THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

Department of Electrical Engineering

Implementing a Software-Defined Radio Ground Station for Satellite Communications

Ilya Y. Lipnitskiy
Spring 2010

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Computer Engineering
and Electrical Engineering
with honors in Electrical Engineering

Reviewed and approved* by the following:

Sven G. Bilén
Associate Professor of Electrical Engineering
Thesis Supervisor

John D. Mitchell
Professor of Electrical Engineering
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

In this project a satellite communication system is designed using a software-defined radio. The system consists of the ground station implemented with a Universal Software Radio Peripheral (USRP) and the Si4432 hardware transceiver to be used on the satellite. Using a software-defined radio makes the communication system very flexible. A spectrally efficient digital modulation scheme is selected and a data packet structure is proposed. The system design considers peculiarities of both the USRP and the hardware Si4432 transceiver. Both the USRP and the Si4432 transceiver are then configured according to the proposed design. Test data is collected from the communication program, validated, and analyzed. Finally, possible improvements to the design are discussed and future opportunities explored.

Table of Contents

List of Figures **iv**

List of Tables **v**

Acknowledgments **vi**

Chapter 1
Introduction **1**
1.1 Satellite Communications 1
1.2 Software-Defined Radio 2
1.3 OSIRIS Lite 3
1.3.1 Ground Station 3
1.4 Thesis Overview 3

Chapter 2
Background **4**
2.1 GNU Radio 4
2.2 Universal Software Radio Peripheral 4
2.3 Si4432 Transceiver 5
2.3.1 Si4432 Packet Structure 5
2.4 Ground Station 6
2.5 Gaussian Frequency-Shift Keying 6
2.5.1 GMSK Modulation 7
2.5.2 GMSK Demodulation 8

Chapter 3
USRP Ground Station Design **10**
3.1 Overview 10

3.2 System Design . . . 11
3.3 Test Packet Structure . . . 12
3.4 Si4432 Transceiver Configuration . . . 12
3.5 USRP Configuration . . . 13
3.5.1 Transmit Path . . . 13
3.5.2 Receive Path . . . 14
3.6 Command-line Operation . . . 15

Chapter 4
Testing and Verification **16**
4.1 Sample Data . . . 16
4.2 Performance Results . . . 17

Chapter 5
Conclusions **18**
5.1 Discussion . . . 18
5.2 Future Work . . . 18

Appendix A
Additional Information **20**
A.1 OLite Communication Protocol . . . 20

Appendix B
Source Code **22**
B.1 benchmark_tx.py, USRP Transmitter Front-end . . . 23
B.2 benchmark_rx.py, USRP Receiver Front-end . . . 26
B.3 main.c, LPC2378 Main Program . . . 28
B.4 Si4432.c, Si4432 Interface Program . . . 31
B.5 packet_utils.py, USRP Packet Handling Utilities . . . 36
B.6 pkt.py, USRP Packet Modulating/Demodulating Block . . . 42
B.7 gmsk.py, USRP GMSK Modulator/Demodulator . . . 46

Bibliography **55**

List of Figures

1.1 A typical SDR system 2

2.1 Si4432 Packet Structure [1] 5
2.2 Data modulated with MSK [2] 7
2.3 GMSK VCO Modulator 7
2.4 GMSK I-Q Modulator 8
2.5 GMSK USRP Demodulator 9

3.1 Test Setup 11
3.2 Si4432 Test System Diagram 11
3.3 USRP Test System Diagram 12
3.4 USRP Transmit Path 14
3.5 USRP Receive Path 14

4.1 Si4432 Signal Spectrum Capture 17

List of Tables

3.1 Test Data Packet Structure 12
3.2 Si4432 TX/RX Configuration Registers 13

4.1 Sample packet sent from the Si4432 to the USRP 16
4.2 Sample packet sent from the USRP to the Si4432 17

A.1 OLite Uplink Command Table [3] 21

Acknowledgments

I want to thank my advisor Dr. Sven G. Bilén for providing his guidance and assistance throughout this project. I also want to thank him for introducing me to the field of software-defined radio through a class that I took with him in the Fall of 2009.

I also want to thank my honors adviser, Dr. John D. Mitchell, for inspiring my interest in the field of Electrical Engineering back in 2006 when I took EE 210H with him.

Finally, I want to thank everybody who has made my Penn State experience during these years something I would never forget. I thank my parents for making this education possible and I thank the great faculty members who shared their knowledge with me during classes and laboratories. Finally, I appreciate all the help I have received from my peers and classmates while working together. Without you, my life would not be as bright and colorful as it is.

CHAPTER 1

Introduction

1.1 Satellite Communications

Satellites are remote objects by their nature. For the majority of the tasks they perform, the results need to be sent back to Earth for analysis. If a communication link fails and cannot be reestablished without touching the satellite, the mission effectively fails. A number of things may cause the link to fail. For example, it could be a bug in the firmware running on the satellite or a problem with the communication protocol. Thus, a robust communication system and a robust command and data handling system are crucial components of any satellite mission.

Many factors determine how robust a particular communication system is. At the physical layer, sophisticated modulation schemes and sufficient signal strength ensure that the signal reaches its destination. At higher layers, long packet transmission delays create a need for a good error correction algorithm so that retransmission is not required. In addition to robustness, many systems require secure communication links to ensure data integrity. Establishing such links may require special handshaking protocols as well as strong encryption. It may also be desirable to use multiple frequency ranges for communication for reasons that are related to both robustness and security. Depending on a particular mission, these requirements may change based on the mission goals.

Size and power constraints force satellite communication to be efficient. It may be nice to have the fastest processor on board that takes measurements, encrypts data with the best scheme available and sends a strong low-noise signal back to Earth. However, the development engineer

usually needs to find a compromise to balance power efficiency, cost, and device functionality. For these reasons, the equipment that is put on a satellite is tailored to specific mission requirements with no room for flexibility. The cost of added flexibility on the satellite is high and components are rarely reused (though adapting them for future options may be an option). Currently, most satellites still use hardware radios. Soon it may become practical to use software radios on the satellites. Regardless of their future in space, software radios are being widely adopted for use in many systems here on Earth.

1.2 Software-Defined Radio

A typical radio system consists of a number of signal processing components. Those include, but are not limited to, a relatively narrow-band antenna, Low-Noise Amplifier (LNA), mixer for a given relatively small frequency range, filters, and specific (de)modulators. Many of the component parameters are fixed and designed to work only for a specific tasks. In fact, we are surrounded with many such radios in our everyday life: car radios, GPS navigators, cellphones, walkie-talkies, television. All of these devices use Electromagnetic Waves to communicate, yet the hardware from one device is mostly incompatible with another. Sometimes we can notice the similarities, for example I remember being able to listen to some radio stations on a TV and vice versa. A GSM phone placed next to a speaker tends to produce strange sounds due to radio interference. This happens because all radios share the same medium to transmit signals, and sometimes signals overlap. Nevertheless, up until recently, it was nearly impossible to build a universal radio to combine all functionality in a single device.

Recent developments in high-speed Analog-to-Digital Converter (ADC) and fast parallelizeable Field-Programmable Gate Array (FPGA) architectures have made software radios possible. The fundamental characteristic of software radio is that software defines the transmitted waveforms, and software demodulates the received waveforms [4]. Ideally, an antenna would be connected directly to an ADC, which would digitize the signal and pass it to the processsor. In current reality this is not feasible. However, by doing minimal processing at the input it is possible to leave most of the signal processing up to software. Figure 1 depicts a sample SDR system. Software radios were first mentioned by J. Mitola in 1993 [5]. A few years later he published another work listing examples of areas where SDR could be used and proposed an architecture for those radios [6].

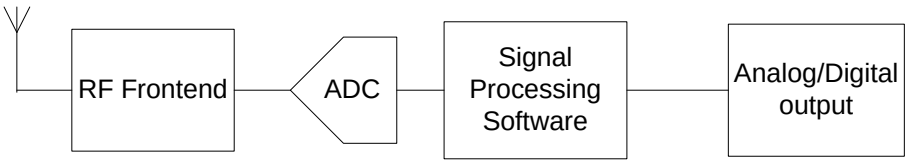

Figure 1.1. A typical SDR system

1.3 OSIRIS Lite

OSIRIS Lite (OLite) is a project, being developed by the Student Space Project Laboratory (SSPL) at The Pennsylvania State University. The project involves launching a high-altitude balloon with the prototype of the satellite to simulate the actual launch. A variety of sensors will be integrated with the system. The system will be controlled using a Field Programmable Gate Array (FPGA). All sensor data will be transmitted to the ground station on Earth using the Si4432 transceiver. The transceiver will utilize defined radio bands, packet structure, and transport protocols.

1.3.1 Ground Station

The ground station is a vital part of the OSIRIS project that communicates with the satellite from the ground.

In Section 1.1 current challenges with satellite communications were introduced. Whereas software radio may be impractical for the satellite system, it makes a lot of sense to implement a software-defined ground station. First, an SDR-based ground station can serve as a hub for multiple satellites using different communication protocols and frequencies. Second, the ground station could process incoming traffic in a dynamic way. The ground station could parse the traffic according to signal frequency. For example, it could put error messages from different satellites into different files. Should the protocol ever change, the only changes would need to be made in software. Finally, should the satellite finish its mission, the general purpose ground station would not be thrown away. Instead, it could be reused for future projects.

1.4 Thesis Overview

The current ground station design uses a conventional hardware transceiver and a microcontroller. This thesis proposes an SDR ground station implementation using the Universal Software Radio Peripheral (USRP). The USRP presents a flexible solution that is adaptable to a variety of different communication specifications. Required functionality is implemented by modifying the Digital Signal Processing (DSP) modules of the GNU Radio platform. Chapter 2 provides the reader with the background information on the GNU Radio platform, the USRP, the Si4432 Transceiver, and the GMSK Modulation. Chapter 3 describes the actual implementation of the communication system. Chapter 4 focuses on system performance and testing. In Chapter 5 the results are discussed and future improvements are suggested. Excerpts of the relevant source code used in this project are presented in Appendix-B.

CHAPTER 2

Background

2.1 GNU Radio

GNU Radio is a free software development toolkit that provides the signal processing runtime and processing blocks to implement software radios using readily-available, low-cost external RF hardware and commodity processors [7].

Performance critical signal processing modules are implemented in C++. Non-performance critical modules and User-Interface (UI) features are developed in Python. The Simplified Wrapper and Interface Generator (SWIG) is used to connect the C++ modules with Python. The modules are connected in a flow graph using Python. Once the program starts, the flow graph is enabled. During program execution, the flow graph can be altered dynamically. Version 3.2.2 of the GNU Radio software is used in this project.

2.2 Universal Software Radio Peripheral

The Universal Software Radio Peripheral (USRP) is a universal radio device created by Matt Ettus to be used primarily with the GNU Radio project. The USRP consists of a small motherboard containing up to four 12-bit 64 Msample/sec ADCs, four 14-bit, 128 Msample/sec DACs, a million gate FPGA, and a programmable USB 2.0 controller. Each fully populated USRP motherboard supports four daughterboards: two for receive and two for transmit [4]. The daughterboards are made for specific frequency ranges. The two daughterboards relevant for this project are the

RFX400 and the RFX900 transceiver boards. The RFX400 operates in the 400–500 MHz range, and the RFX900 operates in the 800–1000 MHz range.

2.3 Si4432 Transceiver

The Silicon Labs Si4432 transceiver was chosen for the hardware implementation of radio communication in the OLite project. This is the transceiver that will be used to communicate data from the satellite. The Si4432 was chosen for a number of reasons. First of all, it has a low power consumption of 18.5 mA during receive, 30 mA at +13 dBm transmit, and 85 mA at +20 dBm transmit. Low power consumption is crucial for a satellite mission. Secondly, the Si4432 supports the frequency range of 240–930 MHz, and our required bands of 450 MHz and 915 MHz fall within that range. The transceiver has a built-in packet handler that allows for efficient packet formation on-chip, reducing the transceiver communication overhead with its controller during TX packet formation. The Si4432 also features high RX sensitivity of −121 dBm, allowing for easier packet reception [1].

2.3.1 Si4432 Packet Structure

In order to communicate digital information over any network the data must be serialized and put in a packet of predefined structure. Once the packet is sent over a network, the receiver must not only recover the digital data (refer to Section 2.5), but also determine where the packet starts and ends. Additionally, often we want to verify packet correctness and/or provide error correction. The general packet structure for the Si4432 packet handler is given in Figure 2.1.

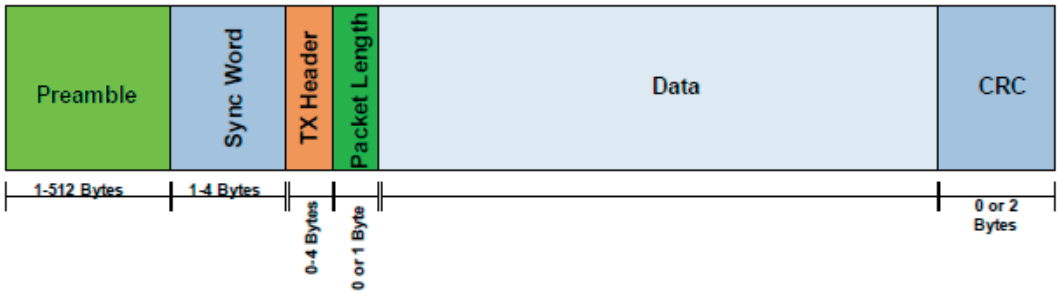

Figure 2.1. Si4432 Packet Structure [1]

The preamble and synchronization word are used to determine where the packet starts, with the preamble defined for Si4432 to be a series of alternating ones and zeroes, starting with a zero [1]. Sync word is a 4-byte sequence defined to be ACDDA4E2 in hexadecimal.

2.4 Ground Station

Initially, the ground station was intended to be built using Si4432 transceivers. However, this work proposes using a USRP to implement the ground station. There are some trade-offs in this decision. Using a pair of Si4432 transceivers leads to greater compatibility, due to the same hardware and firmware being used on both sides. Moreover, being a commercial product the Si4432 features a detailed datasheet and many application notes with detailed descriptions of the TX and RX side. Silicon Labs also provides the developer with an easy-to-use Excel spreadsheet to calculate TX and RX register values based on the specified radio parameters.

The USRP, on the other hand, offers the developer more flexibility than a hardware radio. The added flexibility adds a lot of complexity because every possible detail of the implementation has to be worked out to match the performance of a hardware radio. The GNU Radio project does a good job of lowering the complexity barrier by providing pre-made modules. However, these modules usually need to be tuned substantially to get reasonable performace. Moreover, due to the open-source nature of the GNU Radio project, not much support is provided. There is a mailing list and a wiki and a handful of documents describing some modules, but if one wants to use and understand GNU Radio hours must be spent looking at the actual C++ and Python source code.

The timeframe for this project along with the aforementioned reasons required the focus of this work to shift slightly. Instead of building a complete USRP ground station system, the core functionality was implemented as a proof of concept.

2.5 Gaussian Frequency-Shift Keying

Gaussian Frequency-Shift Keying (GFSK) Modulation was chosen for several reasons. First of all, it is used in many widespread applications such as the GSM network and Bluetooth, which guarantees that devices will keep supporting GFSK in the near future. The Si4432 transceiver has GFSK support and is the manufacturer recommended modulation scheme. There are a number of reasons why GFSK is a popular modulation scheme. First of all, it is a frequency keying scheme, so it uses a frequency modulated signal to encode data. Frequency modulation (FM) schemes do not depend on amplitude variation, thereby eliminating a major source of noise. Frequency Shift Keying encodes a digital signal into a continuous waveform by representing each bit as a half-sinusoid. A Gaussian smoothing filter is applied before mixing the signal in order to reduce out-of-band spectrum. The reason we get out-of band spectrum is because of step transitions from one waveform to the other, resulting in frequency spikes. Gaussian filtering mitigates that without decreasing signal quality.

A particular case of GFSK is Gaussian Minimum-Shift Keying (GMSK). GMSK is characterized by *continuous* phase transitions from one symbol to the next. This is accomplished by setting the frequency shift at exactly half the bitrate of data coming in. As a result, symbol transitions always occur at 0 amplitude. Another way to define MSK is to say that it is FSK with a

modulation index (frequency deviation / bitrate) of 0.5. Figure 2.2 illustrates this principle.

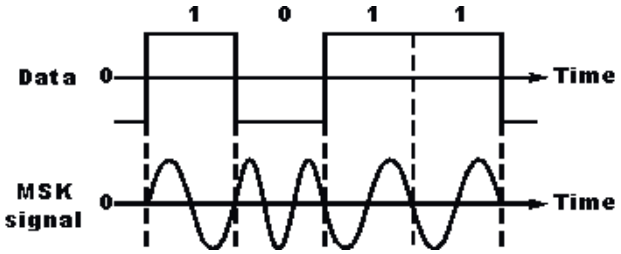

Figure 2.2. Data modulated with MSK [2]

One trade-off involved with choosing the modulation index is spectrum usage vs. inter-symbol interference (ISI). For a single-channel system, a wider spectrum can be used to minimize ISI. For multiple channel or multiple user systems, however, it may be reasonable to have a slightly higher bit error rate (BER) in return for greater spectral efficiency. Si4432 transceiver allows the developer to select the modulation index they prefer to use. The GNU Radio demodulator, however, is tailored more to receiving GMSK. Given the advantages of GMSK, it was decided to configure Si4432 to transmit using GMSK.

2.5.1 GMSK Modulation

There are two main ways to modulate a signal with GMSK. One is to use a voltage controlled FM-modulator that modulates the signal at one-half its bitrate and mixes the signal with a sinusoid of needed frequency. The diagram for this device is given in Figure 2.3.

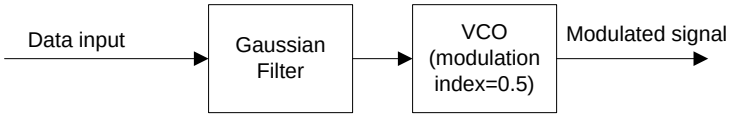

Figure 2.3. GMSK VCO Modulator

The output signal is ready for transmission, but should be passed through a Power Amplifier (PA) before transmitting. This is a simple model that can be implemented using a conventional Voltage-Controlled Oscillator (VCO). The problem with it, however, is that, due to temperature or other ambient conditions, the oscillator's parameters begin to vary, changing the modulation index, which for GMSK must be kept at exactly 0.5. A solution to that is to use a different approach to modulation.

The second approach uses an I-Q modulator to modulate a signal at exactly half its bitrate frequency. The signal is split into an *In-Phase* signal and a *Quadrature* signal, which is the same signal with phase shifted by 90°. The I and Q signals can be thought of as encoding even and odd data, respectively. Thus, it is implicit that the signal will be modulated at one-half its bitrate. The diagram for an I-Q GMSK modulator is provided in Figure 2.4.

Again, the output signal needs to be amplified to be transmitted a reasonable distance. These diagrams provide only a simplified view of modulator blocks. In practice, the Si4432 transceiver

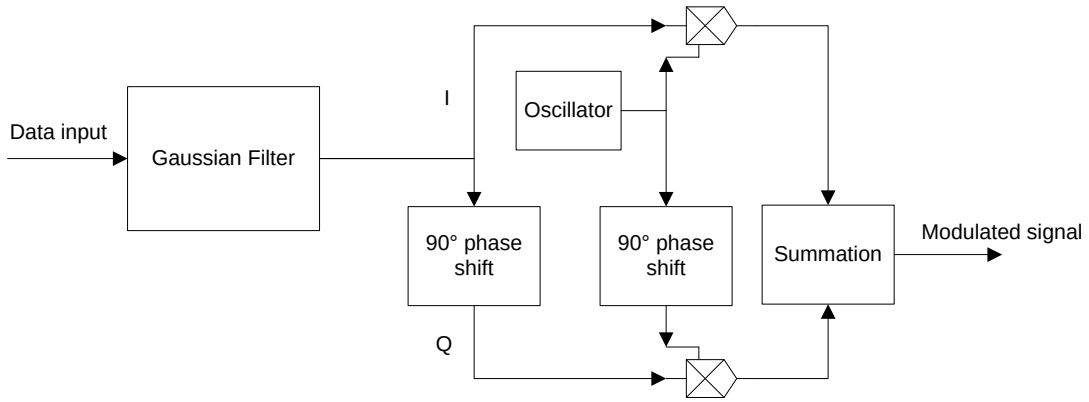

Figure 2.4. GMSK I-Q Modulator

uses a combination of a voltage-controlled oscillator and a $\Delta\Sigma$ Fractional-N Phase-Locked Loop (PLL) synthesizer to generate the signal for the modulator and demodulator. See [8] for details.

2.5.2 GMSK Demodulation

The exact specifics of the Si4432 transceiver demodulation were not published by Silicon Labs. They do however, provide the developer with an Excel spreadsheet to calculate appropriate register values based on the carrier frequency, frequency deviation, and data rate. Therefore we will focus on the GNU Radio implementation of the demodulator.

The demodulator blocks are laid out in *pkt.py* and *gmsk.py* GNU Radio blocks. The demodulator takes in a baseband modulated signal and turns it into a digital data packet. The quadrature FM demodulator block takes a complex baseband signal and converts it to a stream of floating point numbers. The quadrature (I-Q) demodulator structure is similar to the reverse of Figure 2.4. The stream of floats is then passed to the clock recovery block that uses an optimized Mueller & Müller [9] algorithm proposed by Danesfahani *et al.* in [10]. One of the parameters that influences clock recovery is μ. The default value for μ is 0.5. However, on the mailing list, the developers suggest using μ =0.001 [11]. Once the clock has been recovered we know the time locations of symbols in the signal. The symbols (represented as float numbers) are then passed into a binary slicer that assigns negative values a value of 0 and positive values a value of 1. Bytes of data from the slicer contain valid information only in the least significant bit (LSB). This data goes into the correlator that matches the access code and synchronization word and passes the data into a framer sink. The sink then packs the valid packets into the message queue and passes control to higher level routines.

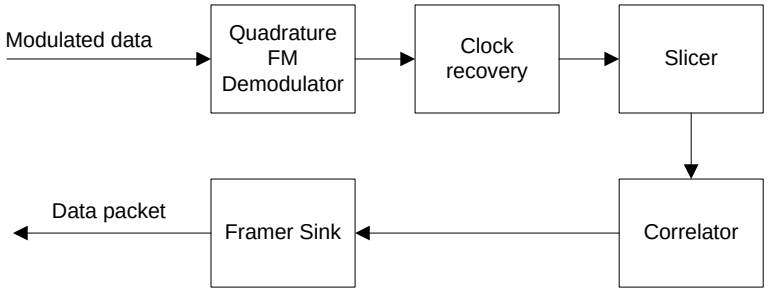

Figure 2.5. GMSK USRP Demodulator

CHAPTER 3

USRP Ground Station Design

3.1 Overview

As was mentioned in Section 2.1, a GNU Radio system is represented with a flow graph. The graph contains blocks that are initialized using *gr.hier_block2.__init__* Python function. Blocks already implemented in C++ are initialized using their respective initialization functions in the GNU Radio architecture. A block is something that has an input and an output of a defined format. For example, the general GMSK demodulator block takes in a complex signal of type *gr.sizeof_gr_complex* and outputs a character string of type *gr.sizeof_char*. The blocks are connected using the *connect()* function. An obvious requirement for connecting two blocks together is that the output of the first block must be of the same type as the input of the second block. A *connect()* statement must contain at least two blocks.

In addition to functional blocks, Python commands are used to interact with the caller of the program. Various information, such as program status, received data, and debug output may be output to *stdout* or to a file sink. For this design, the received packet information and payload are displayed on the calling terminal. The actual photo of the test setup is displayed in Figure 3.1. In the figure, the USRP is a black box on the left, and the Si4432 transceiver is the board on the right.

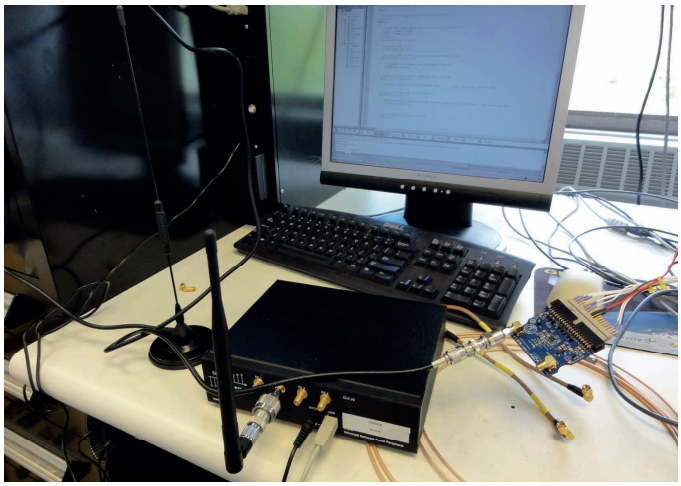

Figure 3.1. Test Setup

3.2 System Design

Our test system consists of a Si4432 transceiver, two antennas, and a USRP. The Si4432 is controlled by an NXP LPC2378 microcontroller on a MCB2300 Evaluation Board provided by Keil. The microcontroller communicates with the Si4432 using an SPI interface. The program running on the microcontroller initializes the Si4432 transceiver by setting up registers. It monitors interrupts set by the Si4432 in case of an event. An event can be triggered by reception of a packet, transmission of a packet, error, or other information. The microcontroller also communicates with the PC via RS232 UART interface. Data received from the UART is put in a packet and transmitted by the Si4432. Data received from the Si4432 is sent over to the UART and displayed to the operator.

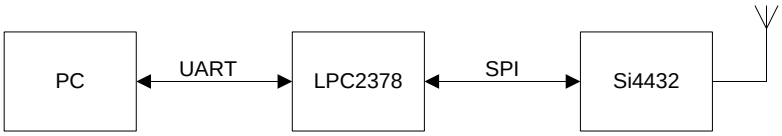

Figure 3.2. Si4432 Test System Diagram

The USRP side of the test system consists of a USRP v1 with RFX400 and RFX900 daughterboards connected to a PC. Initial tuning and signal processing happens on the USRP daughterboards. The received signal is then passed to the 64-MSample/sec ADC and sampled. The resulting baseband signal is passed via Universal System Bus (USB) to the computer running Linux and GNU Radio software. Upon recognizing an incoming packet, the software then displays its contents to the PC user.

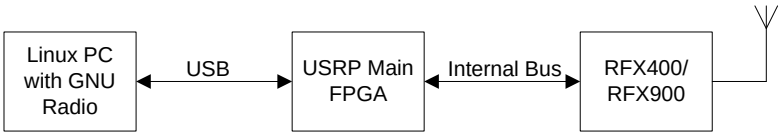

Figure 3.3. USRP Test System Diagram

3.3 Test Packet Structure

In order to transmit and receive packets a matching packet structure must be selected. The sent packet needs to be recognized properly by the receiving side. Specifically, the GNU Radio packet receiving blocks expect a packet structure composed of a preamble and a header that, among other information, includes packet length. While the preamble length is arbitrary, the *framer_sink_1* block expects a 32-bit header composed of two replicated 16-bit headers. If the packet header does not match that format, then the GNU Radio software will discard the whole packet. The Si4432, however, only allows for up to 1 byte of packet length in the explicit length field. Therefore, the packet length needs to be transmitted in the header field of the Si4432 packet structure (depicted in Figure 2.1). Another option would be to write a new GNU Radio block with a different header checking algorithm, but that would take a lot of time. The current solution seems more than reasonable unless header length becomes a concern. It also should be noted that the USRP makes no distinction between Preamble and Synchronization Word, but this does not affect compatibility. Payload CRC field is optional and has been omitted for test purposes, although error checking and correction would need to be implemented for the actual mission. The proposed compatible packet structure is shown below.

Table 3.1. Test Data Packet Structure

Preamble (4 bytes)	Header bits 31–28	bits 27–16	bits 15–12	bits 12–0	Payload
ACDDA4E2	unused	Length (hex)	unused	Length (hex)	Payload

3.4 Si4432 Transceiver Configuration

The Si4432 transceiver configuration is accomplished by setting configuration registers. Detailed register descriptions are available from Silicon Labs in their application note AN440 [12]. All register addresses are given in hexadecimal notation. Most values can be calculated using the formulas provided in AN440. Alternatively, an Excel spreadsheet is provided by Silicon Labs for quicker register value calculation. Leading bits in register addresses that are not listed are assumed to be zero. Both TX and RX carrier frequencies were set to 915 MHz.

The relevant settings for the transmitter include modulation, data source, frequency deviation, and transmit data rate. GFSK modulation with a FIFO (First-In First-Out) modulation source are used to allow for efficient packet handling and retransmission. Transmit data rate and frequency deviation determine the modulation index. Since the transmit data rate was chosen

to be 64 kbps by design, we need a frequency deviation of 32 kHz to achieve an index of 0.5 (GMSK).

Receiver settings were determined using the provided spreadsheet. Relevant register values for the Si4432 transceiver are listed in the table below.

Table 3.2. Si4432 TX/RX Configuration Registers

Register address	Value
1Ch	0x07
1Dh	0x40
1Fh	0x03
20h	0x3F
21h	0x02
22h	0x0C
23h	0x4A
24h	0x04
25h	0x12
6Eh	0x10
6Fh	0x62
70h	0x0C
71h	0x23
72h	0x33
75h	0x75
76h	0xBB
77h	0x80

3.5 USRP Configuration

3.5.1 Transmit Path

For the test transmit program a sample packet was created and hard-coded into the source code. A future modification would allow for custom packet sending or sending from a file. The transmit path was mostly based upon GNU Radio sample code with some modifications and additions to match the specifications of this project.

The following Python script files were used to set up transmit path: *benchmark_tx.py*, *usrp_transmit_path.py*, and *transmit_path.py*. Also, the scripts in *pkt.py*, *packet_utils.py*, and *gmsk.py* were used for both transmit and receive. *benchmark_tx.py* represents the front-end of the program and this is where the packet is defined and taken from user input. Another file affecting packet structure that also required modification is *packet_utils.py*. It contains various packet utilities that assemble and disassemble the packet. Among other things, this script adds/removes the preamble to the packet and appends the header and CRC. The code was modified to reflect the packet structure depicted in Table 3.1.

In addition to Python scripts, a number of C++ blocks were used to construct the transmit path. The diagram of the blocks is provided in Figure 3.4. The transmit path works as follows.

A constructed packet is placed in the packet queue as a byte array. The byte array is converted to float symbol array by the Non-Return to Zero (NRZ) converter block. The symbols are then passed to the Gaussian filter to introduce Gaussian noise for GMSK transmission. The resulting symbols are then FM-Modulated with a phase change of $\frac{\pi}{2}$ (a property of MSK, refer to Section 2.5). Output of the modulator is then connected to a digital-to-analog converter (DAC) that converts digital values into a waveform. The signal is then interpolated by the RFX900 board up to the carrier frequency, which is 915 MHz in this case.

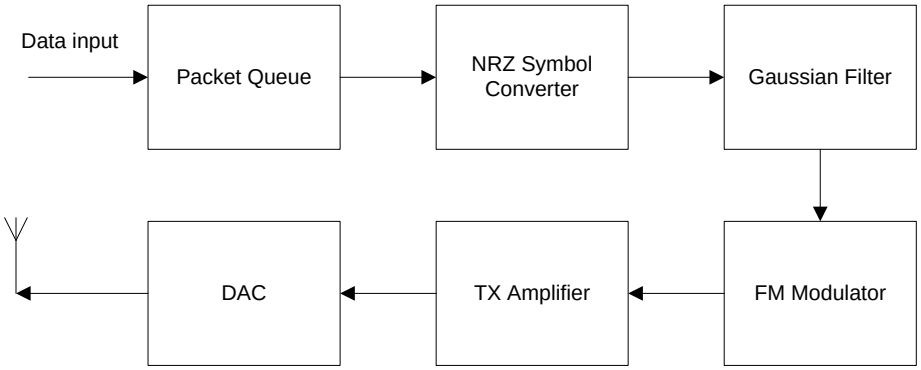

Figure 3.4. USRP Transmit Path

3.5.2 Receive Path

The receiver was trickier to set up because some details of the modulation process are not listed in the Si4432 data sheet. As with the transmitter described in the previous section, the receiver was built using modified sample GNU Radio code. The same packet handling and modulation scripts were used. They include *pkt.py*, *packet_utils.py*, and *gmsk.py*. In addition to that, the receive path was constructed using scripts *benchmark_rx.py*, *usrp_receive_path.py*, and *receive_path.py*. The diagram of the signal processing blocks used is provided in Figure 3.5.

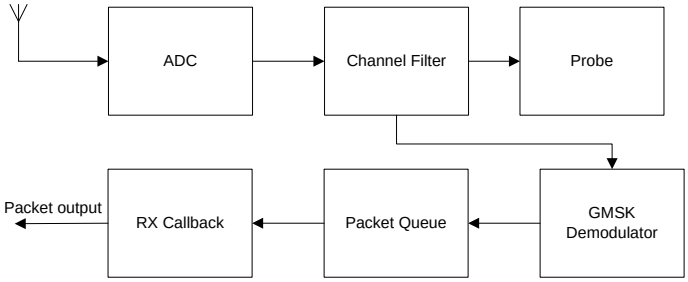

Figure 3.5. USRP Receive Path

The signal is first received by the RFX900 board that sets the center frequency to 915 MHz. The signal at baseband is then sampled by the USRP's analog-to-digital Converter (ADC) using a default decimation factor of 64. Since the ADC samples at 64 MSamples/sec, the digital signal

comes out of the ADC at a rate of 1 Mbit/s. The channel filter block is a low pass filter with a Hanning window. The filter parameters were selected to match the frequency deviation of the Si4432 transmitter. They were then adjusted empirically to get the best performance. The sampling rate was determined to be 280 kHz, the midpoint of the transition band was determined to be 130 kHz and the width of the transition band was determined to be 30 kHz. The probe block is a simple measurement block that communicates with Python scripts to provide carrier sensing indication capability. The output of the channel filter then goes into the GMSK Demodulator block. USRP Demodulation uses M&M clock recovery to recover symbols and was described in Section 2.5.2. Once the packet is in the queue the callback function defined in *benchmark_rx.py* is called and the packet contents are displayed on the terminal.

3.6 Command-line Operation

For the Si4432 transceiver, the code loops indefinitely on the LPC2378 microcontroller once uploaded, and all user interaction happens over the serial port link. Any program supporting serial port communication would work, but a LabVIEW component was also built to customize the terminal window. The transceiver control is explained in more detail in Section 3.2.

For the USRP, the two files initializing the radio are *benchmark_rx.py* and *benchmark_tx.py*.

- Receiver Usage: `./benchmark_rx.py -f freq -S 4 --mu=0.001 -r 64k -m gmsk`
- Transmitter Usage: `./benchmark_tx.py -f freq --size 32 -r 64k -m gmsk`

where `freq` denotes carrier frequency in hertz. For the data transmission it is `915M`, for the beacon it is `450M`. `M` stands for megahertz.

CHAPTER 4

Testing and Verification

4.1 Sample Data

For testing the USRP receiver, sample packets containing a sequence number and payload were continuously sent from the Si4432 transceiver. The TX Power of the Si4432 transmitter was set to −8 dBm. A screen capture from a spectrum analyzer depicts the bandwidth and signal strength generated by the Si4432 during packet send (see Figure 4.1). For the measurement, the transceiver was hooked up directly to the signal analyzer.

Table 4.1. Sample packet sent from the Si4432 to the USRP

Preamble	Header bits 31–28	bits 27–16	bits 15–12	bits 12–0	Payload
ACDDA4E2	unused	0x8	unused	0x8	Seqnum+"123456"

The measurement was made by comparing the sequence number of the received packet with the total number of packets received. Since there was no payload error checking, some packets were also subject to CRC error, but from looking at the received packets the number of such packets was relatively small.

After testing the USRP receiver, the transmitter was also tested by forming sample packets and sending them to the Si4432. The error rate was hard to determine because it is likely that either the Si4432 FIFO or the LPC2378 UART overflowed with packets. One solution to that problem would be to reduce the rate of sending data or to turn off packet data display on the Si4432 terminal. The microcontroller could calculate the percent error and output only that to

the terminal.

Table 4.2. Sample packet sent from the USRP to the Si4432

Preamble	Header bits 31–28	bits 27–16	bits 15–12	bits 12–0	Payload
ACDDA4E2	unused	0x32	unused	0x32	32-byte string

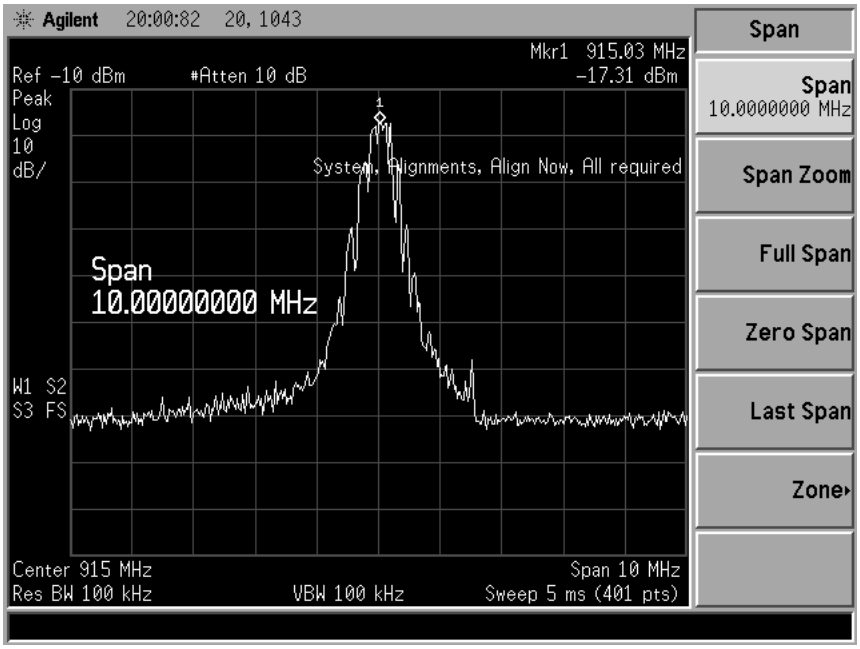

Figure 4.1. Si4432 Signal Spectrum Capture

4.2 Performance Results

Unfortunately, a perfect result was not achieved. The best attainable design for the USRP receiver was able to receive ≈80% of packets. There are a number of sources of error to be considered. First of all, the demodulator is controlled by a set of variables. The values that were used for those variables were either the default or ones suggested on the GNU Radio developer mailing list [11]. Careful analysis and simulations could yield a set of values that produce a better result. However, it would be hard to simulate the communication between the Si4432 and the USRP because Silicon Labs has not releazed the modulator specification for the Si4432 and some of the modulation parameters are unknown. If those specifications are released at some point, then GNU Radio demodulation blocks could be modified to fit the Si4432 modulation specification. Based on the considerations listed above, this result may be considered acceptable for the current proof-of-concept system. However, improving the system's performance should still be one of the top-priority tasks.

CHAPTER 5

Conclusions

5.1 Discussion

This work has shown that a Software-Defined Radio system can be customized to interface with a hardware transceiver. The flexibility of the USRP as well as GNU Radio software allows matching radios that have varying signal characteristics and properties. Existing signal processing blocks can be changed to fit the system design as needed. In this work, some of the changes included creating a new data packet structure as well as modifying the channel filter at the receiver side.

While the project was focused on building the USRP receiver and transmitter, a lot of time was also spent studying Si4432 control registers and the Si4432 transceiver architecture. That was needed because there are many steps in digital packet formation and transmission over the radio. There are also many variables within those steps. Each detail needed to be worked out in order to get a transmission going.

5.2 Future Work

One should keep in mind that this project is more of a proof-of-concept than a complete functioning system. The initial goal was to build a complete system. However, the amount of material needed to be understood combined with the timeframe of this project required the focus to shift to building a rather basic system. Nevertheless, I believe that these basic concepts can be extended and used to build a complete functioning ground station.

A thorough understanding of GNU Radio code was achieved while modifying the system. This knowledge can be used in the future to extend the functionality of the GNU Radio software. In addition to the rich codebase provided by the GNU Radio Project, new blocks can be built and integrated into the GNU Radio Project. One hurdle that was faced is that the Si4432 transceiver only supports CRC-16 for packet error checking, whereas the GNU Radio implements CRC-32. A block implementing CRC-16 CRC would help use the functionality of Si4432 transceiver without delegating CRC calculation to the microcontroller. Another algorithm that can be improved is the header checking algorithm. Currently, the framer sink only checks for two exact copies of a header to accept the packet length. A more sophisticated header error checking algorithm would make the system more robust. Other parameters such as clock recovery variables and filter parameters would benefit from more documentation. Currently, many of the GNU Radio features are not documented and countless hours must be spent looking at the source code to understand what is going on.

Using a radio to transmit digital data also adds a layer of complexity because a single bit error can render the whole packet useless. Long propagation delays associated with satellite communications make it expensive to retransmit an entire packet because of a small error. Adequate error correction mechanisms need to be employed to recover packets from small errors. Higher layer algorithms, such as congestion control and retransmission, may also become necessary as system complexity increases and multiple users start using the system.

APPENDIX A

Additional Information

A.1 OLite Communication Protocol

Table A.1. OLite Uplink Command Table [3]

ID:	Size (bits):	Description:	Type:
0x00	8	GNC Experiment On	General
0x01	8	GNC Experiment Off	
0x02	8	Power Experiment On	
0x03	8	Power Experiment Off	
0x04	8	Comm RSSI Experiment On	
0x05	8	Comm RSSI Experiment Off	
0x06	8	Comm BER Experiment On	
0x07	8	Comm BER Experiment Off	
0x08	8	Patch Heater On	
0x09	8	Patch Heater Off	
0x0A	8	Reset Magnetometer	GNC
0x0B	8	Sun sensor 1 On	
0x0C	8	Sun sensor 1 On	
0x0D	8	Sun sensor 1 On	
0x0E	8	Sun sensor 1 On	
0x0F	8	Sun sensor 1 On	
0x10	8	Sun sensor 1 Off	
0x11	8	Sun sensor 1 Off	
0x12	8	Sun sensor 1 Off	
0x13	8	Sun sensor 1 Off	
0x14	8	Sun sensor 1 Off	
0x15	8	Reset 3.3-V Power Line	Comm
0x16	8	Reset 5V Power Line	
0x17	8	Transceiver – RSSI Mode	
0x18	8	Transceiver – Transmit Mode	
0x19	8	Transceiver – Receive Mode	
0x1A	8	Power Cycle C&DH	Power
0x1B	8	Power Cycle GNC	
0x1C	8	Power Cycle Comm	
0x1D	8	Ascent Mode	CDH
0x1E	8	Float Mode	
		Change Config Files – Last 4 bite: data (0 – 15)	CDH Config Files
0x2	8	Change HK Interval (seconds)	
0x3	8	Change Camera Ascent Interval (Minutes)	
0x4	8	Change Camera Float Interval (Minutes)	

APPENDIX B

Source Code

All Python files were modified and distributed in accordance with GNU Public License. The notice is posted below:

```
# Copyright 2005,2006,2007,2009 Free Software Foundation, Inc.
#
# This file is part of GNU Radio
#
# GNU Radio is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 3, or (at your option)
# any later version.
#
# GNU Radio is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with GNU Radio; see the file COPYING.  If not, write to
# the Free Software Foundation, Inc., 51 Franklin Street,
# Boston, MA 02110-1301, USA.
```

B.1 benchmark_tx.py, USRP Transmitter Front-end

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


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

import random, time, struct, sys

# from current dir
import usrp_transmit_path

#import os
#print os.getpid()
#raw_input('Attach and press enter')

class my_top_block(gr.top_block):
    def __init__(self, modulator, options):
        gr.top_block.__init__(self)

        self.txpath = usrp_transmit_path.usrp_transmit_path(modulator
            , options)

        self.connect(self.txpath)

# /////////////////////////////////////////
#                                   main
# /////////////////////////////////////////

def main():

    def send_pkt(payload='', eof=False):
        return tb.txpath.send_pkt(payload, eof)

    def rx_callback(ok, payload):
        print "ok = %r, payload = '%s'" % (ok, payload)
```

```
mods = modulation_utils.type_1_mods()

parser = OptionParser(option_class=eng_option, conflict_handler="
    resolve")
expert_grp = parser.add_option_group("Expert")

parser.add_option("-m", "--modulation", type="choice", choices=
    mods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%
                          default]"
                            % (', '.join(mods.keys()),))

parser.add_option("-s", "--size", type="eng_float", default=1500,
                      help="set packet size [default=%default]")
parser.add_option("-b", "--bytes", type="eng_float", default=1.0,
                      help="set bytes to transmit [default=%default]"
                          )
parser.add_option("","--discontinuous", action="store_true",
    default=False,
                      help="enable discontinous transmission (bursts
                          of 5 packets)")
parser.add_option("","--from-file", default=None,
                      help="use file for packet contents")

usrp_transmit_path.add_options(parser, expert_grp)

for mod in mods.values():
    mod.add_options(expert_grp)

(options, args) = parser.parse_args ()

if len(args) != 0:
    parser.print_help()
    sys.exit(1)

if options.tx_freq is None:
    sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
    parser.print_help(sys.stderr)
```

```
    sys.exit(1)

if options.from_file is not None:
    source_file = open(options.from_file, 'r')

# build the graph
tb = my_top_block(mods[options.modulation], options)

r = gr.enable_realtime_scheduling()
if r != gr.RT_OK:
    print "Warning: failed to enable realtime scheduling"

tb.start()                              # start flow graph

# generate and send packets
nbytes = int(options.bytes)
n = 0
pktno = 0
pkt_size = int(options.size)

while n < nbytes:
    if options.from_file is None:
                data = "Hello world!!! Hello world!!!!!"  #Test
                    payload, length=30
    else:
        data = source_file.read(pkt_size - 2)
        if data == '':
            break;

    payload = struct.pack('!H', pktno & 0xffff) + data #Now
        payload length = 32 bytes
    send_pkt(payload)
    n += len(payload)
    sys.stderr.write('.')
    if options.discontinuous and pktno % 5 == 4:
        time.sleep(1)
    pktno += 1

send_pkt(eof=True)
```

```
    tb.wait()                       # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

B.2 benchmark_rx.py, USRP Receiver Front-end

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


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

import random
import struct
import sys

# from current dir
import usrp_receive_path

#import os
#print os.getpid()
#raw_input('Attach and press enter: ')

class my_top_block(gr.top_block):
    def __init__(self, demodulator, rx_callback, options):
        gr.top_block.__init__(self)

        # Set up receive path
        self.rxpath = usrp_receive_path.usrp_receive_path(demodulator
            , rx_callback, options)

        self.connect(self.rxpath)
```

```
# ////////////////////////////////////////
#                                    main
# ////////////////////////////////////////

global n_rcvd, n_right

def main():
    global n_rcvd, n_right

    n_rcvd = 0
    n_right = 0

    def rx_callback(ok, payload):
        global n_rcvd, n_right
        (pktno,) = struct.unpack('!H', payload[0:2])
        n_rcvd += 1

        print "Sequence number = %4d  Number Received = %4d" % (
            pktno, n_rcvd, n_right)
                print "Payload = ", string_to_hex_list(payload)


    demods = modulation_utils.type_1_demods()

    # Create Options Parser:
    parser = OptionParser (option_class=eng_option, conflict_handler=
        "resolve")
    expert_grp = parser.add_option_group("Expert")

    parser.add_option("-m", "--modulation", type="choice", choices=
        demods.keys(),
                      default='gmsk',
                      help="Select modulation from: %s [default=%%
                          default]"
                            % (', '.join(demods.keys()),))

    usrp_receive_path.add_options(parser, expert_grp)

    for mod in demods.values():
        mod.add_options(expert_grp)
```

```
    (options, args) = parser.parse_args ()

    if len(args) != 0:
        parser.print_help(sys.stderr)
        sys.exit(1)

    if options.rx_freq is None:
        sys.stderr.write("You must specify -f FREQ or --freq FREQ\n")
        parser.print_help(sys.stderr)
        sys.exit(1)


    # build the graph
    tb = my_top_block(demods[options.modulation], rx_callback,
        options)

    r = gr.enable_realtime_scheduling()
    if r != gr.RT_OK:
        print "Warning: Failed to enable realtime scheduling."

    tb.start()        # start flow graph
    tb.wait()         # wait for it to finish

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
```

B.3 main.c, LPC2378 Main Program

```
//Transceiver interface converter for LPC2378
//Translates UART to SPI, and vice versa

//Written by: David Zhang
//Modified by: Ilya Lipnitskiy

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include "OtherFunctions.h"
#include "UART.h"
#include "IRQ.h"
#include "SPI.h"
#include "Constants.h"

#include "Si4432.h"              //transceiver header file

/************ Interface input / output buffers ******************/
//struct to store the global input buffers, includes the data and a
    counter
volatile struct Buffer UARTBuffer;

int main(void)
{
        /* VARIABLES */
        struct Buffer SIBuffer;          //stores the data from
            transceiver
        char buf[10];
        int count;

        /************ Initialize ***************/
        VIC_init();                                      //install
            interrupt handler
        serial_init(0, 57600);           //initialize UART port 0 for
            input
        LED_init();                                      //initialize
            test LEDs
        SPI_init();                                      //initialize
            SPI
        SI_init();                                       //init
            transceiver

        /************ init *************/
        UARTBuffer.Counter = 0;
                    //reset input buffer counter
        SIBuffer.Counter = 0;

        #if DEBUG
```

```
#define STARTMSG "Board initialization complete"
serial_transmit(ERRPORT, STARTMSG, sizeof(STARTMSG));
#endif

/***************** Main loop ********************/
count = 0;
while(1)
{
        /* USRP TEST CODE, append packet # in hex at
            beginning */
        buf[0] = (char) (count >> 8);
        buf[1] = (char)(count);
        count++;
        strcpy(&buf[2], "123456");

        //SI_transmit(buf, 8);
        /* USRP TEST CODE END */

        //transmits UART buffer to transceiver when there is
            data
        if(UARTBuffer.Counter > 0)
        {
                SI_transmit(UARTBuffer.Data, UARTBuffer.
                    Counter);
                UARTBuffer.Counter = 0;
        }

        //transmits received data to UART when data received
        if(SI_interruptReceived())
        {
                SIBuffer.Counter += SI_read(SIBuffer.Data);
                                //read transceiver FIFO for
                    data
        }

        //transmit received data to host computer
        if(SIBuffer.Counter > 0)
        {
                serial_transmit(0, SIBuffer.Data+2, SIBuffer.
                    Counter-2); //USRP Hack - dont print hex
```

```
                    size.
                SIBuffer.Counter = 0;
            }


            //LED_blink_fast();      //blink test LEDs
            LED_blink();
        }
}
```

B.4 Si4432.c, Si4432 Interface Program

```
//Written by: David Zhang
//Modified by: Ilya Lipnitskiy
#include "SPI.h"
#include "UART.h"
#include "OtherFunctions.h"
#include "Constants.h"
#include <LPC23xx.h>
#include <stdio.h>
#include <string.h>

#include "Si4432.h"
#include "SiRegisters.h"

//send initialize commands to transceiver
void SI_init()
{
        //transceiver status variables
        char version, device, status, mode;
        char startMsg[100];
        unsigned int i;

        #if DEBUG
        /* setup error UART port */
        serial_init(ERRPORT, 57600);
        #endif

        /* setup IRQ pin */
```

```
PINMODE8 |= 1<<17;                    //no pull-up/pull-down
    resistor on IRQ pin
PINSEL8 = 0;                          //set port 4 as GPIO ports

delay(20);                  //wait for transceiver to initialize
SI_clearInterrupts();

/* Transceiver configuration */
SI_register_write(MODE1, registers[MODE1].mode3);        //
    reset
SI_waitForInterrupt();                                   //
    wait for reset to complete
SI_clearInterrupts();                                    //
    read and clear interrupts

for(i = 0; i <= 0x7E; i++)       //reset all RW registers to
    defined default values
        if(registers[i].mode == RW)
                SI_register_write(i, registers[i].def);

//set up interrupts
SI_packetReceivedInterrupt();

//RX mode by default
SI_register_write(MODE1, registers[MODE1].mode2);
            //RX mode by default


#if DEBUG
#define INITMSG "Transceiver setup complete\n"
serial_transmit(ERRPORT, INITMSG, sizeof(INITMSG));
#endif

#if DEBUG
//get transceiver data
device = SI_register_read(DEVICE);
version = SI_register_read(VERSION);
status = SI_register_read(STATUS);
mode = SI_register_read(MODE1);
```

```
        //transmit start message
        sprintf(startMsg,
        "Device ID: %d \nRevision: %d \nStatus: %d \nMode: %d\n",
        device, version, status, mode);
        serial_transmit(ERRPORT, startMsg, strlen(startMsg));
        #endif




}

//write one byte of data to the specified register
void SI_register_write(char reg, char dat)
{
        SPI_register_write(reg, dat);
}

//read one register on the SI4432, the data returned will be
//handled by the SSP interrupt handler and stored in the global
    buffer
char SI_register_read(char reg)
{
        return SPI_register_read(reg);
}

//fills the SI4432's FIFO, then transmit the data
void SI_transmit(char* string, int size)
{
        char interrupt;
        char msg[50]; //debug buffer size

        SI_clearInterrupts();
        SI_packetSentInterrupt();                                    //
            setup interrupts

        //size MUST be an odd number
        //one byte is sent with the register, the rest of the byte is
             sent with
```

```
        //the burst write 2 bytes at a time.  If its even size, the
            last byte will
        //be padded with a 0x00 at the end, so size must also be
            incremented to avoid FIFO overflow error.
        SI_register_write(TX_HEADER2, size); //Write actual size to
            packet header (twice for USRP framer_sink algorithm)
        SI_register_write(TX_HEADER0, size);

        if(size % 2 == 0)
                size++;

        SI_register_write(TX_PACKLEN, size);              //set packet
            length

        SPI_burst_write(FIFO_ACCESS, string, size);       //fill the
            transmit FIFO

        SI_register_write(MODE1, registers[MODE1].mode1);
                    //turn on TX

        SI_waitForInterrupt();            //wait till interrupt pin is
            low (interrupt occured)

        #if DEBUG
        //transmit debug message
        interrupt = SI_register_read(INTERRUPT1);
        sprintf(msg, "\nPacket sent\nint status: %d\nsize: %d\n",
            interrupt, size);
        serial_transmit(ERRPORT, msg, strlen(msg));
        #endif

        //go back to rx mode after packet sent
        SI_clearInterrupts();
        SI_packetReceivedInterrupt();
        SI_register_write(MODE1, registers[MODE1].mode2);
}

//reads Si4432's FIFO data until FIFO empty
int SI_read(char* string)
{
```

```
        char size;                                          //the size of
             the recieved packet
        char msg[50];

        size = SI_register_read(RX_HEADER2);

        #if DEBUG
        //transmit debug message
        sprintf(msg, "\nPacket_Received:_%d_bytes\n", size);
        serial_transmit(ERRPORT, msg, strlen(msg));
        #endif

        SPI_burst_read(FIFO_ACCESS, string, size);

        SI_clearInterrupts();
        SI_packetReceivedInterrupt();
        SI_register_write(MODE1, registers[MODE1].mode2);

        return size;
}

//wait for an interrupt event to occur on transceiver, timeout in ms
int SI_waitForInterrupt()
{
        int timeout = 100000;
        while((!SI_interruptReceived()) && timeout)
                timeout--;

        return 1;
}

//return 1 if interrupt received
int SI_interruptReceived()
{
        if(IRQ_status == 0x00)
                return 1;
        else
                return 0;
}
```

```
//read interrupt registers to clear interrupts
void SI_clearInterrupts()
{
        SI_register_read(INTERRUPT1);
        SI_register_read(INTERRUPT2);
}

//enables packet sent interrupt, disables everything else
void SI_packetSentInterrupt()
{
        SI_register_write(INT_ENABLE1, 0x04);
        SI_register_write(INT_ENABLE2, 0x00);
}

//enables packet received interrupt, disables everything else
void SI_packetReceivedInterrupt()
{
        SI_register_write(INT_ENABLE1, 0x02);
        SI_register_write(INT_ENABLE2, 0x00);
}

//disables all interrupts on transceiver
void SI_disableInterrupt()
{
        SI_register_write(INT_ENABLE1, 0x00);
        SI_register_write(INT_ENABLE2, 0x00);
}
```

B.5 packet_utils.py, USRP Packet Handling Utilities

```
import struct
import numpy
from gnuradio import gru


def conv_packed_binary_string_to_1_0_string(s):
    """
    '\xAF' --> '10101111'
```

```
    """
    r = []
    for ch in s:
        x = ord(ch)
        for i in range(7,-1,-1):
            t = (x >> i) & 0x1
            r.append(t)

    return ''.join(map(lambda x: chr(x + ord('0')), r))

def conv_1_0_string_to_packed_binary_string(s):
    """
    '10101111' -> ('\xAF', False)

    Basically the inverse of conv_packed_binary_string_to_1_0_string,
    but also returns a flag indicating if we had to pad with leading
       zeros
    to get to a multiple of 8.
    """
    if not is_1_0_string(s):
        raise ValueError, "Input must be a string containing only 0's
           and 1's"

    # pad to multiple of 8
    padded = False
    rem = len(s) % 8
    if rem != 0:
        npad = 8 - rem
        s = '0' * npad + s
        padded = True

    assert len(s) % 8 == 0

    r = []
    i = 0
    while i < len(s):
        t = 0
        for j in range(8):
            t = (t << 1) | (ord(s[i + j]) - ord('0'))
        r.append(chr(t))
```

```
        i += 8
    return (''.join(r), padded)


default_access_code = \
  conv_packed_binary_string_to_1_0_string('\xAC\xDD\xA4\xE2')
preamble = \
  conv_packed_binary_string_to_1_0_string('')

def is_1_0_string(s):
    if not isinstance(s, str):
        return False
    for ch in s:
        if not ch in ('0', '1'):
            return False
    return True

def string_to_hex_list(s):
    return map(lambda x: hex(ord(x)), s)


def whiten(s, o):
    sa = numpy.fromstring(s, numpy.uint8)
    z = sa ^ random_mask_vec8[o:len(sa)+o]
    return z.tostring()

def dewhiten(s, o):
    return whiten(s, o)         # self inverse


def make_header(payload_len, whitener_offset=0):
    # Upper nibble is offset(not used for Si4432 testing), lower 12
        bits is len
    val = ((whitener_offset & 0xf) << 12) | (payload_len & 0x0fff)
    #print "offset =", whitener_offset, " len =", payload_len, " val
        =", val
    return struct.pack('!HH', val, val)

def make_packet(payload, samples_per_symbol, bits_per_symbol,
                access_code=default_access_code, pad_for_usrp=True,
```

```
                whitener_offset=0, whitening=False):
    """
    Build a packet, given access code, payload, and whitener offset

    @param payload:                 packet payload, len [0, 4096]
    @param samples_per_symbol:      samples per symbol (needed for
        padding calculation)
    @type   samples_per_symbol:     int
    @param bits_per_symbol:         (needed for padding calculation)
    @type bits_per_symbol:          int
    @param access_code:             string of ascii 0's and 1's
    @param whitener_offset          offset into whitener string to use
        [0-16)

    Packet will have access code at the beginning, followed by length
        , payload
    and finally CRC-32.
    """
    if not is_1_0_string(access_code):
        raise ValueError, "access_code must be a string containing
            only 0's and 1's (%r)" % (access_code,)

    if not whitener_offset >=0 and whitener_offset < 16:
        raise ValueError, "whitener_offset must be between 0 and 15,
            inclusive (%i)" % (whitener_offset,)

    (packed_access_code, padded) =
        conv_1_0_string_to_packed_binary_string(access_code)
    (packed_preamble, ignore) =
        conv_1_0_string_to_packed_binary_string(preamble)

    #payload_with_crc = gru.gen_and_append_crc32(payload) #Don't
        calculate CRC for testing
    payload_with_crc = payload
    #print "outbound crc =", string_to_hex_list(payload_with_crc
        [-4:])

    L = len(payload_with_crc)
    MAXLEN = len(random_mask_tuple)
    if L > MAXLEN:
```

```
            raise ValueError, "len(payload) must be in [0, %d]" % (MAXLEN
                ,)

        if whitening:
            pkt = ''.join((packed_preamble, packed_access_code,
                make_header(L, whitener_offset),
                            whiten(payload_with_crc, whitener_offset), '\
                               x55'))
        else:
            pkt = ''.join((packed_preamble, packed_access_code,
                make_header(L, whitener_offset),
                            (payload_with_crc), '\x55'))

        if pad_for_usrp:
            pkt = pkt + (_npadding_bytes(len(pkt), samples_per_symbol,
                bits_per_symbol) * '\x55')

        #print "make_packet: len(pkt) =", len(pkt)
        return pkt

    def _npadding_bytes(pkt_byte_len, samples_per_symbol, bits_per_symbol
        ):
        """
        Generate sufficient padding such that each packet ultimately ends
        up being a multiple of 512 bytes when sent across the USB.  We
        send 4-byte samples across the USB (16-bit I and 16-bit Q), thus
        we want to pad so that after modulation the resulting packet
        is a multiple of 128 samples.

        @param ptk_byte_len: len in bytes of packet, not including
            padding.
        @param samples_per_symbol: samples per bit (1 bit / symbolwidth
            GMSK)
        @type samples_per_symbol: int
        @param bits_per_symbol: bits per symbol (log2(modulation order))
        @type bits_per_symbol: int

        @returns number of bytes of padding to append.
        """
        modulus = 128
```

```
    byte_modulus = gru.lcm(modulus/8, samples_per_symbol) *
        bits_per_symbol / samples_per_symbol
    r = pkt_byte_len % byte_modulus
    if r == 0:
        return 0
    return byte_modulus - r


def unmake_packet(whitened_payload_with_crc, whitener_offset=0,
    dewhitening=False):
    """
    Return (ok, payload)

    @param whitened_payload_with_crc: string
    """

    if dewhitening:
        payload_with_crc = dewhiten(whitened_payload_with_crc,
            whitener_offset)
    else:
        payload_with_crc = (whitened_payload_with_crc)

    #ok, payload = gru.check_crc32(payload_with_crc) #Disable CRC
        check for testing purposes

    if 0:
        print "payload_with_crc =", string_to_hex_list(
            payload_with_crc)
        print "ok = %r, len(payload) = %d" % (ok, len(payload))
        print "payload =", string_to_hex_list(payload)

    return False, payload_with_crc


# FYI, this PN code is the output of a 15-bit LFSR
random_mask_tuple = (
  '''omitted''')

random_mask_vec8 = numpy.array(random_mask_tuple, numpy.uint8)
```

B.6 pkt.py, USRP Packet Modulating/Demodulating Block

```
from math import pi
from gnuradio import gr, packet_utils
import gnuradio.gr.gr_threading as _threading


# ////////////////////////////////////
#                     mod/demod with packets as i/o
# ////////////////////////////////////

class mod_pkts(gr.hier_block2):
    """
    Wrap an arbitrary digital modulator in our packet handling
        framework.

    Send packets by calling send_pkt
    """
    def __init__(self, modulator, access_code=None, msgq_limit=2,
        pad_for_usrp=True, use_whitener_offset=False):
        """
        Hierarchical block for sending packets

        Packets to be sent are enqueued by calling send_pkt.
        The output is the complex modulated signal at baseband.

        @param modulator: instance of modulator class (gr_block or
            hier_block2)
        @type modulator: complex baseband out
        @param access_code: AKA sync vector
        @type access_code: string of 1's and 0's between 1 and 64
            long
        @param msgq_limit: maximum number of messages in message
            queue
        @type msgq_limit: int
        @param pad_for_usrp: If true, packets are padded such that
            they end up a multiple of 128 samples
```

```
        @param use_whitener_offset: If true, start of whitener XOR
            string is incremented each packet

        See gmsk_mod for remaining parameters
        """

        gr.hier_block2.__init__(self, "mod_pkts",
                                gr.io_signature(0, 0, 0),
                                                # Input
                                    signature
                                gr.io_signature(1, 1, gr.
                                    sizeof_gr_complex)) # Output
                                    signature

        self._modulator = modulator
        self._pad_for_usrp = pad_for_usrp
        self._use_whitener_offset = use_whitener_offset
        self._whitener_offset = 0

        if access_code is None:
            access_code = packet_utils.default_access_code
        if not packet_utils.is_1_0_string(access_code):
            raise ValueError, "Invalid access_code %r. Must be string
                of 1's and 0's" % (access_code,)
        self._access_code = access_code

        # accepts messages from the outside world
        self._pkt_input = gr.message_source(gr.sizeof_char,
            msgq_limit)
        self.connect(self._pkt_input, self._modulator, self)

    def send_pkt(self, payload='', eof=False):
        """
        Send the payload.

        @param payload: data to send
        @type payload: string
        """
        if eof:
```

```
            msg = gr.message(1) # tell self._pkt_input we're not
                sending any more packets
        else:
            # print "original_payload =", string_to_hex_list(payload)
            pkt = packet_utils.make_packet(payload,
                                           self._modulator.
                                               samples_per_symbol(),
                                           self._modulator.
                                               bits_per_symbol(),
                                           self._access_code,
                                           self._pad_for_usrp,
                                           self._whitener_offset)
            #print "pkt =", string_to_hex_list(pkt)
            msg = gr.message_from_string(pkt)
            if self._use_whitener_offset is True:
                self._whitener_offset = (self._whitener_offset + 1) %
                    16

        self._pkt_input.msgq().insert_tail(msg)



class demod_pkts(gr.hier_block2):
    """
    Wrap an arbitrary digital demodulator in our packet handling
        framework.

    The input is complex baseband.  When packets are demodulated,
        they are passed to the
    app via the callback.
    """

    def __init__(self, demodulator, access_code=None, callback=None,
        threshold=-1):
        """
        Hierarchical block for demodulating and deframing packets.

        The input is the complex modulated signal at baseband.
        Demodulated packets are sent to the handler.
```

```
@param demodulator: instance of demodulator class (gr_block
    or hier_block2)
@type demodulator: complex baseband in
@param access_code: AKA sync vector
@type access_code: string of 1's and 0's
@param callback:  function of two args: ok, payload
@type callback: ok: bool; payload: string
@param threshold: detect access_code with up to threshold
    bits wrong (-1 -> use default)
@type threshold: int
"""

gr.hier_block2.__init__(self, "demod_pkts",
                        gr.io_signature(1, 1, gr.
                            sizeof_gr_complex), # Input
                            signature
                        gr.io_signature(0, 0, 0))
                                        # Output
                            signature

self._demodulator = demodulator
if access_code is None:
    access_code = packet_utils.default_access_code
if not packet_utils.is_1_0_string(access_code):
    raise ValueError, "Invalid access_code %r. Must be string
        of 1's and 0's" % (access_code,)
self._access_code = access_code

if threshold == -1:
    threshold = 12              # FIXME raise exception

self._rcvd_pktq = gr.msg_queue()          # holds packets
    from the PHY
self.correlator = gr.correlate_access_code_bb(access_code,
    threshold)

self.framer_sink = gr.framer_sink_1(self._rcvd_pktq)
self.connect(self, self._demodulator, self.correlator, self.
    framer_sink)
```

```
        self._watcher = _queue_watcher_thread(self._rcvd_pktq,
            callback)

class _queue_watcher_thread(_threading.Thread):
    def __init__(self, rcvd_pktq, callback):
        _threading.Thread.__init__(self)
        self.setDaemon(1)
        self.rcvd_pktq = rcvd_pktq
        self.callback = callback
        self.keep_running = True
        self.start()


    def run(self):
        while self.keep_running:
            msg = self.rcvd_pktq.delete_head()
            ok, payload = packet_utils.unmake_packet(msg.to_string(),
                int(msg.arg1()))
            if self.callback:
                self.callback(ok, payload)
```

B.7 gmsk.py, USRP GMSK Modulator/Demodulator

```
# See gnuradio-examples/python/digital for examples

from gnuradio import gr
from gnuradio import modulation_utils
from math import pi
import numpy
from pprint import pprint
import inspect

# default values (used in __init__ and add_options)
_def_samples_per_symbol = 2
_def_bt = 0.35
_def_verbose = False
_def_log = False

_def_gain_mu = None
```

```
_def_mu = 0.5
_def_freq_error = 0.0
_def_omega_relative_limit = 0.005


# ////////////////////////////////////////
#                                    GMSK modulator
# ////////////////////////////////////////

class gmsk_mod(gr.hier_block2):

    def __init__(self,
                 samples_per_symbol=_def_samples_per_symbol,
                 bt=_def_bt,
                 verbose=_def_verbose,
                 log=_def_log):
        """
        Hierarchical block for Gaussian Minimum Shift Key (GMSK)
        modulation.

        The input is a byte stream (unsigned char) and the
        output is the complex modulated signal at baseband.

        @param samples_per_symbol: samples per baud >= 2
        @type samples_per_symbol: integer
        @param bt: Gaussian filter bandwidth * symbol time
        @type bt: float
        @param verbose: Print information about modulator?
        @type verbose: bool
        @param debug: Print modualtion data to files?
        @type debug: bool
        """

        gr.hier_block2.__init__(self, "gmsk_mod",
                                gr.io_signature(1, 1, gr.sizeof_char)
                                    ,       # Input signature
                                gr.io_signature(1, 1, gr.
                                    sizeof_gr_complex)) # Output
                                    signature
```

```
self._samples_per_symbol = samples_per_symbol
self._bt = bt

if not isinstance(samples_per_symbol, int) or
    samples_per_symbol < 2:
     raise TypeError, ("samples_per_symbol must be an integer
        >= 2, is %r" % (samples_per_symbol,))

ntaps = 4 * samples_per_symbol                # up to 3
    bits in filter at once
sensitivity = (pi / 2) / samples_per_symbol   # phase
    change per bit = pi / 2

# Turn it into NRZ data.
self.nrz = gr.bytes_to_syms()

# Form Gaussian filter
# Generate Gaussian response (Needs to be convolved with
    window below).
self.gaussian_taps = gr.firdes.gaussian(
        1,                    # gain
        samples_per_symbol,   # symbol_rate
        bt,                   # bandwidth * symbol time
        ntaps                 # number of taps
        )

self.sqwave = (1,) * samples_per_symbol       # rectangular
    window
self.taps = numpy.convolve(numpy.array(self.gaussian_taps),
    numpy.array(self.sqwave))
self.gaussian_filter = gr.interp_fir_filter_fff(
    samples_per_symbol, self.taps)

# FM modulation
self.fmmod = gr.frequency_modulator_fc(sensitivity)

if verbose:
     self._print_verbage()

if log:
```

```
        self._setup_logging()

    # Connect & Initialize base class
    self.connect(self, self.nrz, self.gaussian_filter, self.fmmod
        , self)

def samples_per_symbol(self):
    return self._samples_per_symbol

def bits_per_symbol(self=None):     # staticmethod that's also
    callable on an instance
    return 1
bits_per_symbol = staticmethod(bits_per_symbol)      # make it a
    static method.


def _print_verbage(self):
    print "bits per symbol = %d" % self.bits_per_symbol()
    print "Gaussian filter bt = %.2f" % self._bt


def _setup_logging(self):
    print "Modulation logging turned on."
    self.connect(self.nrz,
                 gr.file_sink(gr.sizeof_float, "nrz.dat"))
    self.connect(self.gaussian_filter,
                 gr.file_sink(gr.sizeof_float, "gaussian_filter.
                     dat"))
    self.connect(self.fmmod,
                 gr.file_sink(gr.sizeof_gr_complex, "fmmod.dat"))


def add_options(parser):
    """
    Adds GMSK modulation-specific options to the standard parser
    """
    parser.add_option("", "--bt", type="float", default=_def_bt,
                      help="set bandwidth-time product [default=%
                          default] (GMSK)")
add_options=staticmethod(add_options)
```

```
    def extract_kwargs_from_options(options):
        """
        Given command line options, create dictionary suitable for
            passing to __init__
        """
        return modulation_utils.extract_kwargs_from_options(gmsk_mod.
            __init__,
                                                            ('self',)
                                                                ,
                                                                options
                                                                )
    extract_kwargs_from_options=staticmethod(
        extract_kwargs_from_options)



# ////////////////////////////////////////////////
#                                GMSK demodulator
# ////////////////////////////////////////////////

class gmsk_demod(gr.hier_block2):

    def __init__(self,
                 samples_per_symbol=_def_samples_per_symbol,
                 gain_mu=_def_gain_mu,
                 mu=_def_mu,
                 omega_relative_limit=_def_omega_relative_limit,
                 freq_error=_def_freq_error,
                 verbose=_def_verbose,
                 log=_def_log):
        """
        Hierarchical block for Gaussian Minimum Shift Key (GMSK)
        demodulation.

        The input is the complex modulated signal at baseband.
        The output is a stream of bits packed 1 bit per byte (the LSB
            )
```

```
@param samples_per_symbol: samples per baud
@type samples_per_symbol: integer
@param verbose: Print information about modulator?
@type verbose: bool
@param log: Print modualtion data to files?
@type log: bool

Clock recovery parameters.  These all have reasonble defaults
    .

@param gain_mu: controls rate of mu adjustment
@type gain_mu: float
@param mu: fractional delay [0.0, 1.0]
@type mu: float
@param omega_relative_limit: sets max variation in omega
@type omega_relative_limit: float, typically 0.000200 (200
    ppm)
@param freq_error: bit rate error as a fraction
@param float
"""

gr.hier_block2.__init__(self, "gmsk_demod",
                        gr.io_signature(1, 1, gr.
                            sizeof_gr_complex), # Input
                            signature
                        gr.io_signature(1, 1, gr.sizeof_char)
                            )       # Output signature

self._samples_per_symbol = samples_per_symbol
self._gain_mu = gain_mu
self._mu = mu
self._omega_relative_limit = omega_relative_limit
self._freq_error = freq_error

if samples_per_symbol < 2:
    raise TypeError, "samples_per_symbol >= 2, is %f" %
        samples_per_symbol

self._omega = samples_per_symbol*(1+self._freq_error)
```

```
        if not self._gain_mu:
            self._gain_mu = 0.175

        self._gain_omega = .25 * self._gain_mu * self._gain_mu
                # critically damped

        # Demodulate FM
        sensitivity = (pi / 2) / samples_per_symbol
        self.fmdemod = gr.quadrature_demod_cf(1.0 / sensitivity)

        # the clock recovery block tracks the symbol clock and
            resamples as needed.
        # the output of the block is a stream of soft symbols (float)
        self.clock_recovery = gr.clock_recovery_mm_ff(self._omega,
            self._gain_omega,
                                                      self._mu, self.
                                                          _gain_mu,
                                                      self.
                                                          _omega_relative_limit
                                                          )


        # slice the floats at 0, outputting 1 bit (the LSB of the
            output byte) per sample
        self.slicer = gr.binary_slicer_fb()

        if verbose:
            self._print_verbage()

        if log:
            self._setup_logging()

        # Connect & Initialize base class
        self.connect(self, self.fmdemod, self.clock_recovery, self.
            slicer, self)

    def samples_per_symbol(self):
        return self._samples_per_symbol

    def bits_per_symbol(self=None):   # staticmethod that's also
        callable on an instance
```

```
    return 1
bits_per_symbol = staticmethod(bits_per_symbol)       # make it a
    static method.


def _print_verbage(self):
    print "bits per symbol = %d" % self.bits_per_symbol()
    print "M&M clock recovery omega = %f" % self._omega
    print "M&M clock recovery gain mu = %f" % self._gain_mu
    print "M&M clock recovery mu = %f" % self._mu
    print "M&M clock recovery omega rel. limit = %f" % self.
        _omega_relative_limit
    print "frequency error = %f" % self._freq_error


def _setup_logging(self):
    print "Demodulation logging turned on."
    self.connect(self.fmdemod,
                 gr.file_sink(gr.sizeof_float, "fmdemod.dat"))
    self.connect(self.clock_recovery,
                 gr.file_sink(gr.sizeof_float, "clock_recovery.dat
                     "))
    self.connect(self.slicer,
                 gr.file_sink(gr.sizeof_char, "slicer.dat"))

def add_options(parser):
    """
    Adds GMSK demodulation-specific options to the standard
        parser
    """
    parser.add_option("", "--gain-mu", type="float", default=
        _def_gain_mu,
                      help="M&M clock recovery gain mu [default=%
                          default] (GMSK/PSK)")
    parser.add_option("", "--mu", type="float", default=_def_mu,
                      help="M&M clock recovery mu [default=%
                          default] (GMSK/PSK)")
    parser.add_option("", "--omega-relative-limit", type="float",
         default=_def_omega_relative_limit,
```

```
                            help="M&M clock recovery omega relative
                                limit [default=%default] (GMSK/PSK)")
        parser.add_option("", "--freq-error", type="float", default=
            _def_freq_error,
                            help="M&M clock recovery frequency error [
                                default=%default] (GMSK)")
    add_options=staticmethod(add_options)

    def extract_kwargs_from_options(options):
        """
        Given command line options, create dictionary suitable for
            passing to __init__
        """
        return modulation_utils.extract_kwargs_from_options(
            gmsk_demod.__init__,
                                                            ('self',)
                                                                ,
                                                                options
                                                                )
    extract_kwargs_from_options=staticmethod(
        extract_kwargs_from_options)


#
# Add these to the mod/demod registry
#
modulation_utils.add_type_1_mod('gmsk', gmsk_mod)
modulation_utils.add_type_1_demod('gmsk', gmsk_demod)
```

Bibliography

[1] Silicon Labs (2009) *Si4430/31/32 ISM Transceiver Data Sheet*.

[2] "What is GMSK Modulation - Gaussian Minimum Shift Keying," `http://www.radio-electronics.com/info/rf-technology-design/pm-phase-modulation/what-is-gmsk-gaussian-minimum-shift-keying-tutorial.php`.

[3] The Pennsylvania State University: Student Space Programs Laboratory (2009) *OSIRIS Lite Balloon Payload Preliminary Design Review*.

[4] Blossom, E. (2004) "GNU radio: tools for exploring the radio frequency spectrum," *Linux Journal*, **2004**(122), p. 4.

[5] Mitola III, J. (1993) "Software radios: Survey, critical evaluation and future directions," *IEEE Aerospace and Electronic Systems Magazine*, **8**(4), pp. 25–36.

[6] ——— (1995) "The Software Radio Architecture," *IEEE Communications Magazine*, pp. 26–38.

[7] "GNU Radio - Overview - gnuradio.org," `http://gnuradio.org/redmine/projects/show/gnuradio`.

[8] Barrett, C. (1999) *Fractional/Integer-N PLL Basics*, Texas Instruments: Wireless Communication Business Unit.

[9] K.H. Mueller, M. M. (1976) "Timing Recovery in Digital Synchronous Data Receivers," *IEEE Transactions on Communications*, **COM-24**(5).

[10] G.R. Danesfahani, T. J. (1995) "Optimisation of modified Mueller and Müller algorithm," *Electronics Letters*, **31**(13).

[11] Stevenson, B. (2009), "Benchmark-rx/tx.py with QPSK modulation," `http://www.mail-archive.com/discuss-gnuradio@gnu.org/msg19241.html`.

[12] Silicon Labs (2009) *AN440: Si4430/31/32 Register Descriptions*.

Vita

Ilya Y. Lipnitskiy

Education:
The Pennsylvania State University, Schreyer Honors College
Bachelor of Science in Computer Engineering and Electrical Engineering,
Honors in Electrical Engineering
Awards/Activities:
CSE Curriculum Committee Student Representative
Raytheon State College Scholarship in Computer Science and Engineering
Dean's List
Nominated for the 2007-2008 Co-op Student of the Year Award
Relevant Courses:
Software Defined Radio, FPGA design, Microcontroller design, Operating systems
Graduate-level course in computer networks (CSE 514)
Work Experience:
Summer internship
SoleNet, Inc., Gaithersburg, MD May 2009-August 2009
Engineering International Co-op & Internship Program
Imperial Energy Corporation PLC, Tomsk, Russia August 2007-July 2008
Computer Experience/Skills:
C/C++, Assembly, MATLAB, Verilog HDL, Java, PHP, Python, MySQL, HTML
Language:
Fluent in English and Russian; reading level competence in German