFrNpRdiv4_9to2, RorFrNpRdiv4_1to0

            if (Fout > Fclk / 4):
                  NpR = (2 ** -26) * int(2 ** 26 * Fclk / Fout)
                  data1 = 0;
                  data2 = int(floor(NpR))
                  data3 = int(2 ** 26 * (NpR - floor(NpR)))
            else:
                  NpR = (2 ** -24) * int(2 ** 24 * Fclk / Fout)
                  data1 = 1
                  data2 = int(floor(NpR / 4))
                  data3 = int(2 ** 26 * (NpR / 4 - floor(NpR / 4)))

            Ngt = data1
            NorNdiv4 = data2
            RorFrNpRdiv4_25to18 = data3 >> 18
            temp = data3 % (2 ** 18)
            RorFrNpRdiv4_17to10 = temp >> 10
            temp = data3 % (2 ** 10)
            RorFrNpRdiv4_9to2 = temp >> 2
            RorFrNpRdiv4_1to0 = data3 % (2 ** 2)

            return (Ngt, NorNdiv4, RorFrNpRdiv4_25to18, RorFrNpRdiv4_17to10,
RorFrNpRdiv4_9to2, RorFrNpRdiv4_1to0)

      def calc_phase_vars(self, Fclk, Fout, phsh):
            #
```

```
            #@param Fclk: Clock frequency of board (Hz)
            #@type Fclk: float
            #@param Fout: Desired clock frequency for one of three frequency
synthesizers (Hz)
            #@type Fout: float
            #@param phsh: Desired phase shift in degrees
            #@type phsh: float
            #
            # Calculate RFIC register variables to set phase of frequency
synthesizers
            # data1 is NGT3_phsh, D7, a single bit
            # data2 is NorNdiv4_phsh, D6-D0, up to 7 bits
            # data3 is RorFrNpRdiv4_phsh, up to 26 bits
            # D7-D0, D7-D0, D7-D0, D7-D6
            # Returns Ngt_phsh, NorNdiv4_phsh, RorFrNpRdiv4_25to18_phsh,
            # RorFrNpRdiv4_17to10_phsh, RorFrNpRdiv4_9to2_phsh,
RorFrNpRdiv4_1to0_phsh

            if (Fout <= Fclk / 4):
                  mod1 = phsh - 360 * floor(phsh / 360)
                  NpR = (2 ** -24) * int(2 ** 24 * Fclk / Fout)
                  tmp = (1 + mod1 / 360 / 2) * NpR
            else:
                  mod1 = phsh - 360 * floor(phsh / 360)
                  NpR = (2 ** -26) * int(2 ** 26 * Fclk / Fout)
                  tmp = (1 + mod1 / 360 / 2) * NpR

            if (tmp < 4):
                  NpR_ph = (2 ** -26) * int(2 ** 26 * (1 + mod1 / 360 / 8) *
NpR)
                  data1 = 0
                  data2 = int(floor(NpR_ph))
                  data3 = int(2 ** 26 * (NpR_ph - floor(NpR_ph)))
            elif ((tmp >=4) and (tmp < 508)):
                  NpR_ph = (2 ** -24) * int(2 ** 24 * tmp)
                  data1 = 1
                  data2 = int(floor(NpR_ph / 4))
                  data3 = int(2 ** 26 * (NpR_ph / 4 - floor(NpR_ph / 4)))
            elif (tmp >= 508):
                  NpR_ph = (2 ** -24) * int(2 ** 24 * (1 + (mod1 - 360) / 360
/2) * NpR)
                  data1 = 1
                  data2 = int(floor(NpR_ph / 4))
                  data3 = int(2 ** 26 * (NpR_ph / 4 - floor(NpR_ph / 4)))

            Ngt_phsh = data1
            NorNdiv4_phsh = data2
            RorFrNpRdiv4_25to18_phsh = data3 >> 18
            temp = data3 % (2 ** 18)
            RorFrNpRdiv4_17to10_phsh = temp >> 10
            temp = data3 % (2 ** 10)
            RorFrNpRdiv4_9to2_phsh = temp >> 2
            RorFrNpRdiv4_1to0_phsh = data3 % (2 ** 2)

            return (Ngt_phsh, NorNdiv4_phsh, RorFrNpRdiv4_25to18_phsh,
RorFrNpRdiv4_17to10_phsh, RorFrNpRdiv4_9to2_phsh, RorFrNpRdiv4_1to0_phsh)
```

```
      def set_rx_freq(self, target_freq):
            #
            #@param target_freq: desired receiver frequency in Hz
            #@returns (ok, actual_baseband_freq) where:
            #   ok is True or False and indicates success or failure,
            #   actual_baseband_freq is the RF frequency that corresponds to
DC in the IF.
            #

            # Go through Quadrature Generation Initialization Sequence

            #target_freq = target_freq + 4000000

            if (target_freq <= 500000000):
                  # Below 500 MHz
                  print 'Below 500 MHz, divide by 2'
                  # Use QuIET frequency divided by 2
                  # Step 1
                  self.X1sel_32to39_3 = 0
                  self.X1sel_40to47_3 = 62
                  self.X2sel_32to36_3 = 0
                  self.X2sel_37to41_3 = 0
                  self.X4sel_32to36_3 = 0
                  self.X4sel_37to41_3 = 0

                  self.set_reg_139()
                  self.set_reg_140()
                  self.set_reg_141()
                  self.set_reg_142()
                  self.set_reg_143()
                  self.set_reg_144()

                  # Step 2
                  self.X1sel_40to47_3 = 63

                  self.set_reg_140()

                  try_freq = target_freq * 2

            elif ((target_freq > 500000000) and (target_freq <= 1000000000)):
                  # Between 500 MHz and 1 GHz
                  print 'Between 500 MHz and 1 GHz'
                  # Use QuIET frequency
                  # Step 1
                  self.X1sel_32to39_3 = 1
                  self.X1sel_40to47_3 = 192
                  self.X2sel_32to36_3 = 0
                  self.X2sel_37to41_3 = 0
                  self.X4sel_32to36_3 = 0
                  self.X4sel_37to41_3 = 0

                  self.set_reg_139()
                  self.set_reg_140()
                  self.set_reg_141()
                  self.set_reg_142()
                  self.set_reg_143()
                  self.set_reg_144()
```

```
                # Step 2
                self.X1sel_32to39_3 = 73

                self.set_reg_139()

                # Step 3
                self.X1sel_32to39_3 = 201

                self.set_reg_139()

                try_freq = target_freq

                # Set Quadrature Generator Charge/Discharge Taps
                self.DischargeTap16_0to3_3 = 6
                self.ChargeTap16_0to3_3 = 7
                self.DischargeTapn_0to3_3 = 0
                self.ChargeTapn16_0to3_3 = 5

                # Set Quadrature Generator Delays
                self.Qg00degDelay_0to4_3 = 16
                self.Qg90degDelay_0to4_3 = 31
                self.Qg180degDelay_0to4_3 = 0
                self.Qg270degDelay_0to4_3 = 31

                self.set_reg_133()
                self.set_reg_134()
                self.set_reg_135()
                self.set_reg_136()
                self.set_reg_137()
                self.set_reg_138()

            elif ((target_freq > 1000000000) and (target_freq <=
2000000000)):
                # Between 1 GHz and 2 GHz
                print 'Between 1 GHz and 2 GHz, multiply by 2'
                # Use QuIET multiplied by 2
                # Step 1
                self.X1sel_32to39_3 = 0
                self.X1sel_40to47_3 = 0
                self.X2sel_32to36_3 = 0
                self.X2sel_37to41_3 = 7
                self.X4sel_32to36_3 = 0
                self.X4sel_37to41_3 = 0

                self.set_reg_139()
                self.set_reg_140()
                self.set_reg_141()
                self.set_reg_142()
                self.set_reg_143()
                self.set_reg_144()

                # Step 2
                self.X2sel_32to36_3 = 9

                self.set_reg_141()
```

```
                # Step 3
                self.X2sel_32to36_3 = 25

                self.set_reg_141()

                # Step 4
                self.X2sel_32to36_3 = 16

                self.set_reg_141()

                try_freq = target_freq / 2

                # Set Quadrature Generator Charge/Discharge Taps
                self.DischargeTap16_0to3_3 = 9
                self.ChargeTap16_0to3_3 = 3
                self.DischargeTapn_0to3_3 = 3
                self.ChargeTapn16_0to3_3 = 5

                # Set Quadrature Generator Delays
                self.Qg00degDelay_0to4_3 = 31
                self.Qg90degDelay_0to4_3 = 31
                self.Qg180degDelay_0to4_3 = 0
                self.Qg270degDelay_0to4_3 = 31

                self.set_reg_133()
                self.set_reg_134()
                self.set_reg_135()
                self.set_reg_136()
                self.set_reg_137()
                self.set_reg_138()

            elif ((target_freq > 2000000000) and (target_freq <=
4000000000)):
                # 2 to 4 GHz
                print 'From 2 to 4 GHz, multiply by 4'
                # Use QuIET frequency multiplied by 4
                # Step 1
                self.X1sel_32to39_3 = 0
                self.X1sel_40to47_3 = 0
                self.X2sel_32to36_3 = 0
                self.X2sel_37to41_3 = 0
                self.X4sel_32to36_3 = 0
                self.X4sel_37to41_3 = 7

                self.set_reg_139()
                self.set_reg_140()
                self.set_reg_141()
                self.set_reg_142()
                self.set_reg_143()
                self.set_reg_144()

                # Step 2
                self.X4sel_32to36_3 = 9

                self.set_reg_143()

                # Step 3
```

```
        self.X4sel_32to36_3 = 25

        self.set_reg_143()

        try_freq = target_freq / 4

        # Set Quadrature Generator Charge/Discharge Taps
        self.DischargeTap16_0to3_3 = 16
        self.ChargeTap16_0to3_3 = 0
        self.DischargeTapn_0to3_3 = 7
        self.ChargeTapn16_0to3_3 = 7

        # Set Quadrature Generator Delays
        self.Qg00degDelay_0to4_3 = 0
        self.Qg90degDelay_0to4_3 = 31
        self.Qg180degDelay_0to4_3 = 0
        self.Qg270degDelay_0to4_3 = 31

        self.set_reg_133()
        self.set_reg_134()
        self.set_reg_135()
        self.set_reg_136()
        self.set_reg_137()
        self.set_reg_138()

    elif (target_freq > 4000000000):
        # Above 4 GHz, doesn't work
        return (False, target_freq)

        # FIXME

        '''# Above 4 GHz
        print 'Above 4 GHz, multiply by 8'
        # Use QuIET frequency multiplied by 8
        # Step 1
        self.X1sel_32to39_3 = 0
        self.X1sel_40to47_3 = 0
        self.X2sel_32to36_3 = 0
        self.X2sel_37to41_3 = 0
        self.X4sel_32to36_3 = 0
        self.X4sel_37to41_3 = 0
        self.X8sel_32to36_3 = 0
        self.X8sel_41_3 = 0
        self.X8sel_37to40_3 = 7

        self.set_reg_139()
        self.set_reg_140()
        self.set_reg_141()
        self.set_reg_142()
        self.set_reg_143()
        self.set_reg_144()
        self.set_reg_145()
        self.set_reg_146()

        # Step 2
        self.X8sel_32to36_3 = 9
```

```
                self.set_reg_145()

                # Step 3
                self.X8sel_32to36_3 = 25

                self.set_reg_145()

                try_freq = target_freq / 8

                # Set Quadrature Generator Charge/Discharge Taps
                self.ChargeTap16_0to3_3 = 15
                self.ChargeTapn16_0to3_3 = 15

                self.DischargeTap16_0to3_3 = 6
                self.DischargeTapn16_0to3_3 = 4

                self.set_reg_137()
                self.set_reg_138()'''


            self.Foutrx = target_freq

            (self.Ngt3_3, self.NorNdiv4_3, self.RorFrNpRdiv4_25to18_3,
self.RorFrNpRdiv4_17to10_3, self.RorFrNpRdiv4_9to2_3,
self.RorFrNpRdiv4_1to0_3) = self.calc_freq_vars(self.Fclk, try_freq)

            self.set_reg_104()
            self.set_reg_105()
            self.set_reg_106()
            self.set_reg_107()
            self.set_reg_108()

            return (1, target_freq)
            #FIXME -- How do I know if the RFIC successfully attained the
desired frequency?#


      def set_tx_freq(self, target_freq):
            #
            #@param target_freq: desired transmitter frequency in Hz
            #@returns (ok, actual_baseband_freq) where:
            #   ok is True or False and indicates success or failure,
            #   actual_baseband_freq is the RF frequency that corresponds to
DC in the IF.
            #

            # Go through Quadrature Generation Initialization Sequence

            # FIXME
            #target_freq = target_freq + 4000000
            #target_freq = target_freq + 1000000

            if (target_freq <= 500000000):
                  print 'Below 500 MHz, divide by 2'
                  # Use QuIET frequency divided by 2
                  # Step 1
                  self.X1sel_32to39 = 0
```

```
        self.X1sel_40to47 = 62
        self.X2sel_32to36 = 0
        self.X2sel_37to41 = 0
        self.X4sel_32to36 = 0
        self.X4sel_37to41 = 0

        self.set_reg_35()
        self.set_reg_36()
        self.set_reg_37()
        self.set_reg_38()
        self.set_reg_39()
        self.set_reg_40()

        # Step 2
        self.X1sel_40to47 = 63

        self.set_reg_36()

        try_freq = target_freq * 2

    elif ((target_freq > 500000000) and (target_freq <= 1000000000)):
        print 'Between 500 MHz and 1 GHz'
        # Use QuIET frequency
        # Step 1
        self.X1sel_32to39 = 1
        self.X1sel_40to47 = 192
        self.X2sel_32to36 = 0
        self.X2sel_37to41 = 0
        self.X4sel_32to36 = 0
        self.X4sel_37to41 = 0

        self.set_reg_35()
        self.set_reg_36()
        self.set_reg_37()
        self.set_reg_38()
        self.set_reg_39()
        self.set_reg_40()

        # Step 2
        self.X1sel_32to39 = 73

        self.set_reg_35()

        # Step 3
        self.X1sel_32to39 = 201

        self.set_reg_35()

        try_freq = target_freq

        # Set Quadrature Generator Charge/Discharge Taps and Delays
        self.Qg00degDelay_0to4 = 15
        self.Qg90degDelay_0to4 = 12
        self.Qg180degDelay_0to4 = 3
        self.Qg270degDelay_0to4 = 12

        self.set_reg_29()
```

```
                self.set_reg_30()
                self.set_reg_31()
                self.set_reg_32()

                self.DischargeTap16_0to3 = 1
                self.ChargeTap16_0to3 = 8
                self.DischargeTapn_0to3 = 7
                self.ChargeTapn16_0to3 = 0

                self.set_reg_33()
                self.set_reg_34()

            elif ((target_freq > 1000000000) and (target_freq <=
2000000000)):
                print 'Between 1 GHz and 2 GHz, multiply by 2'
                # Use QuIET multiplied by 2
                # Step 1
                self.X1sel_32to39 = 0
                self.X1sel_40to47 = 0
                self.X2sel_32to36 = 0
                self.X2sel_37to41 = 7
                self.X4sel_32to36 = 0
                self.X4sel_37to41 = 0

                self.set_reg_35()
                self.set_reg_36()
                self.set_reg_37()
                self.set_reg_38()
                self.set_reg_39()
                self.set_reg_40()

                # Step 2
                self.X2sel_32to36 = 9

                self.set_reg_37()

                # Step 3
                self.X2sel_32to36 = 25

                self.set_reg_37()

                # Step 4
                #self.X2sel_32to36 = 16

                #self.set_reg_37()

                try_freq = target_freq / 2

                # Set Quadrature Generator Charge/Discharge Taps and Delays
                self.Qg00degDelay_0to4 = 7
                self.Qg90degDelay_0to4 = 8
                self.Qg180degDelay_0to4 = 7
                self.Qg270degDelay_0to4 = 5

                self.set_reg_29()
                self.set_reg_30()
                self.set_reg_31()
```

```
                self.set_reg_32()

                self.DischargeTap16_0to3 = 1
                self.ChargeTap16_0to3 = 13

                self.DischargeTapn_0to3 = 3
                self.ChargeTapn16_0to3 = 9

                self.set_reg_33()
                self.set_reg_34()

            elif ((target_freq > 2000000000) and (target_freq <=
4000000000)):
                print '2-4 GHz, multiply by 4'
                # Use QuIET frequency multiplied by 4
                # Step 1
                self.X1sel_32to39 = 0
                self.X1sel_40to47 = 0
                self.X2sel_32to36 = 0
                self.X2sel_37to41 = 0
                self.X4sel_32to36 = 0
                self.X4sel_37to41 = 7

                self.set_reg_35()
                self.set_reg_36()
                self.set_reg_37()
                self.set_reg_38()
                self.set_reg_39()
                self.set_reg_40()

                # Step 2
                self.X4sel_32to36 = 9

                self.set_reg_39()

                # Step 3
                self.X4sel_32to36 = 25

                self.set_reg_39()

                try_freq = target_freq / 4

                # Set Quadrature Generator Charge/Discharge Taps and Delays
                self.Qg00degDelay_0to4 = 0
                self.Qg90degDelay_0to4 = 17
                self.Qg180degDelay_0to4 = 15
                self.Qg270degDelay_0to4 = 20

                self.set_reg_29()
                self.set_reg_30()
                self.set_reg_31()
                self.set_reg_32()

                self.DischargeTap16_0to3 = 15
                self.ChargeTap16_0to3 = 0

                self.DischargeTapn_0to3 = 10
```

```
                self.ChargeTapn16_0to3 = 8

                self.set_reg_33()
                self.set_reg_34()

          elif (target_freq > 4000000000):
                # Above 4 GHz, doesn't work
                return (False, target_freq)

          self.Fouttx = target_freq

          (self.Ngt3, self.NorNdiv4, self.RorFrNpRdiv4_25to18,
self.RorFrNpRdiv4_17to10, self.RorFrNpRdiv4_9to2, self.RorFrNpRdiv4_1to0) =
self.calc_freq_vars(self.Fclk, try_freq)

          self.set_reg_0()
          self.set_reg_1()
          self.set_reg_2()
          self.set_reg_3()
          self.set_reg_4()

          return (1, target_freq)
          #FIXME -- How do I know if the RFIC successfully attained the
desired frequency?#

    def set_fb_freq(self, target_freq):
          #
          #@param target_freq: desired transmitter frequency in Hz
          #@returns (ok, actual_baseband_freq) where:
          #   ok is True or False and indicates success or failure,
          #   actual_baseband_freq is the RF frequency that corresponds to
DC in the IF.
          #

          # Go through Quadrature Generation Initialization Sequence

          if (target_freq <= 500000000):
                print 'Below 500 MHz, divide by 2'
                # Use QuIET frequency divided by 2
                # Step 1
                self.X1sel_32to39_2 = 0
                self.X1sel_40to47_2 = 62
                self.X2sel_32to36_2 = 0
                self.X2sel_37to41_2 = 0
                self.X4sel_32to36_2 = 0
                self.X4sel_37to41_2 = 0

                self.set_reg_83()
                self.set_reg_84()
                self.set_reg_85()
                self.set_reg_86()
                self.set_reg_87()
                self.set_reg_88()

                # Step 2
                self.X1sel_40to47_2 = 63
```

```
                self.set_reg_84()

                try_freq = target_freq * 2

            elif ((target_freq > 500000000) and (target_freq <= 1000000000)):
                print 'Between 500 MHz and 1 GHz'
                # Use QuIET frequency
                # Step 1
                self.X1sel_32to39_2 = 1
                self.X1sel_40to47_2 = 192
                self.X2sel_32to36_2 = 0
                self.X2sel_37to41_2 = 0
                self.X4sel_32to36_2 = 0
                self.X4sel_37to41_2 = 0

                self.set_reg_83()
                self.set_reg_84()
                self.set_reg_85()
                self.set_reg_86()
                self.set_reg_87()
                self.set_reg_88()

                # Step 2
                self.X1sel_32to39_2 = 73

                self.set_reg_83()

                # Step 3
                self.X1sel_32to39_2 = 201

                self.set_reg_83()

                try_freq = target_freq

                # Set Quadrature Generator Charge/Discharge Taps
                # FIXME
                #self.ChargeTap16_0to3_2 = 7
                #self.ChargeTapn16_0to3_2 = 5

                #self.DischargeTap16_0to3_2 = 6
                #self.DischargeTapn16_0to3_2 = 0

                #self.set_reg_81()
                #self.set_reg_82()

            elif ((target_freq > 1000000000) and (target_freq <=
2000000000)):
                print 'Between 1 GHz and 2 GHz, multiply by 2'
                # Use QuIET multiplied by 2
                # Step 1
                self.X1sel_32to39_2 = 0
                self.X1sel_40to47_2 = 0
                self.X2sel_32to36_2 = 0
                self.X2sel_37to41_2 = 7
                self.X4sel_32to36_2 = 0
                self.X4sel_37to41_2 = 0
```

```
                self.set_reg_83()
                self.set_reg_84()
                self.set_reg_85()
                self.set_reg_86()
                self.set_reg_87()
                self.set_reg_88()

                # Step 2
                self.X2sel_32to36_2 = 9

                self.set_reg_85()

                # Step 3
                self.X2sel_32to36_2 = 25

                # Step 4
                #self.X2sel_32to36 = 16

                self.set_reg_85()

                try_freq = target_freq / 2

                # Set Quadrature Generator Charge/Discharge Taps
                # FIXME
                #self.ChargeTap16_0to3_2 = 7
                #self.ChargeTapn16_0to3_2 = 8

                #self.DischargeTap16_0to3_2 = 15
                #self.DischargeTapn16_0to3_2 = 15

                #self.set_reg_81()
                #self.set_reg_82()

            elif ((target_freq > 2000000000) and (target_freq <=
4000000000)):
                print '2-4 GHz, multiply by 4'
                # Use QuIET frequency multiplied by 4
                # Step 1
                self.X1sel_32to39_2 = 0
                self.X1sel_40to47_2 = 0
                self.X2sel_32to36_2 = 0
                self.X2sel_37to41_2 = 0
                self.X4sel_32to36_2 = 0
                self.X4sel_37to41_2 = 7

                self.set_reg_83()
                self.set_reg_84()
                self.set_reg_85()
                self.set_reg_86()
                self.set_reg_87()
                self.set_reg_88()

                # Step 2
                self.X4sel_32to36_2 = 9

                self.set_reg_87()
```

```
                  # Step 3
                  self.X4sel_32to36_2 = 25

                  self.set_reg_87()

                  try_freq = target_freq / 4

                  # Set Quadrature Generator Charge/Discharge Taps
                  # FIXME
                  #self.ChargeTap16_0to3_2 = 15
                  #self.ChargeTapn16_0to3_2 = 15

                  #self.DischargeTap16_0to3_2 = 6
                  #self.DischargeTapn16_0to3_2 = 4

                  #self.set_reg_81()
                  #self.set_reg_82()

            elif (target_freq > 4000000000):
                  # Above 4 GHz, doesn't work
                  return (False, target_freq)

            self.Foutfb = target_freq

            (self.Ngt3_2, self.NorNdiv4_2, self.RorFrNpRdiv4_25to18_2,
self.RorFrNpRdiv4_17to10_2, self.RorFrNpRdiv4_9to2_2,
self.RorFrNpRdiv4_1to0_2) = self.calc_freq_vars(self.Fclk, try_freq)

            self.set_reg_48()
            self.set_reg_49()
            self.set_reg_50()
            self.set_reg_51()
            self.set_reg_52()

            return (1, target_freq)
            #FIXME -- How do I know if the RFIC successfully attained the
desired frequency?#

      def set_rx_phase(self, phsh):
            #
            #@param phsh: desired phase shift in degrees
            #@returns (ok) where:
            #   ok is True or False and indicates success or failure
            #

            phsh = phsh % 360

            if (self.Foutrx <= 500000000):
                  synth_freq = self.Foutrx * 2
            elif ( (self.Foutrx > 500000000) and (self.Foutrx <=
1000000000)):
                  synth_freq = self.Foutrx
            elif ( (self.Foutrx > 1000000000) and (self.Foutrx <
2000000000)):
                  synth_freq = self.Foutrx / 2
            elif (self.Foutrx > 2000000000):
                  synth_freq = self.Foutrx / 4
```

```
            (self.Qu_tx_Ngt3_3, self.NorNdiv4_phsh_3,
self.RorFrNpRdiv4_phsh_25to18_3, self.RorFrNpRdiv4_phsh_17to10_3,
self.RorFrNpRdiv4_phsh_9to2_3, self.RorFrNpRdiv4_phsh_1to0_3) =
self.calc_phase_vars(self.Fclk, synth_freq, phsh)

            self.set_reg_109()
            self.set_reg_110()
            self.set_reg_111()
            self.set_reg_112()
            self.set_reg_113()

            return (1)
            #FIXME -- How do I know if the RFIC successfully attained the
desired phase?#


      def set_tx_phase(self, phsh):
            #
            #@param phsh: desired phase shift in degrees
            #@returns (ok) where:
            #   ok is True or False and indicates success or failure
            #

            phsh = phsh % 360

            if (self.Fouttx <= 500000000):
                  synth_freq = self.Fouttx * 2
            elif ( (self.Fouttx > 500000000) and (self.Fouttx <=
1000000000)):
                  synth_freq = self.Fouttx
            elif ( (self.Fouttx > 1000000000) and (self.Fouttx <
2000000000)):
                  synth_freq = self.Fouttx / 2
            elif (self.Fouttx > 2000000000):
                  synth_freq = self.Fouttx / 4

            (self.Qu_tx_Ngt3_3, self.NorNdiv4_phsh_3,
self.RorFrNpRdiv4_phsh_25to18_3, self.RorFrNpRdiv4_phsh_17to10_3,
self.RorFrNpRdiv4_phsh_9to2_3, self.RorFrNpRdiv4_phsh_1to0_3) =
self.calc_phase_vars(self.Fclk, synth_freq, phsh)

            self.set_reg_5()
            self.set_reg_6()
            self.set_reg_7()
            self.set_reg_8()
            self.set_reg_9()

            #FIXME -- How do I know if the RFIC successfully attained the
desired phase?#
            return (1)

      def set_fb_phase(self, phsh):
            #
            #@param phsh: desired phase shift in degrees
            #@returns (ok) where:
            #   ok is True or False and indicates success or failure
```

```
            #

            phsh = phsh % 360

            if (self.Foutfb <= 500000000):
                  synth_freq = self.Foutfb * 2
            elif ( (self.Foutfb > 500000000) and (self.Foutfb <=
1000000000)):
                  synth_freq = self.Foutfb
            elif ( (self.Foutfb > 1000000000) and (self.Foutfb <
2000000000)):
                  synth_freq = self.Foutfb / 2
            elif (self.Foutfb > 2000000000):
                  synth_freq = self.Foutfb / 4

            (self.Qu_tx_Ngt3_3, self.NorNdiv4_phsh_3,
self.RorFrNpRdiv4_phsh_25to18_3, self.RorFrNpRdiv4_phsh_17to10_3,
self.RorFrNpRdiv4_phsh_9to2_3, self.RorFrNpRdiv4_phsh_1to0_3) =
self.calc_phase_vars(self.Fclk, synth_freq, phsh)

            self.set_reg_53()
            self.set_reg_54()
            self.set_reg_55()
            self.set_reg_56()
            self.set_reg_57()

            #FIXME -- How do I know if the RFIC successfully attained the
desired phase?#
            return (1)

      def set_rx_bw(self, bw):
            #
            #@param bw: desired bandwidth in Hz
            #
            # Available bandwidth: 4.25 kHz to 14 MHz (baseband)
            # FIXME
            print 'Desired bandwidth: %s' % (bw)
            if bw <= 5250:
                  # Set BW to 3.532 kHz
                  self.rx_rfp = 3
                  self.rx_cp_12to8 = 31
                  self.rx_cp_7to0 = 240

                  self.rx_rv = 7
                  self.rx_cv_10to3 = 254
                  self.rx_cv_2to0 = 0

                  self.rx_rq = 7
                  self.rx_cq_9to8 = 3
                  self.rx_cq_7to0 = 240

            elif bw > 5250 and bw <= 10500:
                  # Set BW to 7.065 kHz
                  self.rx_rfp = 3
                  self.rx_cp_12to8 = 31
                  self.rx_cp_7to0 = 240
```

```
            self.rx_rv = 5
            self.rx_cv_10to3 = 254
            self.rx_cv_2to0 = 0

            self.rx_rq = 5
            self.rx_cq_9to8 = 3
            self.rx_cq_7to0 = 240

        elif bw > 10500 and bw <= 21000:
            # Set BW to 14.130 kHz
            self.rx_rfp = 2
            self.rx_cp_12to8 = 31
            self.rx_cp_7to0 = 240

            self.rx_rv = 4
            self.rx_cv_10to3 = 254
            self.rx_cv_2to0 = 0

            self.rx_rq = 4
            self.rx_cq_9to8 = 3
            self.rx_cq_7to0 = 240

        elif bw > 21000 and bw <= 42000:
            # Set BW to 28.259 kHz
            self.rx_rfp = 2
            self.rx_cp_12to8 = 15
            self.rx_cp_7to0 = 240

            self.rx_rv = 3
            self.rx_cv_10to3 = 254
            self.rx_cv_2to0 = 0

            self.rx_rq = 3
            self.rx_cq_9to8 = 3
            self.rx_cq_7to0 = 240

        elif bw > 42000 and bw <= 84500:
            # Set BW to 56.518 kHz
            self.rx_rfp = 2
            self.rx_cp_12to8 = 7
            self.rx_cp_7to0 = 240

            self.rx_rv = 2
            self.rx_cv_10to3 = 254
            self.rx_cv_2to0 = 0

            self.rx_rq = 2
            self.rx_cq_9to8 = 3
            self.rx_cq_7to0 = 240

        elif bw > 84500 and bw <= 169500:
            # Set BW to 113.036 kHz
            self.rx_rfp = 2
            self.rx_cp_12to8 = 3
            self.rx_cp_7to0 = 240

            self.rx_rv = 1
```

```
            self.rx_cv_10to3 = 254
            self.rx_cv_2to0 = 0

            self.rx_rq = 1
            self.rx_cq_9to8 = 3
            self.rx_cq_7to0 = 240

        elif bw > 169500 and bw <= 339000:
            # Set BW to 226.072 kHz
            self.rx_rfp = 2
            self.rx_cp_12to8 = 1
            self.rx_cp_7to0 = 240

            self.rx_rv = 1
            self.rx_cv_10to3 = 126
            self.rx_cv_2to0 = 0

            self.rx_rq = 1
            self.rx_cq_9to8 = 1
            self.rx_cq_7to0 = 240

        elif bw > 339000 and bw <= 667000:
            # Set BW to 452.145 kHz
            self.rx_rfp = 1
            self.rx_cp_12to8 = 1
            self.rx_cp_7to0 = 240

            self.rx_rv = 0
            self.rx_cv_10to3 = 254
            self.rx_cv_2to0 = 0

            self.rx_rq = 1
            self.rx_cq_9to8 = 0
            self.rx_cq_7to0 = 240

        elif bw > 667000 and bw <= 1356000:
            # Set BW to 904.289 kHz
            self.rx_rfp = 1
            self.rx_cp_12to8 = 0
            self.rx_cp_7to0 = 240

            self.rx_rv = 0
            self.rx_cv_10to3 = 126
            self.rx_cv_2to0 = 0

            self.rx_rq = 0
            self.rx_cq_9to8 = 3
            self.rx_cq_7to0 = 240

        elif bw > 1356000 and bw <= 2712500:
            # Set BW to 1808.579 kHz
            self.rx_rfp = 1
            self.rx_cp_12to8 = 0
            self.rx_cp_7to0 = 112

            self.rx_rv = 0
            self.rx_cv_10to3 = 62
```

```
        self.rx_cv_2to0 = 0

        self.rx_rq = 0
        self.rx_cq_9to8 = 1
        self.rx_cq_7to0 = 240

    elif bw > 2712500 and bw <= 5425500:
        # Set BW to 3617.157 kHz
        self.rx_rfp = 0
        self.rx_cp_12to8 = 0
        self.rx_cp_7to0 = 112

        self.rx_rv = 0
        self.rx_cv_10to3 = 30
        self.rx_cv_2to0 = 0

        self.rx_rq = 0
        self.rx_cq_9to8 = 0
        self.rx_cq_7to0 = 240

    elif bw > 5425500 and bw <= 10851000:
        # Set BW to 7234.315 kHz
        self.rx_rfp = 0
        self.rx_cp_12to8 = 0
        self.rx_cp_7to0 = 48

        self.rx_rv = 0
        self.rx_cv_10to3 = 14
        self.rx_cv_2to0 = 0

        self.rx_rq = 0
        self.rx_cq_9to8 = 0
        self.rx_cq_7to0 = 112

    elif bw > 10851000:
        # Set BW to 14468.630 kHz
        self.rx_rfp = 0
        self.rx_cp_12to8 = 0
        self.rx_cp_7to0 = 16

        self.rx_rv = 0
        self.rx_cv_10to3 = 6
        self.rx_cv_2to0 = 0

        self.rx_rq = 0
        self.rx_cq_9to8 = 0
        self.rx_cq_7to0 = 48

    self.set_reg_198()
    self.set_reg_199()
    self.set_reg_200()
    self.set_reg_201()
    self.set_reg_202()
    self.set_reg_203()
    self.set_reg_204()
```

```
def set_tx_bw(self, bw):
      #
      #@param bw: desired bandwidth in Hz
      #
      # Available bandwidth: 6.25 kHz to 14+ MHz (baseband)
      # FIXME
      print 'Desired bandwidth: %s' % (bw)
      if bw <= 20000:
            # Set BW to 12.5 kHz
            self.tx_p1_bw = 3
            self.tx_p2_bw2 = 15

      elif bw > 20000 and bw <= 37500:
            # Set BW to 25 kHz
            self.tx_p1_bw = 3
            self.tx_p2_bw2 = 7

      elif bw > 37500 and bw <= 75000:
            # Set BW to 50 kHz
            self.tx_p1_bw = 3
            self.tx_p2_bw2 = 3

      elif bw > 75000 and bw <= 150000:
            # Set BW to 100 kHz
            self.tx_p1_bw = 3
            self.tx_p2_bw2 = 1

      elif bw > 150000 and bw <= 425000:
            # Set BW to 200 kHz
            self.tx_p1_bw = 3
            self.tx_p2_bw2 = 0

      elif bw > 425000 and bw <= 1125000:
            # Set BW to 750 kHz
            self.tx_p1_bw = 1
            self.tx_p2_bw2 = 15

      elif bw > 1125000 and bw <= 2250000:
            # Set BW to 1.5 MHz
            self.tx_p1_bw = 1
            self.tx_p2_bw2 = 7

      elif bw > 2250000 and bw <= 4500000:
            # Set BW to 3 MHz
            self.tx_p1_bw = 1
            self.tx_p2_bw2 = 3

      elif bw > 4500000 and bw <= 9000000:
            # Set BW to 6 MHz
            self.tx_p1_bw = 1
            self.tx_p2_bw2 = 1

      elif bw > 9000000 and bw <= 13000000:
            # Set BW to 12 MHz
            self.tx_p1_bw = 1
            self.tx_p2_bw2 = 0
```

```
            elif bw > 13000000:
                  # Set BW to 14+ MHz
                  self.tx_p1_bw = 0
                  self.tx_p2_bw2 = 0

            self.set_reg_173()
            self.set_reg_174()

      def set_fb_bw(self, bw):
            #
            #@param bw: desired bandwidth in Hz
            #
            # Available bandwidth: 5 MHz to 14+ MHz (baseband)
            # FIXME
            print 'Desired bandwidth: %s' % (bw)
            if bw <= 7500000:
                  # Set BW to 5 MHz
                  self.tx_bb_fdbk_bw = 3

            elif bw > 7500000 and bw <= 12000000:
                  # Set BW to 10 MHz
                  self.tx_bb_fdbk_bw = 1

            elif bw > 12000000:
                  # Set BW to 14+ MHz
                  self.tx_bb_fdbk_bw = 0

            self.set_reg_156()

      def enable_tx_fb(self):
            #
            # Enable transmitter feedback to RX port for DC offset
correction, etc.
            #
            # FIXME
            print 'Enabling Transmit Feedback'

            # Disable RX Filter
            self.rx_foe = 0
            self.set_reg_196()

            # Enable Baseband Feedback, TX I and Q via RX I and Q
            self.tx_bb_fdbk_en = 3
            self.set_reg_157()

            # Disable Baseband Feedback Calibration
            # FIXME
            #self.tx_bb_fdbk_cal_en = 0

            # Enable Baseband Feedback Cartesian Forward Path
            self.tx_bb_fdbk_cart_fwd_en = 1
            self.set_reg_156()

            # Enable Cartesian Feedback Path
            self.tx_cart_en = 1
            self.set_reg_160()
```

```
        # Enable Cartesian Feedback
        self.CartesianFeedbackpathenable = 1

        # Enable Cartesian Feedback Path DCOC
        self.CartesianFeedbackpathDCOCenable = 1
        self.set_reg_166()

        # Set Cartesian Feedback Path Amplifier Gain
        self.CartesianFBpathAmp1Gain = 0
        self.CartesianFBpathAmp2Gain = 0
        self.CartesianFBpathAmp3Gain = 0
        self.CartesianFBpathAmp4Gain = 0
        self.set_reg_169()

        # Enable Cartesian Feedback Path Zero
        self.CartesianFBpathZeroEnable = 1
        self.set_reg_170()

    def disable_tx_fb(self):
        #
        # Disable transmitter feedback to RX port
        #
        # FIXME
        print 'Disabling Transmit Feedback'

        # Enable RX Filter
        self.rx_foe = 1
        self.set_reg_196()

        # Disable Baseband Feedback
        self.tx_bb_fdbk_en = 0
        self.set_reg_157()

        # Enable Baseband Feedback Calibration
        # FIXME
        #self.tx_bb_fdbk_cal_en = 1

        # Disable Baseband Feedback Cartesian Forward Path
        self.tx_bb_fdbk_cart_fwd_en = 0
        self.set_reg_156()

        # Disable Cartesian Feedback Path
        self.tx_cart_en = 0
        self.set_reg_160()

        # Disable Cartesian Feedback
        self.CartesianFeedbackpathenable = 0

        # Disable Cartesian Feedback Path DCOC
        self.CartesianFeedbackpathDCOCenable = 0
        self.set_reg_166()

        # Set Cartesian Feedback Path Amplifier Gain
        self.CartesianFBpathAmp1Gain = 3
        self.CartesianFBpathAmp2Gain = 3
        self.CartesianFBpathAmp3Gain = 3
        self.CartesianFBpathAmp4Gain = 3
```

```
            self.set_reg_169()

            # Disable Cartesian Feedback Path Zero
            self.CartesianFBpathZeroEnable = 0
            self.set_reg_170()

      def RSSI(self):
            # Return fade, clip from the two RX-side ADCs.
            #@returns fade, clip
            # variables proportional to how much fading (low signal strength)
            # or clipping (high signal strength) is going on

            # Turn off test mux
            self.TestMuxBufferEnable = 0 #Disable Test Mux Buffer
            self.TestMuxEnable = 0 #Disable Test Mux
            self.TestMuxSetting = 0 #Four Output Description (Test1, Test2,
Test3, Test4)
            self.set_reg_222()

            # Turn on on-channel detectors
            # Off-channel doesn't work - leave it off
            self.rx_onchen = 1 #Enables on-channel detector.
            self.rx_offchen = 0 #Disables off-channel detector
            self.set_reg_196()

            # Set clip and fade thresholds
            self.rx_offch = 1 #Sets the Clip Threshold for the Off-channel
Detector
            self.rx_onchf = 0 #Sets the Fade Threshold for the On-channel
Detector relative to the On-channel clip point.
            self.rx_onchc = 2 #Sets the Clip Threshold for the On-channel
Detector
            self.set_reg_197()

            fade = self.u.read_aux_adc(self.which, 0)
            clip = self.u.read_aux_adc(self.which, 1)
            return (fade, clip)


class db_rfic_base(db_base.db_base):
      #
      #Abstract base class for all RFIC boards.
      #
      #Derive board specific subclasses from db_rfic_base_{tx,rx}
      #

      def __init__(self, usrp, which):
            #
            #@param usrp: instance of usrp.source_c
            #@param which: which side: 0 or 1 corresponding to side A or B
respectively
            #@type which: int
            #

            # sets _u  _which _tx and _slot
            db_base.db_base.__init__(self, usrp, which)
```

```
            self.rfic = _get_or_make_rfic(usrp, which)

      def __del__(self):

            #FIXME#
            return True

      def is_quadrature(self):
            #
            #Return True if this board requires both I and Q analog channels.
            #
            #This bit of info is useful when setting up the USRP Rx mux
register.
            #
            return True

      def freq_range(self):
            # Return frequency range of RFIC daughterboard
            #FIXME#
            return (1e8, 2.5e6, 1e3)

      # ------------------------------------------------------------------
-------------


class db_rfic_tx(db_rfic_base):
      def __init__(self, usrp, which):
            #
            #@param usrp: instance of usrp.sink_c
            #@param which: 0 or 1 corresponding to side TX_A or TX_B,
respectively.
            #
            print "db_rfic_tx: __init__"
            db_rfic_base.__init__(self, usrp, which)

            # Get digital block out of digital reset state
            self.rfic.Rst_n_async = 1
            self.rfic.set_reg_24()

            # Turn on forward baseband reference section
            self.rfic.tx_bb_en = 1
            # FIXME
            #self.rfic.set_reg_156()

            # Unroutes the Cartesian error signal through the BB Correction
feedback
            # FIXME
            self.rfic.tx_bb_fdbk_cart_err_en = 0

            # Routes the Cartesian feedback signal through the BB Correction
feedback
            # FIXME
            self.rfic.tx_bb_fdbk_cart_fb_en = 1
            self.rfic.set_reg_156()

            # Turn on baseband feedback section
```

```
            # FIXME
            #self.rfic.tx_bb_fdbk_en = 3
            #self.rfic.set_reg_157()

            # Turn on forward RF transmit path
            self.rfic.RFForwardPathEnable_toMUX = 1
            self.rfic.set_reg_175()

            # Turn on Cartesian FB path switch to forward summer
            self.rfic.CartesianFBpathSwitchtoforwardSummer = 1
            self.rfic.set_reg_168()

            # Turn on Cartesian zero
            self.CartesianFBpathZeroEnable = 1
            self.rfic.set_reg_170()

            # Select TX output path, default tx1
            # FIXME
            #self.rfic.tx_output_channel_sel = 2
            self.rfic.tx_output_channel_sel = 1
            self.rfic.set_reg_172()

            # Set TX Channel 1 Gain
            # The gain control on TX channel 1 is controlled by this DAC
            # The maximum voltage is 2.2 volts, which corresponds to 2750
            # This controls about 35 dB of gain ONLY ON TX 1
            self.rfic.u.write_aux_dac(self.rfic.which, 3, 2750)


            # POR On.  This enables the clock that drives the digital block
(which provides the tap selection process).  It must be enabled to generate
an output.  See Byp_fine, address 10, bit 6
            self.rfic.Clk_driver_en = 1

            # POR On
            self.rfic.qu_reg_en = 1

            # POR On
            self.rfic.qq_reg_en = 1

            # POR Off
            self.rfic.win_rst = 0

            # POR On
            self.rfic.fineEn = 0

            # POR Off
            self.rfic.fineEnb = 1

            # POR On
            #self.rfic.rsffEn = 0

            # POR On
            self.rfic.dl_en = 1

            # POR On
            self.rfic.cp_en = 1
```

```
            self.rfic.set_reg_20()
            self.rfic.set_reg_21()

      def __del__(self):
            # print "rfic_base_tx.__del__"
            # Power down

            # Turn off output channel
            self.rfic.tx_output_channel_sel = 0
            self.rfic.set_reg_172()

            # Turn off forward RF transmit path
            self.rfic.RFForwardPathEnable_toMUX = 0
            self.rfic.set_reg_17

            # Turn off forward baseband reference section
            self.rfic.tx_bb_en = 0
            self.rfic.set_reg_156()

            # Unroutes the Cartesian error signal through the BB Correction
feedback
            # FIXME
            self.rfic.tx_bb_fdbk_cart_err_en = 0

            # Unroutes the Cartesian feedback signal through the BB
Correction feedback
            self.rfic.tx_bb_fdbk_cart_fb_en = 0
            self.rfic.set_reg_156()

            # Turn off Cartesian FB path switch to forward summer
            self.rfic.CartesianFBpathSwitchtoforwardSummer = 0
            self.rfic.set_reg_168()

            # Turn off Cartesian zero
            self.CartesianFBpathZeroEnable = 0
            self.rfic.set_reg_170()

            # Turn off baseband feedback section
            # FIXME
            #self.rfic.tx_bb_fdbk_en = 0
            #self.rfic.set_reg_157()

            # POR Off.  This enables the clock that drives the digital block
(which provides the tap selection process).  It must be enabled to generate
an output.  See Byp_fine, address 10, bit 6
            self.rfic.Clk_driver_en = 0

            # POR Off
            self.rfic.qu_reg_en = 0

            # POR Off
            self.rfic.qq_reg_en = 0

            # POR Off
            self.rfic.win_rst = 0
```

```
            # POR Off
            self.rfic.fineEn = 0

            # POR Off
            self.rfic.fineEnb = 0

            # POR Off
            #self.rfic.rsffEn = 0

            # POR Off
            self.rfic.dl_en = 0

            # POR Off
            self.rfic.cp_en = 0

            self.rfic.set_reg_20()
            self.rfic.set_reg_21()

            # Put digital block in digital reset state
            self.rfic.Rst_n_async = 0
            self.rfic.set_reg_24()

            db_rfic_base.__del__(self)

      def select_tx_antenna(self, which_antenna):
            #
            #Specify which antenna port to use for transmission.
            #@param which_antenna: either 'tx1', 'tx2' or 'tx3'
            #
            if which_antenna in (0, 'tx1'):
                  self.rfic.tx_output_channel_sel = 1
                  self.rfic.set_reg_172()
            elif which_antenna in (1, 'tx2'):
                  self.rfic.tx_output_channel_sel = 2
                  self.rfic.set_reg_172()
            elif which_antenna in (2, 'tx3'):
                  self.rfic.tx_output_channel_sel = 4
                  self.rfic.set_reg_172()
            else:
                  raise ValueError, "which_antenna must be either 'tx1',
'tx2' or 'tx3'"

      def gain_range(self):
            # Gain range for transmitter, in dB, 0 to 45 in increments of 5
dB
            return (0.0, 45.0, 5)

      def set_gain(self, gain):
            # Set transmit gain, in dB
            return self.rfic.set_tx_gain(gain)

      def set_freq(self, target_freq):
            # Set transmit frequency, in Hz
            return self.rfic.set_tx_freq(target_freq)

      def set_phase(self, phase):
            # Set transmit phase offset, in degrees
```

```
            return self.rfic.set_tx_phase(phase)

      def set_bw(self, bw):
            # Set transmit bandwidth, in Hz
            return self.rfic.set_tx_bw(bw)

      def spectrum_inverted(self):
            # FIXME
            # Return True if the dboard gives an inverted spectrum
            return True
            #return False


class db_rfic_rx(db_rfic_base):
      def __init__(self, usrp, which):
            #
            #@param usrp: instance of usrp.sink_c
            #@param which: 0 or 1 corresponding to side TX_A or TX_B,
respectively.
            #
            print "db_rfic_rx: __init__"
            db_rfic_base.__init__(self, usrp, which)

            # Get digital block out of digital reset state
            self.rfic.Rst_n_async_3 = 1
            self.rfic.set_reg_128()

            # Set RX LNA port to LNA1 (SGO non-chopping mixer)
            # FIXME
            self.rfic.rx_lna = 1
            #self.rfic.rx_lna = 5

            # Set LNA bias
            self.rfic.rx_lnab = 1

            # Enable LO clock to mixer
            self.rfic.rx_rxchen = 1

            self.rfic.set_reg_205()

            # Enable RX Filter
            self.rfic.rx_fen = 1

            # Enable baseband filter chopper clock
            self.rfic.rx_chcken = 1

            # Enable chopper clock to all mixers
            self.rfic.rx_cen = 7

            # Set chopper divide setting
            # FIXME
            #self.rfic.rx_chck = 0
            self.rfic.rx_chck = 1

            self.rfic.set_reg_195()
```

```
# Enable filter output
self.rfic.rx_foe = 1

# Enable on-channel detector
#self.rfic.rx_onchen = 1

# Enable off-channel detector
#self.rfic.rx_offchen = 1


self.rfic.set_reg_196()

# Set BQ filter Q to 1.33
self.rfic.rx_qs = 2

# Set BQ resistor value to 1.4 kohms
self.rfic.rx_rq = 0

self.rfic.set_reg_198()

# Set VGA resistor value to 2.5 kohms
self.rfic.rx_rv = 0

# Set PMA Rf resistor to 5 kohms
self.rfic.rx_rfp = 00

self.rfic.set_reg_199()

# Set compensation control
self.rfic.rx_cc = 0

self.rfic.set_reg_203()

# Enable DCOC DAC
self.rfic.rx_den = 1

self.rfic.set_reg_192()

# Enable DCOC comparator
self.rfic.rx_cmpen = 1

self.rfic.set_reg_193()

# RC Tune enable
# FIXME
#self.rfic.rx_ten = 1
self.rfic.rx_ten = 0

# RC Tune ramp circuit enable
# FIXME
#self.rfic.rx_ren = 1
self.rfic.rx_ren = 0

# Select DCOC/RC Tune divider, divide by 8
self.rfic.rx_dv = 3

self.rfic.set_reg_194()
```

```
            # Enable DCOC
            self.rfic.rx_dcoc = 1

            self.rfic.set_reg_193()

            # POR On.  This enables the clock that drives the digital block
(which provides the tap selection process).  It must be enabled to generate
an output.  See Byp_fine, address 10, bit 6
            self.rfic.Clk_driver_en_3 = 1

            # POR On
            self.rfic.qu_reg_en_3 = 1

            # POR On
            self.rfic.qq_reg_en_3 = 1

            # POR Off
            self.rfic.win_rst_3 = 0

            # POR On
            self.rfic.fineEn_3 = 0

            # POR Off
            self.rfic.fineEnb_3 = 1

            # POR Off
            #self.rfic.rsffEn_3 = 0

            # POR On
            self.rfic.dl_en_3 = 1

            # POR On
            self.rfic.cp_en_3 = 1

            self.rfic.set_reg_124()
            self.rfic.set_reg_125()


      def __del__(self):
            # print "rfic_base_rx.__del__"
            # Power down

            # Set RX LNA path (off)
            self.rfic.rx_lna = 0

            # Disable LO clock to mixer
            self.rfic.rx_rxchen = 0

            self.rfic.set_reg_205()

            # Disable RX Filter
            self.rfic.rx_fen = 0

            # Disable baseband filter chipper clock
            self.rfic.rx_chcken = 0
```

```
            # Disable chopper clock to all mixers
            self.rfic.rx_cen = 0

            self.rfic.set_reg_195()

            # Disable filter output
            self.rfic.rx_foe = 0

            # Disable on-channel detector
            self.rfic.rx_onchen = 0

            # Disable off-channel detector
            self.rfic.rx_offchen = 0

            self.rfic.set_reg_196()

            # Disable DCOC DAC
            self.rfic.rx_den = 0

            self.rfic.set_reg_192()

            # Disable DCOC comparator
            self.rfic.rx_cmpen = 0

            self.rfic.set_reg_193()

            # RC Tune disable
            self.rfic.rx_ten = 0

            # RC Tune ramp circuit disable
            self.rfic.rx_ren = 0

            self.rfic.set_reg_194()

            # Disable DCOC
            self.rfic.rx_dcoc = 0

            self.rfic.set_reg_193()

            # POR Off.  This enables the clock that drives the digital block
(which provides the tap selection process).  It must be enabled to generate
an output.  See Byp_fine, address 10, bit 6
            self.rfic.Clk_driver_en_3 = 0

            # POR Off
            self.rfic.qu_reg_en_3 = 0

            # POR Off
            self.rfic.qq_reg_en_3 = 0

            # POR Off
            self.rfic.win_rst_3 = 0

            # POR Off
            self.rfic.fineEn_3 = 0

            # POR Off
```

```
            self.rfic.fineEnb_3 = 0

            # POR Off
            #self.rfic.rsffEn_3 = 0

            # POR Off
            self.rfic.dl_en_3 = 0

            # POR Off
            self.rfic.cp_en_3 = 0

            self.rfic.set_reg_124()
            self.rfic.set_reg_125()

            # Put digital block into digital reset state
            self.rfic.Rst_n_async_3 = 0
            self.rfic.set_reg_58()

            db_rfic_base.__del__(self)

      def select_rx_antenna(self, which_antenna):
            #
            #Specify which antenna port to use for reception.
            #@param which_antenna: either 'LNA1', 'LNA2', 'LNA3', 'LNA4' or
'MIX5'
            #
            if which_antenna in (0, 'LNA1'):
                  self.rfic.rx_lna = 1
                  self.rfic.set_reg_205()
            elif which_antenna in (1, 'LNA2'):
                  self.rfic.rx_lna = 2
                  self.rfic.set_reg_205()
            elif which_antenna in (2, 'LNA3'):
                  self.rfic.rx_lna = 3
                  self.rfic.set_reg_205()
            elif which_antenna in (3, 'LNA4'):
                  self.rfic.rx_lna = 4
                  self.rfic.set_reg_205()
            elif which_antenna in (4, 'MIX5'):
                  self.rfic.rx_lna = 5
                  self.rfic.set_reg_205()
            else:
                  raise ValueError, "which_antenna must be either 'LNA1',
'LNA2', 'LNA3', 'LNA4' or 'MIX5'"

      def gain_range(self):
            # Receiver gain range, in dB
            return (0.0, 38.0, 1)

      def set_gain(self, gain):
            # Set receiver gain, in dB
            return self.rfic.set_rx_gain(gain)

      def set_freq(self, target_freq):
            # Set receiver frequency, in Hz
            return self.rfic.set_rx_freq(target_freq)
```

```
        def set_phase(self, phase):
              # Set receiver phase offset, in degrees
              return self.rfic.set_rx_phase(phase)

        def set_bw(self, bw):
              # Set receiver bandwidth, in Hz
              return self.rfic.set_rx_bw(bw)

        def enable_fb(self):
              # Enable transmitter feedback to receiver for DC offset, etc.
              return self.rfic.enable_tx_fb()

        def disable_fb(self):
              # Disable transmitter feedback to receiver
              return self.rfic.disable_tx_fb()

        def fb_gain_range(self):
              # Feedback gain range, in dB
              # FIXME
              return (0.0, 40.0, 5)

        def set_fb_gain(self, gain):
              # Set feedback gain, in dB
              return self.rfic.set_fb_gain(gain)

        def set_fb_freq(self, target_freq):
              # Set feedback frequency, in Hz
              return self.rfic.set_fb_freq(target_freq)

        def set_fb_phase(self, phase):
              # Set feedback phase offset, in degrees
              return self.rfic.set_fb_phase(phase)

        def set_fb_bw(self, bw):
              # Set feedback bandwidth, in Hz
              return self.rfic.set_fb_bw(bw)

        def RSSI(self):
              # Get received signal strength indicators
              # Returns (fade, clip)
              # Fade is proportional to how often the signal is low
              # Clip is proportional to how often the signal is high
              return self.rfic.RSSI()


#------------------------------------------------------------
# hook these daughterboard classes into the auto-instantiation framework
db_instantiator.add(usrp_dbid.RFIC_TX, lambda usrp, which : (db_rfic_tx(usrp,
which),))
db_instantiator.add(usrp_dbid.RFIC_RX, lambda usrp, which : (db_rfic_rx(usrp,
which),))
```

Appendix B: RF Testing Procedure and Complete Results

Test 1: Noise Floor

Procedure:

1. Turn on HP 8648C Signal Generator (Agilent E4438C for high-speed test). Wait one hour for device to settle, to ensure correct calibration.
2. Boot host computer with GNU Radio 3.0.
3. Plug daughterboard under test into USRP, side A. Ensure that the boards fit together securely and that the daughterboard is seated properly.
4. Using USB 2.0 cable, connect USRP to host computer.
5. Using adapters, if necessary, connect signal generator RF output to RX input on daughterboard, using coax cable.
6. If using RFIC daughterboard, edit daughterboard driver to default to desired RX input. Open terminal and change directories to gnuradio/gr-usrp/src and run "sudo make install" to reinstall driver.
7. On host computer, open terminal and change directories to gnuradio/gnuradio-examples/python/usrp/
8. On host computer, run "usrp_fft.py –d 256 –g <maximum> -f <frequency>" (<maximum> is the maximum gain of the receiver on the daughterboard, <frequency> is the desired frequency – e.g. to test the RFIC at 400 MHz, run "usrp_fft.py –d 256 –g 38 –f 400M"). Change the usrp_fft plot window to take up the whole screen, to see the maximum resolution.
9. Set the frequency of the signal generator to the desired frequency plus 100 kHz (to avoid DC offset, etc. and ensure the signal is easily visible in the usrp_fft plot)
10. Set the function of the signal generator to FM, at 10 kHz. Turn off modulation sources.
11. Set the amplitude of the signal generator to -70 dBm.
12. Turn on RF output of the signal generator.
13. On usrp_fft plot, find signal from signal generator.
14. Adjust amplitude and frequency of signal generator slightly, to make sure the signal you see is the one from the signal generator.
15. Reduce the amplitude of the signal generator until you cannot see the signal anymore on the usrp_fft plot.
16. Record the minimum amplitude, in dBm, from the signal generator, where the signal is visible.
17. Increase the amplitude to 10 dB above the recorded amplitude.
18. Ensure that the signal appears to be 10 dB above the noise floor.
19. If it is not, repeat steps 10 to 17. If it is, this is the noise floor of the daughterboard under test.

Results:

Table 10: Receiver Noise Floor Test, RFIC Input RX1

Receiver Noise Floor Test				
Device Under Test:	RFIC Daughterboard (Input RX1)			
Test Equipment:	HP 8648C Signal Generator, USRP, GNU Radio 3.0			
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)	
			Gain: 38 dB (maximum)	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Noise Floor (dBm):
400	1/2x			-132
900	1x			-132
1800	2x			-130
2400	4x			-116

Table 11: Receiver Noise Floor Test, RFIC Input RX3

Receiver Noise Floor Test				
Device Under Test:	RFIC Daughterboard (Input RX3)			
Test Equipment:	HP 8648C Signal Generator, USRP, GNU Radio 3.0			
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)	
			Gain: 38 dB (maximum)	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Noise Floor (dBm):
400	1/2x			-99
900	1x			-102
1800	2x			-102
2400	4x			-92

Table 12: Receiver Noise Floor Test, RFIC Input MIX5

Receiver Noise Floor Test				
Device Under Test:	RFIC Daughterboard (Input MIX5)			
Test Equipment:	HP 8648C Signal Generator, USRP, GNU Radio 3.0			
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)	
			Gain: 38 dB (maximum)	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Noise Floor (dBm):
400	1/2x			-118
900	1x			-118
1800	2x			-118
2400	4x			-101

Table 13: High-Frequency Receiver Noise Floor Test, RFIC Input RX1

High-Frequency Receiver Noise Floor Test				
Device Under Test:	RFIC Daughterboard (Input RX1)			
Test Equipment:	Agilent E4458C Signal Generator, USRP, GNU Radio 3.0			
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)	
			Gain: 38 dB (maximum)	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Noise Floor (dBm):
3000	4x			-109
3500	4x			-112
4000	4x			-101

Table 14: Receiver Noise Floor Test, RFX-Series

Receiver Noise Floor Test				
Device Under Test:	RFX-Series Daughterboards			
Test Equipment:	HP 8648C Signal Generator, USRP, GNU Radio 3.0			
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)	
			Gain: maximum	
Frequency (MHz):	RFX-Series Model			Noise Floor (dBm):
400	RFX400 (max. gain: 65 dB)			-135
900	RFX900 (max. gain: 90 dB)			-126
1800	RFX1800 (max. gain: 90 dB)			-116
2400	RFX2400 (max. gain: 90 dB)			-105

Test 2: IIP3

Procedure:

1. Turn on both HP 8648C Signal Generators. Wait one hour for device to settle, to ensure correct calibration.
2. Turn on HP 8594E Spectrum Analyzer. Wait one hour for device to settle, to ensure correct calibration.
3. Set spectrum analyzer to desired frequency, with a span of 1 MHz.
4. Boot host computer with GNU Radio 3.0.
5. Plug daughterboard under test into USRP, side A. Ensure that the boards fit together securely and that the daughterboard is seated properly.
6. Using USB 2.0 cable, connect USRP to host computer.
7. If using RFIC daughterboard, edit daughterboard driver to default to desired RX input. Open terminal and change directories to gnuradio/gr-usrp/src and run "sudo make install" to reinstall driver.
8. Using adapters, if necessary, and a T-connector, connect both signal generator RF outputs to RX input on daughterboard, using coax cable.
9. On host computer, open terminal and change directories to gnuradio/gnuradio-examples/python/usrp/
10. On host computer, run "usrp_fft.py –d 256 –g 0 -f <frequency>" (<frequency> is the desired frequency – e.g. to test the RFIC at 400 MHz, run "usrp_fft.py –d 256 –g 0 –f 400M"). Change the usrp_fft plot window to take up the whole screen, to see the maximum resolution.
11. Set the frequency of the signal generators to near the desired frequency, 20 kHz apart. E.g., if the desired frequency is 400 MHz, set one signal generator to 400.1 MHz and the other to 400.12 MHz.
12. Set the function of the signal generators to FM, at 10 kHz. Turn off modulation sources.
13. Set the amplitude of the signal generators to -70 dBm.
14. Turn on RF output of the signal generators.
15. Ensure that both signals are visible in usrp_fft plot.
16. Increase amplitude of both signal generators until 3rd-harmonic is clearly visible in usrp_fft plot. Change frequencies of signal generators, if necessary, making sure to keep signals 20 kHz apart, to see 3rd-harmonic.
17. Turn off and on RF output of both signal generators, one at a time, while watching usrp_fft plot to ensure that the signal in question is a harmonic of both signals. Turn both RF outputs back on.
18. Adjust frequency of one signal generator to ensure that the signal in question is at the correct frequency (20 kHz from one of the signal generator signals).
19. Turn off RF outputs of both signal generators.
20. Unplug signal generators from daughterboard.
21. Plug signal generators into spectrum analyzer, using same T-connector and coax cable.
22. Turn on RF outputs of both signal generators. View the signals on the spectrum analyzer, adjusting the amplitude of the spectrum analyzer if necessary. Adjust amplitudes of signal generators until they are equal. Record this amplitude, in dBm, as P_{IN}. This step records the actual input power – the power displayed by the signal generators will not be accurate due to losses in the T-connector.

23. Turn off RF outputs of both signal generators.
24. Unplug both signal generators from spectrum analyzer.
25. Plug both signal generators into daughterboard.
26. Turn on RF outputs of both signal generators.
27. Locate 3rd harmonic on usrp_fft plot.
28. Adjust frequency and amplitude of one signal generator to match those of the 3rd-harmonic.
29. Turn off RF outputs of both signal generators.
30. Unplug both signal generators from daughterboard.
31. Plug both signal generators into spectrum analyzer.
32. Turn on RF output of signal generator at desired frequency and amplitude of 3rd-harmonic.
33. Record this amplitude, in dBm, on the spectrum analyzer as P_3.
34. Use the following formula to solve for P_{IIP3}. This is the IIP3 of the receiver daughterboard.

$$P_{IN} + x = P_3 + 3 * x = P_{IIP3}$$

Results:

Table 15: Receiver IIP3 Test, RFIC Input RX1

<table>
<tr><td colspan="5">Receiver IIP3 Test</td></tr>
<tr><td>Device Under Test:</td><td colspan="4">RFIC Daughterboard (Input RX1)</td></tr>
<tr><td>Test Equipment:</td><td colspan="4">HP 8648C Signal Generator (x2), USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer</td></tr>
<tr><td rowspan="3">Spectrum Analyzer Program:</td><td rowspan="3">usrp_fft.py</td><td rowspan="3">Settings:</td><td colspan="2">Decimation Rate: 256 (maximum)</td></tr>
<tr><td colspan="2">Gain: 0 dB</td></tr>
<tr><td colspan="2">Frequency Offset: 20 kHz</td></tr>
<tr><td>Frequency (MHz):</td><td colspan="3">Mode (synthesizer frequency multiplier):</td><td>IIP3 (dBm):</td></tr>
<tr><td>400</td><td colspan="3">1/2x</td><td>-2.4</td></tr>
<tr><td>900</td><td colspan="3">1x</td><td>-4.8</td></tr>
<tr><td>1800</td><td colspan="3">2x</td><td>-2.7</td></tr>
<tr><td>2400</td><td colspan="3">4x</td><td>1.3</td></tr>
</table>

Table 16: Receiver IIP3 Test, RFIC Input RX3

Receiver IIP3 Test			
Device Under Test:	RFIC Daughterboard (Input RX3)		
Test Equipment:	HP 8648C Signal Generator (x2), USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer		
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)
			Gain: 0 dB
			Frequency Offset: 20 kHz
Frequency (MHz):	Mode (synthesizer frequency multiplier):		IIP3 (dBm):
400	1/2x		-15.3
900	1x		-15.1
1800	2x		-14.2
2400	4x		-7.1

Table 17: Receiver IIP3 Test, RFIC Input MIX5

Receiver IIP3 Test			
Device Under Test:	RFIC Daughterboard (Input MIX5)		
Test Equipment:	HP 8648C Signal Generator (x2), USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer		
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)
			Gain: 0 dB
			Frequency Offset: 20 kHz
Frequency (MHz):	Mode (synthesizer frequency multiplier):		IIP3 (dBm):
400	1/2x		-32.0
900	1x		-24.8
1800	2x		-24.8
2400	4x		-18.2

Table 18: Receiver IIP3 Test, RFX-Series

<table>
<tr><td colspan="5">Receiver IIP3 Test</td></tr>
<tr><td>Device Under Test:</td><td colspan="4">RFX-Series Daughterboards</td></tr>
<tr><td>Test Equipment:</td><td colspan="4">HP 8648C Signal Generator (x2), USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer</td></tr>
<tr><td rowspan="3">Spectrum Analyzer Program:</td><td rowspan="3">usrp_fft.py</td><td rowspan="3">Settings:</td><td colspan="2">Decimation Rate: 256 (maximum)</td></tr>
<tr><td colspan="2">Gain: 0 dB</td></tr>
<tr><td colspan="2">Frequency Offset: 20 kHz</td></tr>
<tr><td>Frequency (MHz):</td><td colspan="3">RFX-Series Model:</td><td>IIP3 (dBm):</td></tr>
<tr><td>400</td><td colspan="3">RFX400</td><td>0.8</td></tr>
<tr><td>900</td><td colspan="3">RFX900</td><td>0.5</td></tr>
<tr><td>1800</td><td colspan="3">RFX1800</td><td>-4.6</td></tr>
<tr><td>2400</td><td colspan="3">RFX2400</td><td>1.0</td></tr>
</table>

Test 3: IIP2

Procedure:

1. Turn on both HP 8648C Signal Generators. Wait one hour for device to settle, to ensure correct calibration.
2. Turn on HP 8594E Spectrum Analyzer. Wait one hour for device to settle, to ensure correct calibration.
3. Set spectrum analyzer to desired frequency divided by 2, with a span of 2 MHz.
4. Boot host computer with GNU Radio 3.0.
5. Plug daughterboard under test into USRP, side A. Ensure that the boards fit together securely and that the daughterboard is seated properly.
6. Using USB 2.0 cable, connect USRP to host computer.
7. If using RFIC daughterboard, edit daughterboard driver to default to desired RX input. Open terminal and change directories to gnuradio/gr-usrp/src and run "sudo make install" to reinstall driver.
8. Using adapters, if necessary, and a T-connector, connect both signal generator RF outputs to RX input on daughterboard, using coax cable.
9. On host computer, open terminal and change directories to gnuradio/gnuradio-examples/python/usrp/
10. On host computer, run "usrp_fft.py –d 256 –g 0 -f <frequency>" (<frequency> is the desired frequency – e.g. to test the RFIC at 400 MHz, run "usrp_fft.py –d 256 –g 0 –f 400M"). Change the usrp_fft plot window to take up the whole screen, to see the maximum resolution.
11. Set the frequency of the signal generators to half the desired frequency, plus 550 kHz and minus 450 kHz. E.g., if the desired frequency is 400 MHz, set one signal generator to 199.55 MHz and the other to 200.55 MHz.
12. Set the function of the signal generators to FM, at 10 kHz. Turn off modulation sources.
13. Set the amplitude of the signal generators to -70 dBm.
14. Turn on RF output of the signal generators.
15. Increase amplitude of both signal generators until 2^{nd}-harmonic is clearly visible in usrp_fft plot. Change frequencies of signal generators, if necessary, making sure to keep signals 1 MHz apart, to see 2^{nd}-harmonic.
16. Turn off and on RF output of both signal generators, one at a time, while watching usrp_fft plot to ensure that the signal in question is a harmonic of both signals. Turn both RF outputs back on.
17. Adjust frequency of one signal generator to ensure that the signal in question is at the correct frequency (the sum of the frequencies of the two signal generators).
18. Turn off RF outputs of both signal generators.
19. Unplug signal generators from daughterboard.
20. Plug signal generators into spectrum analyzer, using same T-connector and coax cable.
21. Turn on RF outputs of both signal generators. View the signals on the spectrum analyzer, adjusting the amplitude of the spectrum analyzer if necessary. Adjust amplitudes of signal generators until they are equal. Record this amplitude, in dBm, as P_{IN}. This step records the actual input power – the power displayed by the signal generators will not be accurate due to losses in the T-connector.
22. Turn off RF outputs of both signal generators.

23. Unplug both signal generators from spectrum analyzer.
24. Plug both signal generators into daughterboard.
25. Turn on RF outputs of both signal generators.
26. Locate 2nd harmonic on usrp_fft plot.
27. Adjust frequency and amplitude of one signal generator to match those of the 2nd-harmonic.
28. Turn off RF outputs of both signal generators.
29. Unplug both signal generators from daughterboard.
30. Plug both signal generators into spectrum analyzer.
31. Turn on RF output of signal generator at desired frequency and amplitude of 2nd-harmonic.
32. Record this amplitude, in dBm, on the spectrum analyzer as P_2.
33. Use the following formula to solve for P_{IIP2}. This is the IIP2 of the receiver daughterboard.

$$P_{IN} + x = P_2 + 2 * x = P_{IIP2}$$

Results:

Table 19: Receiver IIP2 Test, RFIC Input RX1

Receiver IIP2 Test				
Device Under Test:	RFIC Daughterboard (Input RX1)			
Test Equipment:	HP 8648C Signal Generator (x2), USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)	
			Gain: 0 dB	
			Frequency Offset: 1 MHz	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			IIP2 (dBm):
400	1/2x			18.3
900	1x			15.6
1800	2x			14.2
2400	4x			6.1

Table 20: Receiver IIP2 Test, RFIC Input RX3

Receiver IIP2 Test				
Device Under Test:	RFIC Daughterboard (Input RX3)			
Test Equipment:	HP 8648C Signal Generator (x2), USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)	
			Gain: 0 dB	
			Frequency Offset: 1 MHz	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			IIP2 (dBm):
400	1/2x			60.9
900	1x			47.2
1800	2x			45.6
2400	4x			29.4

Table 21: Receiver IIP2 Test, RFIC Input MIX5

Receiver IIP2 Test				
Device Under Test:	RFIC Daughterboard (Input MIX5)			
Test Equipment:	HP 8648C Signal Generator (x2), USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Spectrum Analyzer Program:	usrp_fft.py	Settings:	Decimation Rate: 256 (maximum)	
			Gain: 0 dB	
			Frequency Offset: 1 MHz	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			IIP2 (dBm):
400	1/2x			24.8
900	1x			17.8
1800	2x			18.8
2400	4x			12.6

Table 22: Receiver IIP2 Test, RFX-Series

<table>
<tr><td colspan="5">Receiver IIP2 Test</td></tr>
<tr><td>Device Under Test:</td><td colspan="4">RFX-Series Daughterboards</td></tr>
<tr><td>Test Equipment:</td><td colspan="4">HP 8648C Signal Generator (x2), USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer</td></tr>
<tr><td rowspan="3">Spectrum Analyzer Program:</td><td rowspan="3">usrp_fft.py</td><td rowspan="3">Settings:</td><td colspan="2">Decimation Rate: 256 (maximum)</td></tr>
<tr><td colspan="2">Gain: 0 dB</td></tr>
<tr><td colspan="2">Frequency Offset: 1 MHz</td></tr>
<tr><td>Frequency (MHz):</td><td colspan="3">RFX-Series Model:</td><td>IIP2 (dBm):</td></tr>
<tr><td>400</td><td colspan="3">RFX400</td><td>8.6</td></tr>
<tr><td>900</td><td colspan="3">RFX900</td><td>57.8</td></tr>
<tr><td>1800</td><td colspan="3">RFX1800</td><td>16.8</td></tr>
<tr><td>2400</td><td colspan="3">RFX2400</td><td>62.0</td></tr>
</table>

Test 4: Transmitter Output Power

Procedure:

1. Turn on HP 8594E Spectrum Analyzer. Wait one hour for device to settle, to ensure correct calibration.
2. Set spectrum analyzer to desired frequency, with a span of 1 MHz.
3. Boot host computer with GNU Radio 3.0.
4. Plug daughterboard under test into USRP, side A. Ensure that the boards fit together securely and that the daughterboard is seated properly.
5. Using USB 2.0 cable, connect USRP to host computer.
6. If using RFIC daughterboard, edit daughterboard driver to default to desired TX output. Open terminal and change directories to gnuradio/gr-usrp/src and run "sudo make install" to reinstall driver.
7. Using adapters, if necessary, connect desired TX output on daughterboard to RF input on spectrum analyzer, using coax cable.
8. On host computer, open terminal and change directories to gnuradio/gnuradio-examples/python/usrp/
9. On host computer, run "usrp_siggen.py -f <frequency>". If testing the RFIC, run "usrp_siggen_rfic.py –f <frequency>" (<frequency> is the desired frequency – e.g. to test the RFIC at 400 MHz, run "usrp_siggen_rfic.py –f 400M"). Usrp_siggen_rfic.py is the same program as usrp_siggen.py, except that GNU Radio has been forced to recognize the RFIC daughterboard in both transmitter slots.
10. Adjust amplitude on spectrum analyzer, if necessary, to see signal.
11. Find transmitted signal, 100 kHz above the desired frequency, on the spectrum analyzer. Record the amplitude of this signal in dBm. This is the transmitter output power.

Results:

Table 23: Transmitter Power Test, RFIC Output TX1

Transmitter Power Test				
Device Under Test:	RFIC Daughterboard (Output TX1)			
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)	
			Gain: 45 dB (maximum)	
			Waveform: Complex Sinusoid	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Output Power (dBm)
400	1/2x			6.2
900	1x			0.1
1800	2x			-2.0
2400	4x			-15.0

Table 24: Transmitter Power Test, RFIC Output TX2

Transmitter Power Test				
Device Under Test:	RFIC Daughterboard (Output TX2)			
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)	
			Gain: 45 dB (maximum)	
			Waveform: Complex Sinusoid	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Output Power (dBm)
400	1/2x			7.9
900	1x			3.2
1800	2x			-3.0
2400	4x			-13.0

Table 25: Transmitter Power Test, RFX-Series

Transmitter Power Test				
Device Under Test:	RFX-Series Daughterboards			
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)	
			Gain: 45 dB (maximum)	
			Waveform: Complex Sinusoid	
Frequency (MHz):	RFX-Series Model:			Output Power (dBm)
400	RFX400			22.6
900	RFX900			22.0
1800	RFX1800			20.8
2400	RFX2400			12.3

Test 5: Transmitter LO Suppression

Procedure:

1. Turn on HP 8594E Spectrum Analyzer. Wait one hour for device to settle, to ensure correct calibration.
2. Set spectrum analyzer to desired frequency, with a span of 1 MHz.
3. Boot host computer with GNU Radio 3.0.
4. Plug daughterboard under test into USRP, side A. Ensure that the boards fit together securely and that the daughterboard is seated properly.
5. Using USB 2.0 cable, connect USRP to host computer.
6. If using RFIC daughterboard, edit daughterboard driver to default to desired TX output. Open terminal and change directories to gnuradio/gr-usrp/src and run "sudo make install" to reinstall driver.
7. Using adapters, if necessary, connect desired TX output on daughterboard to RF input on spectrum analyzer, using coax cable.
8. On host computer, open terminal and change directories to gnuradio/gnuradio-examples/python/usrp/
9. On host computer, run "usrp_siggen.py -f <frequency> -w 200000". If testing the RFIC, run "usrp_siggen_rfic.py –f <frequency>" (<frequency> is the desired frequency – e.g. to test the RFIC at 400 MHz, run "usrp_siggen_rfic.py –f 400M"). Usrp_siggen_rfic.py is the same program as usrp_siggen.py, except that GNU Radio has been forced to recognize the RFIC daughterboard in both transmitter slots.
10. Adjust amplitude on spectrum analyzer, if necessary, to see signal.
11. Find transmitted signal, 200 kHz above the specified RF frequency, on the spectrum analyzer. Record the amplitude of this signal in dBm. This is the transmitter output power, P_{OUT}.
12. Find LO, at desired frequency, on the spectrum analyzer. Record the amplitude of this signal in dBm. This is the LO power, P_{LO}.
13. Using the equation below, calculate S_C, the LO suppression.

$$S_C = P_{OUT} - P_{LO}$$

Results:

Table 26: Transmitter LO Suppression Test, RFIC Output TX1

Transmitter LO Suppression Test			
Device Under Test:	RFIC Daughterboard (Output TX1)		
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer		
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)
			Gain: 45 dB (maximum)
			Waveform: Complex Sinusoid
			Waveform Frequency: 200 kHz

Frequency (MHz):	Mode (synthesizer frequency multiplier):	Suppression (dBc):
400	1/2x	34.1
900	1x	27.0
1800	2x	27.9
2400	4x	24.9

Table 27: Transmitter LO Suppression Test, RFIC Output TX2

Transmitter LO Suppression Test			
Device Under Test:	RFIC Daughterboard (Output TX2)		
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer		
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)
			Gain: 45 dB (maximum)
			Waveform: Complex Sinusoid
			Waveform Frequency: 200 kHz

Frequency (MHz):	Mode (synthesizer frequency multiplier):	Suppression (dBc):
400	1/2x	31.3
900	1x	26.6
1800	2x	14.6
2400	4x	18.7

Table 28: Transmitter LO Suppression Test, RFX-Series

Transmitter LO Suppression Test				
Device Under Test:	RFX-Series Daughterboards			
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)	
			Gain: 45 dB (maximum)	
			Waveform: Complex Sinusoid	
			Waveform Frequency: 200 kHz	
Frequency (MHz):	RFX-Series Model:			Suppression (dBc):
400	RFX400			41.8
900	RFX900			50.3
1800	RFX1800			43.2
2400	RFX2400			36.1

Test 6: Transmitter 2nd-Order Harmonic Suppression

Procedure:

1. Turn on HP 8594E Spectrum Analyzer. Wait one hour for device to settle, to ensure correct calibration.
2. Set spectrum analyzer to desired frequency, with a span of 1 MHz.
3. Boot host computer with GNU Radio 3.0.
4. Plug daughterboard under test into USRP, side A. Ensure that the boards fit together securely and that the daughterboard is seated properly.
5. Using USB 2.0 cable, connect USRP to host computer.
6. If using RFIC daughterboard, edit daughterboard driver to default to desired TX output. Open terminal and change directories to gnuradio/gr-usrp/src and run "sudo make install" to reinstall driver.
7. Using adapters, if necessary, connect desired TX output on daughterboard to RF input on spectrum analyzer, using coax cable.
8. On host computer, open terminal and change directories to gnuradio/gnuradio-examples/python/usrp/
9. On host computer, run "usrp_siggen.py -f <frequency> -w 200000". If testing the RFIC, run "usrp_siggen_rfic.py –f <frequency>" (<frequency> is the desired frequency – e.g. to test the RFIC at 400 MHz, run "usrp_siggen_rfic.py –f 400M"). Usrp_siggen_rfic.py is the same program as usrp_siggen.py, except that GNU Radio has been forced to recognize the RFIC daughterboard in both transmitter slots.
10. Adjust amplitude on spectrum analyzer, if necessary, to see signal.
11. Find transmitted signal, 200 kHz above the specified RF frequency, on the spectrum analyzer. Record the amplitude of this signal in dBm. This is the transmitter output power, P_{OUT}.
12. Set the spectrum analyzer to twice the desired frequency. E.g., if the desired frequency is 400 MHz, set the spectrum analyzer to 800 MHz.
13. Find 2nd-order harmonic, near twice the desired frequency, on the spectrum analyzer. Record the amplitude of this signal in dBm. This is the 2nd-order harmonic power, P_{2H}.
14. Using the equation below, calculate S_{2H}, the 2nd-harmonic suppression.

$$S_{2H} = P_{OUT} - P_{2H}$$

Results:

Table 29: Transmitter 2nd-Order Harmonic Suppression Test., RFIC Output TX1

Transmitter 2nd-Order Harmonic Suppression Test				
Device Under Test:	RFIC Daughterboard (Output TX1)			
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)	
			Gain: 45 dB (maximum)	
			Waveform: Complex Sinusoid	
			Waveform Frequency: 200 kHz	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Suppression (dBc):
400	1/2x			22.8
900	1x			23.2

Table 30: Transmitter 2nd-Order Harmonic Suppression Test, RFIC Output TX2

Transmitter 2nd-Order Harmonic Suppression Test				
Device Under Test:	RFIC Daughterboard (Output TX2)			
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)	
			Gain: 45 dB (maximum)	
			Waveform: Complex Sinusoid	
			Waveform Frequency: 200 kHz	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Suppression (dBc):
400	1/2x			19.6
900	1x			22.6

Table 31: Transmitter 2nd-Order Harmonic Suppression Test, RFX-Series

Transmitter 2nd-Order Harmonic Suppression Test			
Device Under Test:	RFX-Series Daughterboards		
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer		
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)
			Gain: 45 dB (maximum)
			Waveform: Complex Sinusoid
			Waveform Frequency: 200 kHz

Frequency (MHz):	RFX-Series Model	Suppression (dBc):
400	RFX400	34.7
900	RFX900	38.8

Test 7: Transmitter 3rd-Order Harmonic Suppression

Procedure:

15. Turn on HP 8594E Spectrum Analyzer. Wait one hour for device to settle, to ensure correct calibration.
16. Set spectrum analyzer to desired frequency, with a span of 1 MHz.
17. Boot host computer with GNU Radio 3.0.
18. Plug daughterboard under test into USRP, side A. Ensure that the boards fit together securely and that the daughterboard is seated properly.
19. Using USB 2.0 cable, connect USRP to host computer.
20. If using RFIC daughterboard, edit daughterboard driver to default to desired TX output. Open terminal and change directories to gnuradio/gr-usrp/src and run "sudo make install" to reinstall driver.
21. Using adapters, if necessary, connect desired TX output on daughterboard to RF input on spectrum analyzer, using coax cable.
22. On host computer, open terminal and change directories to gnuradio/gnuradio-examples/python/usrp/
23. On host computer, run "usrp_siggen.py -f <frequency> -w 200000". If testing the RFIC, run "usrp_siggen_rfic.py –f <frequency>" (<frequency> is the desired frequency – e.g. to test the RFIC at 400 MHz, run "usrp_siggen_rfic.py –f 400M"). Usrp_siggen_rfic.py is the same program as usrp_siggen.py, except that GNU Radio has been forced to recognize the RFIC daughterboard in both transmitter slots.
24. Adjust amplitude on spectrum analyzer, if necessary, to see signal.
25. Find transmitted signal, 200 kHz above the specified RF frequency, on the spectrum analyzer. Record the amplitude of this signal in dBm. This is the transmitter output power, P_{OUT}.
26. Set the spectrum analyzer to three times the desired frequency. E.g., if the desired frequency is 400 MHz, set the spectrum analyzer to 1200 MHz.
27. Find 3rd-order harmonic, near three times the desired frequency, on the spectrum analyzer. Record the amplitude of this signal in dBm. This is the 3rd-order harmonic power, P_{3H}.
28. Using the equation below, calculate S_{3H}, the 3rd-harmonic suppression.

$$S_{3H} = P_{OUT} - P_{3H}$$

Results:

Table 32 Transmitter 3rd-Order Harmonic Suppression Test, RFIC Output TX1

Transmitter 3rd-Order Harmonic Suppression Test				
Device Under Test:	RFIC Daughterboard (Output TX1)			
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)	
			Gain: 45 dB (maximum)	
			Waveform: Complex Sinusoid	
			Waveform Frequency: 200 kHz	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Suppression (dBc):
400	1/2x			18.7
900	1x			22.6

Table 33: Transmitter 3rd-Order Harmonic Suppression Test, RFIC Output TX2

Transmitter 3rd-Order Harmonic Suppression Test				
Device Under Test:	RFIC Daughterboard (Output TX2)			
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer			
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)	
			Gain: 45 dB (maximum)	
			Waveform: Complex Sinusoid	
			Waveform Frequency: 200 kHz	
Frequency (MHz):	Mode (synthesizer frequency multiplier):			Suppression (dBc):
400	1/2x			19.0
900	1x			26.7

Table 34: Transmitter 3rd-Order Harmonic Suppression Test, RFX-Series

Transmitter 3rd-Order Harmonic Suppression Test			
Device Under Test:	RFX-Series Daughterboards		
Test Equipment:	USRP, GNU Radio 3.0, HP 8594E Spectrum Analyzer		
Signal Generator Program	usrp_siggen.py	Settings:	Signal Amplitude: 16000 (digital)
			Gain: 45 dB (maximum)
			Waveform: Complex Sinusoid
			Waveform Frequency: 200 kHz

Frequency (MHz):	RFX-Series Model:	Suppression (dBc):
400	RFX400	48.4
900	RFX900	41.7

Appendix C: usrp_siggen_rfic.py

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

from gnuradio import gr, gru
from gnuradio import usrp
from gnuradio.eng_option import eng_option
from gnuradio import eng_notation
from optparse import OptionParser
import sys


class my_graph(gr.flow_graph):
    def __init__ (self):
        gr.flow_graph.__init__(self)

        # controllable values
        self.interp = 64
        self.waveform_type = gr.GR_SIN_WAVE
        self.waveform_ampl = 16000
        self.waveform_freq = 100.12345e3
        self.waveform_offset = 0
        self._instantiate_blocks ()
        self.set_waveform_type (self.waveform_type)

    def usb_freq (self):
        return self.u.dac_freq() / self.interp

    def usb_throughput (self):
        return self.usb_freq () * 4

    def set_waveform_type (self, type):
        '''
        valid waveform types are: gr.GR_SIN_WAVE, gr.GR_CONST_WAVE,
        gr.GR_UNIFORM and gr.GR_GAUSSIAN
        '''
        self._configure_graph (type)
        self.waveform_type = type

    def set_waveform_ampl (self, ampl):
        self.waveform_ampl = ampl
        self.siggen.set_amplitude (ampl)
        self.noisegen.set_amplitude (ampl)

    def set_waveform_freq (self, freq):
        self.waveform_freq = freq
        self.siggen.set_frequency (freq)

    def set_waveform_offset (self, offset):
        self.waveform_offset = offset
        self.siggen.set_offset (offset)

    def set_interpolator (self, interp):
        self.interp = interp
        self.siggen.set_sampling_freq (self.usb_freq ())
        self.u.set_interp_rate (interp)
```

```
    def _instantiate_blocks (self):
        self.src = None
        self.u = usrp.sink_c (0, self.interp)
      #self.u = usrp.usrp1.sink_c(0, self.interp, 1, 0x98, 0, 0, "", "")

      # This line forces GNU Radio to recognize the RFIC daughterboard on
both sides A and B of the attached USRP
      self.u.db = (usrp.db_instantiator._instantiator_map[160](self.u, 0),
usrp.db_instantiator._instantiator_map[160](self.u, 1))
      #

        self.siggen = gr.sig_source_c (self.usb_freq (),
                                       gr.GR_SIN_WAVE,
                                       self.waveform_freq,
                                       self.waveform_ampl,
                                       self.waveform_offset)

        self.noisegen = gr.noise_source_c (gr.GR_UNIFORM,
                                           self.waveform_ampl)

        # self.file_sink = gr.file_sink (gr.sizeof_gr_complex, "siggen.dat")

    def _configure_graph (self, type):
        was_running = self.is_running ()
        if was_running:
            self.stop ()
        self.disconnect_all ()
        if type == gr.GR_SIN_WAVE or type == gr.GR_CONST_WAVE:
            self.connect (self.siggen, self.u)
            # self.connect (self.siggen, self.file_sink)
            self.siggen.set_waveform (type)
            self.src = self.siggen
        elif type == gr.GR_UNIFORM or type == gr.GR_GAUSSIAN:
            self.connect (self.noisegen, self.u)
            self.noisegen.set_type (type)
            self.src = self.noisegen
        else:
            raise ValueError, type
        if was_running:
            self.start ()

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

        @param target_freq: frequency in Hz
        @rypte: bool

        Tuning is a two step process.  First we ask the front-end to
        tune as close to the desired frequency as it can.  Then we use
        the result of that operation and our target_frequency to
        determine the value for the digital up converter.
        """
        r = self.u.tune(self.subdev._which, self.subdev, target_freq)
        if r:
```

```
            #print "r.baseband_freq =",
eng_notation.num_to_str(r.baseband_freq)
            #print "r.dxc_freq      =", eng_notation.num_to_str(r.dxc_freq)
            #print "r.residual_freq =",
eng_notation.num_to_str(r.residual_freq)
            #print "r.inverted      =", r.inverted
            return True

        return False



def main ():
    parser = OptionParser (option_class=eng_option)
    parser.add_option ("-T", "--tx-subdev-spec", type="subdev", default=(0,
0),
                       help="select USRP Tx side A or B")
    parser.add_option ("-f", "--rf-freq", type="eng_float", default=None,
                       help="set RF center frequency to FREQ")
    parser.add_option ("-i", "--interp", type="int", default=64,
                       help="set fgpa interpolation rate to INTERP
[default=%default]")

    parser.add_option ("--sine", dest="type", action="store_const",
const=gr.GR_SIN_WAVE,
                       help="generate a complex sinusoid [default]",
default=gr.GR_SIN_WAVE)
    parser.add_option ("--const", dest="type", action="store_const",
const=gr.GR_CONST_WAVE,
                       help="generate a constant output")
    parser.add_option ("--gaussian", dest="type", action="store_const",
const=gr.GR_GAUSSIAN,
                       help="generate Gaussian random output")
    parser.add_option ("--uniform", dest="type", action="store_const",
const=gr.GR_UNIFORM,
                       help="generate Uniform random output")

    parser.add_option ("-w", "--waveform-freq", type="eng_float",
default=100e3,
                       help="set waveform frequency to FREQ
[default=%default]")
    parser.add_option ("-a", "--amplitude", type="eng_float", default=16e3,
                       help="set waveform amplitude to AMPLITUDE
[default=%default]", metavar="AMPL")
    parser.add_option ("-o", "--offset", type="eng_float", default=0,
                       help="set waveform offset to OFFSET
[default=%default]")
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help()
        raise SystemExit

    if options.rf_freq is None:
        sys.stderr.write("usrp_siggen: must specify RF center frequency with
-f RF_FREQ\n")
        parser.print_help()
```

```
        raise SystemExit

    fg = my_graph()
    fg.set_interpolator (options.interp)
    fg.set_waveform_type (options.type)
    fg.set_waveform_freq (options.waveform_freq)
    fg.set_waveform_ampl (options.amplitude)
    fg.set_waveform_offset (options.offset)

    # determine the daughterboard subdevice we're using
    if options.tx_subdev_spec is None:
        #options.tx_subdev_spec = usrp.pick_tx_subdevice(fg.u)
      options.tx_subdev_spec = (0, 0)

    m = usrp.determine_tx_mux_value(fg.u, options.tx_subdev_spec)
    print "mux = %#04x" % (m,)
    fg.u.set_mux(m)
    fg.subdev = usrp.selected_subdev(fg.u, options.tx_subdev_spec)
    print "Using TX d'board %s" % (fg.subdev.side_and_name(),)

    fg.subdev.set_gain(fg.subdev.gain_range()[1])    # set max Tx gain

    if not fg.set_freq(options.rf_freq):
        sys.stderr.write('Failed to set RF frequency\n')
        raise SystemExit

    fg.subdev.set_enable(True)                       # enable transmitter

    try:
        fg.run()
    except KeyboardInterrupt:
        pass

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

Appendix D: Permission from Matt Ettus

From: Matt Ettus [matt@ettus.com]
Sent: Monday, March 09, 2009 1:11 AM
To: tbrisebo@vt.edu
Cc: Randall Nealy
Subject: Re: RFIC-based USRP Daughterboard

tbrisebo@vt.edu wrote:
[…]

> On another subject, my master's thesis is on the RFIC-based
> daughterboard. I would like to ask your permission to use a few
> photos of the USRP and daughterboards in my thesis. I would like to
> use the photos below and maybe a few others.
> http://www.ettus.com/images/USRP.jpg
> http://www.ettus.com/images/Flex400.jpg

No problem. Feel free to use any of the pictures or diagrams on either gnuradio.org or ettus.com

Thanks,
Matt

Bibliography

[1] M. Ettus. Ettus Research LLC. Universal Software Radio Peripheral The Foundation for Complete Software Radio Systems. [Online]. Available: http://www.ettus.com/downloads/er_ds_usrp_v5b.pdf

[2] M. Ettus. Ettus Research LLC. Transciever Daughterboards for the USRP Software Radio System. [Online]. Available: http://www.ettus.com/download/er_ds_transciever_dbrds_v5b.pdf

[3] M. Ettus. Ettus Research LLC. USRP2 The Next Generation of Software Radio Systems. [Online]. Available: http://www.ettus.com/downloads/ettus_ds_usrp2_v2.pdf

[4] GNU Radio. USRP Daughterboard: RFX400. [Online]. Available: http://gnuradio.org/trac/wiki/UsrpDBoardRFX400

[5] G. Carafo, T. Gradishar, J. Heck, S. Machan, G. Nagaraj, S. Olson, R. Salvi, B. Stengel, B. Ziemer, "A 100 MHz – 2.5 GHz Direct Conversion CMOS Transceiver for SDR Applications," in *2007 IEEE Radio Frequency Integrated Circuits Symposium*, 2007.

[6] GNU Radio. (2009). GNU Radio Homepage. [Online]. Available: http://gnuradio.org/trac/wiki

[7] GNU Radio. (2007). Usrp_wfm_rcv_nogui.py. [Online]. Available: http://gnuradio.org/trac/browser/gnuradio/tags/releases/3.1.3/gnuradio-examples/python/usrp/usrp_wfm_rcv_nogui.py

[8] F. Abbas. (2009, Jun.). The USRP under 1.5X Magnifiying Lens! [Online]. Available: http://gnuradio.org/trac/attachment/wiki/UsrpFAQ/USRP_Documentation.pdf

[9] M. Ettus. Ettus Research LLC. USRP User's and Developer's Guide. [Online]. Available: http://www.olifantasia.com/gnuradio/usrp/files/usrp_guide.pdf

[10] F. Ge, A. Young, T. Brisebois, Q. Chen, C. Bostian, "Software Defined Radio Execution Latency," in *SDR '08 Technical Conference and Product Exposition*, 2008.

[11] Analog Devices. (2005). 140 MHz to 1000 MHz Quadrature Modulator AD8345. [Online]. Available: http://www.analog.com/static/imported-files/data_sheets/AD8345.pdf

[12] M. Ettus. Ettus Research LLC. (2005, Oct.). Flex400 Common. [Online]. Available: http://gnuradio.org/trac/browser/usrp-hw/trunk/rfx/common400.ps

[13] M. Ettus. Ettus Research LLC. (2005, Oct.). Flex400 Upconverter. [Online]. Available: http://gnuradio.org/trac/browser/usrp-hw/trunk/rfx/trans400.ps

[14] M. Ettus. Ettus Research LLC. (2005, Oct.). Flex400 Downconverter. [Online]. Available: http://gnuradio.org/trac/browser/usrp-hw/trunk/rfx/rcv400.ps

[15] Analog Devices. (2004). Integrated Synthesizer and VCO ADF4360-0. [Online]. Available: http://www.analog.com/static/imported-files/data_sheets/ADF4360-0.pdf

[16] GNU Radio. (2007). Db_flexrf.py. [Online]. Available: http://gnuradio.org/trac/browser/gnuradio/tags/releases/3.1.3/gr-usrp/src/db_flexrf.py

[17] Analog Devices. PLL Synthesizers/VCOs | RF/IF Components | Analog Devices. [Online]. Available: http://www.analog.com/en/rfif-components/pll-synthesizersvcos/products/index.html

[18] RFIC/Quiet Team, "Technical Specification, Application & Evaluation for the SDR RFIC Version 1.4," Motorola. 2006.

[19] RFIC/Quiet Team, "Technical Specification, Application & Evaluation for the SDR RFIC Version 2.3," Motorola. 2006.

[20] GNU Radio. (2007). Db_base.py. [Online]. Available: http://gnuradio.org/trac/browser/gnuradio/tags/releases/3.1.3/gr-usrp/src/db_base.py

[21] F. Abbas. (2007, Nov.). Simple User Manual for GNU Radio 3.1.1. [Online]. Available: http://www.ece.jhu.edu/~cooper/SWRadio/Simple-Gnuradio-User-Manual-v1.0.pdf

[22] Cypress Semiconductor Corporation. (2002, Jun.). CY7C68013 EZ-USB FX2 USB Microcontroller High-Speed USB Peripheral Controller. [Online]. Available: http://www.keil.com/dd/docs/datashts/cypress/cy7c68xxx_ds.pdf

[23] A. Schooler, Z. Ye, Y. Kim, "SPI Signal Processing Version 1.4," Motorola. 2004.

[24] GNU Radio. (2007). Db_wbx.py. [Online]. Available: http://gnuradio.org/trac/browser/gnuradio/tags/releases/3.1.3/gr-usrp/src/db_wbx.py

[25] GNU Radio. (2007). Db_xcvr2450.py. [Online]. Available: http://gnuradio.org/trac/browser/gnuradio/tags/releases/3.1.3/gr-usrp/src/db_xcvr2450.py

[26] Motorola, "QuIET Analog Bit Descriptions for RFIC4."

[27] B. Ziemer, "RFIC4a Evaluation Board Alignment Procedure Version 1.03," Motorola. 2008.

[28] GNU Radio. (2006). Usrp_fft.py. [Online]. Available: http://gnuradio.org/trac/browser/gnuradio/tags/releases/3.0rc3/gnuradio-examples/python/usrp/usrp_fft.py

[29] GNU Radio. (2006). Usrp_siggen.py. [Online]. Available: http://gnuradio.org/trac/browser/gnuradio/tags/releases/3.0rc3/gnuradio-examples/python/usrp/usrp_siggen.py

[30] AsicAhead, “AA 1001 WiMAX 802.16 wideband RFIC transceiver.”