GNU RADIO TESTBED

by

Naveen Manicka

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Spring 2007

© 2007 Naveen Manicka
All Rights Reserved
GNU RADIO TESTBED

by

Naveen Manicka

Approved:

Chien-Chung Shen, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved:

David Saunders, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved:

Tom M. Apple, Ph.D.
Dean of the College of Arts and Sciences

Approved:

Carolyn A. Thoroughgood, Ph.D.
Vice Provost for Research and Graduate Studies
ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Professor Chien-Chung Shen, for introducing me to the very interesting world of wireless networks, for encouraging me to pursue my research topic and for his guidance over the course of my time at the University. While giving his students a large degree of independence and flexibility to manage their time and projects, he is always available and willing to help in any way possible. Whether reviewing presentations, acquiring the necessary equipment or brainstorming about projects, his enthusiasm for new ideas and genuine interest in student’s work are admirable and deeply appreciated.

I would also like to thank my current and former fellow colleagues at the DEGAS Networking Lab, Sundaram Rajagopalan aka Sonny, Ilknur Aydin, Justin Yackoski, Xiaofeng Han and Ke Li, for making my time in the lab such an enjoyable and memorable experience. Sonny and Ilknur helped me in the process of becoming a member of the lab when I first joined and were always a source of good advice and discussion. Sonny always had a witty or interesting comment to make, which helped during the course of a busy day in the lab. Special thanks to every one for putting up with my initial radio experiments (lots of static and some music) and the soldering (frequently filling the lab with the very pleasant odor of lead and solder). Thanks to all of you for making my time at DEGAS special.
I would additionally like to acknowledge the CIS Department faculty, especially, Dr. Paul Amer, Dr. Errol Lloyd, Dr. Martin Swany and Dr. Kathleen McCoy, for their exceptional teaching and guidance in the classes I took under them.

I would also like to thank my friends and family for all the support over the years, and particularly the last several months of intense work on this research project. I appreciate you putting up with my limited amount of free time, long days at the lab, delays before returning phone calls, and frequent (and often unsolicited!) technical talk. I would like to give special thanks to my sister and brother-in-law, Vasuki and Vinay, who are also UD Graduates, for their support and guidance during my transfer to UD from Colorado and during the course of my stay here.

Last but certainly not the least, I’d especially like to thank my parents, Dhanapal and Kalaivani, for taking interest in my work and encouraging me from my toy racing car days all the way through graduate school. Thanks for providing me with the tools I needed to succeed and for supporting me along the way. You guys are the best!
# TABLE OF CONTENTS

LIST OF TABLES ........................................................................................................ viii
LIST OF FIGURES ......................................................................................................... ix
ABSTRACT ..................................................................................................................... xi

Chapter

1 INTRODUCTION ....................................................................................................... 12

1.1 Radio Basics ....................................................................................................... 12

1.1.1 Radio Frequency......................................................................................... 12
1.1.2 Radio Transmission ................................................................................... 14

1.2 Hardware Radio Shortcomings........................................................................... 16

1.3 Thesis Organization ......................................................................................... 17

2 SOFTWARE DEFINED RADIO ........................................................................ 18

2.1 Introduction ....................................................................................................... 18
2.2 Digital Signal Sampling ................................................................................... 20

2.2.1 RF Digitization........................................................................................... 21
2.2.2 IF Digitization ........................................................................................... 23
2.2.3 Baseband Digitization ............................................................................... 23

3 GNU RADIO ....................................................................................................... 24

3.1 GNU Radio Basics ........................................................................................... 24
3.2 GNU Radio Hello World ................................................................................... 26

3.2.1 Code Walkthrough .................................................................................... 27
3.2.2 Summary .................................................................................................... 28

3.3 USRP (Universal Software Radio Peripheral) ............................................... 28

3.3.1 USRP Features ............................................................................................ 30
4.5.1 Filtering A Noisy Channel .......................................................... 58
4.5.2 QAM Modulation / Demodulation .............................................. 62

5 MAC PROTOCOL IMPLEMENTATION FRAMEWORK ....................... 64

5.1 Motivation ..................................................................................... 64
5.2 Using QualNet Code ...................................................................... 64
5.3 Drawing Parallels ......................................................................... 66
5.4 Packet Processing Blocks .............................................................. 69

5.4.1 Packet Received From PHY Layer ......................................... 69
5.4.2 Network Layer Has Packet To Send ....................................... 69
5.4.3 Packet Format ......................................................................... 71
5.4.4 Packet Modulation and Demodulation ................................... 71

5.4 Related Work ................................................................................ 72

5.4.1 Emulab Based GNU Radio Testbed ..................................... 72
5.4.2 Evaluating A GNU Radio Testbed ......................................... 73
5.4.3 Flexible MAC/PHY Multihop Testbed ................................. 73

6 MATLAB INTERFACE ...................................................................... 74

6.1 Tunnel.py Command Line Options ........................................... 74
6.2 Matlab GUI For Running The Program .................................... 75
6.3 The Logging Option ................................................................... 77
6.4 Supported Modulation Techniques .......................................... 80

7 CONCLUSION AND FUTURE WORK ............................................. 86

8 BIBLIOGRAPHY ............................................................................. 88

Appendix

A CODE – MATLAB INTERFACE – TUNNEL.PY .......................... 90
B CODE – MATLAB GRAPH GUI ..................................................... 96
# LIST OF TABLES

Table 1.1  Radio frequency ranges ................................................................. 12

Table 3.1  Dial tone Python code ................................................................. 26

Table 5.1  Real-time scheduling support in tunnel.py ................................. 68

Table 5.2  Packet receive code snippet ....................................................... 69

Table 5.3  Packet send code snippet ............................................................ 70

Table 6.1  Callback function of the matlab interface .................................... 76

Table 6.2  The logging using file sinks ....................................................... 78

Table 6.3  Read_complex_binary function .................................................. 79

Table 6.4  Callback function for GMSK GUI ................................................. 83
LIST OF FIGURES

Figure 1.1 Typical GSM receiver architecture ........................................................ 16
Figure 1.2 Mobility Vs Speed.................................................................................. 17
Figure 2.1 Typical radio receiver............................................................................. 19
Figure 2.2 Typical software radio block diagram.................................................... 19
Figure 2.3 Sampling of an analog signal ................................................................. 20
Figure 3.1 Block diagram of GNU Radio components ........................................... 25
Figure 3.2 Transmit and receive paths................................................................. 25
Figure 3.3 A USRP motherboard and four daughterboards ................................. 29
Figure 3.4 USRP block diagram............................................................................ 32
Figure 3.5 MUX block diagram............................................................................ 35
Figure 3.6 Digital down converter block diagram.............................................. 36
Figure 4.1 Channel Noise Generation ................................................................. 59
Figure 4.2 Channel Noise Parameters ............................................................... 60
Figure 4.3 Channel Noise FFT Output ............................................................... 61
Figure 4.4 QAM Modulator - Demodulator .......................................................... 63
Figure 5.1 GNU Radio MAC protocol framework using wrapper functions ........ 65
Figure 6.1 Snapshot of the Matlab GUI interface for tunnel.py......................... 75
Figure 6.2 Initial screen for GMSK based logging............................................. 81
Figure 6.3 Plots of logged GMSK data ................................................................. 82
Figure 6.4  Plots of logged DQPSK data ................................................................. 84
Figure 6.5  Plots of logged DBPSK data ................................................................. 85
ABSTRACT

A software-defined radio (SDR) system is a radio communication system which can tune to any frequency band and receive different modulations across a large frequency spectrum by means of a programmable hardware which is controlled by software. An SDR performs significant amounts of signal processing in a general purpose computer, or a reconfigurable piece of digital electronics. SDR’s can provide the protocol engineer with wireless testbeds that are fully programmable at the DLC, MAC and PHY. The benefit of this flexibility highly depends on the performance and usability of the specific SDR.

In this thesis, we seek to explore the viability of using GNU Radio (an open source SDR implementation) as a testbed to test MAC protocols developed in a simulation environment and develop a framework to reuse the simulation code, with minimal modifications. Also, document a beginner’s guide to the installation and use of GNU Radio.
1.1 Radio Basics

1.1.1 Radio Frequency

Radio frequency, or RF, refers to that portion of the electromagnetic spectrum in which electromagnetic waves can be generated by alternating current fed to an antenna. Any device which works within the RF region is called a radio. Many wireless technologies are based on RF propagation.

Table 1.1  Radio frequency ranges
(http://en.wikipedia.org/wiki/Radio_frequency)

<table>
<thead>
<tr>
<th>Band name</th>
<th>Abbr</th>
<th>Frequency Wavelength</th>
<th>Example uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extremely low</td>
<td>ELF</td>
<td>3–30 Hz 100,000 km – 10,000 km</td>
<td>Communication with submarines</td>
</tr>
<tr>
<td>Super low</td>
<td>SLF</td>
<td>30–300 Hz</td>
<td>Communication with submarines</td>
</tr>
<tr>
<td>Frequency</td>
<td>ULF</td>
<td>LF</td>
<td>MF</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>Frequency</td>
<td>10,000 km – 1000 km</td>
<td>300–3000 Hz 1000 km – 100 km</td>
<td>30–300 kHz 10 km – 1 km</td>
</tr>
<tr>
<td>Communication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>within mines</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Submarine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>communication</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>avalanche beacons</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wireless heart</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rate monitors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>geophysics</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Navigation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>time signals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM longwave</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>broadcasting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM (Medium-wave)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>broadcasts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shortwave</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>broadcasts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and over-the-horizon aviation communications</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FM, television</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>broadcasts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and line-of-sight</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ground-to-aircraft</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and aircraft-to-aircraft communications</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>television</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>broadcasts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mobile phones</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wireless LAN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bluetooth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and Two-Way Radios such as FRS and GMRS Radios</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>microwave</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>devices</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wireless LAN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>most modern Radars</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radio astronomy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high-speed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>microwave radio relay</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>relay</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1.1.2 Radio Transmission

Information in radio waves is transmitted by modulating a carrier signal of higher frequency (usually in the range of MHz or GHz). Receivers extract information from these waves and present it as audio, video or any other suitable form to the end user. This process requires multiple steps that are carried out in a chain of hardware components.

Typically an antenna receives the radio waves in the MHz or GHz bands. Consider the case of GSM networks. Most GSM networks operate in the 900 MHz or 1800 MHz bands. Some countries in the Americas (including the United States and Canada) use the 850 MHz and 1900 MHz bands because the 900 and 1800 MHz frequency bands have already been allocated.

In the 900 MHz band, the uplink frequency band is 890-915 MHz, and the downlink frequency band is 935-960 MHz. This 25 MHz bandwidth is subdivided into 124 carrier frequency channels, each spaced 200 kHz apart.

The antenna passes the RF signal to the receiver. Since the RF signal contains a mix of channels, it is difficult and expensive to decode the information directly. Instead the decoding is done in a series of steps, described briefly below and then the required signal is extracted.

As shown in Figure 1.1, the first stage is to separate the required frequency bandwidth using an RF filter. The filter selects the desired channel bandwidth, in this case, 200 kHz, but since the communication channel can change, the filter must be tunable. Accurate tunable filters are expensive; hence cheaper filters are used at the RF stage. Consequently, the output is not strictly 200 kHz and can contain parts of the adjacent channels. To eliminate the adjacent channels, the signal is first down converted to a lower and fixed frequency called intermediate frequency
(IF). The required channel would be down converted from, say 900 MHz to 3 MHz. At this point, the signal is filtered again by an IF filter to eliminate adjacent components. Since all channels are down converted to the same IF, the filter does not need to be tunable and can be highly accurate at lower cost. Information is easily extracted from IF through demodulation techniques. The transmission pipeline is similar to the reception pipeline but follows the reverse path. The generated signal is modulated into an intermediate frequency and up converted to the required RF frequency and transmitted by the antenna.

As an example, the real-world GSM handset includes more than this simple series of functionality. The GSM handset includes a powerful digital signal processor (DSP) core to encode, encrypt, interleave, packetize, transmit, receive, de-packetize, de-interleave, de-encrypt, and de-encode the data going to and coming from the voiceband A/D and D/A converters. An equally powerful microcontroller, combined with a hardware burst processor, controls the timing necessary to implement the time-division multiple-access (TDMA) and frequency hopping functions to keep the phone call on a specific time and frequency channel. The microcontroller also implements the man-machine interface, and operates all the necessary protocols for communication to the base stations.
1.2 Hardware Radio Shortcomings

The hardware-based approach to traditional radio design imposes a set of limitations. First, traditional radios have low flexibility to adapt to new services and standards. Since each hardware element of the radio chain performs a specific function, these components are designed to operate in a particular frequency band (RF) and standard. When the frequency or any of the parameters of the standard changes, traditional radios cannot correctly decode the information. Before being able to operate under the new conditions, the system must be redesigned and hardware modules have to be replaced. Since redesigning, manufacturing and replacing hardware components imply higher costs and longer time-to-market, traditional radios suffer longer time-to-market and higher costs for the development and manufacturing of new products. Also, because of this, traditional radios are limited in the number of services they can provide.
Consider enabling multiple services in a single radio device, for example, GSM, WiMax (802.16) and WiFi (802.11), to allow for seamless data connections, though at different data rates. This device would require separate chains of hardware elements to be integrated, one for each standard. However, the physical limitations of the phone, including space and battery, would limit the number of services that could be integrated in one device.

1.3 Thesis Organization

Chapters 1 and 2 give an introduction to the concepts of radio and SDR. Chapter 3 talks about GNU Radio, its features and an installation and use guide. Chapter 4 introduces an interesting tool, the GNU Radio Companion. Chapters 5 and 6 deal with the testbed framework and the associated Matlab interface.
Chapter 2
SOFTWARE DEFINED RADIO

2.1 Introduction

Contrary to traditional technology, SDR follows a software-based approach that could remove drawbacks of current radios. Software pieces and not hardware components treat the signals to extract the information. A typical radio receiver is shown in Figure 2.1.

The idea behind software-defined radio is to do all that modulation and demodulation with software instead of using dedicated circuitry. The most obvious benefit is that instead of having to build extra circuitry to handle different types of radio signals, you can just load an appropriate program. One instant your computer could be an AM radio, the next a wireless data transceiver and then perhaps a HDTV set. This flexibility of software could be leveraged to do things that are difficult, if not impossible, with traditional radio setups.
Figure 2.1  Typical radio receiver

The method of extracting information in software requires the received signal to be transformed into the digital domain for processing (i.e., A/D Conversion), in contrast the transmit path does the opposite (i.e., D/A Conversion) while sending the signal to the antenna, as shown in Figure 2.2.

Figure 2.2  Typical software radio block diagram
2.2 Digital Signal Sampling

Sampling or digitization is the process of converting an analog signal (a function of continuous time or space) into a numeric sequence (a function of discrete time or space). This process is done in accordance with the Nyquist Criterion, which states

"Exact reconstruction of a continuous-time baseband signal from its samples is possible if the signal is bandlimited and the sampling frequency is greater than twice the signal bandwidth."

Then the condition for exact reconstructability from samples at a uniform sampling rate (in samples per unit time) is: $f_s > 2B$

![Figure 2.3 Sampling of an analog signal](image)
Signal processing techniques process the samples to extract the information. Ideally, digitization right after the antenna, i.e. before the RF front end, is the best approach, since it allows processing the signal fully in software. However, this kind of digitization is currently impossible to implement due to the state-of-the-art of analog-digital converters (A/D) and the limitations on computational capacity of contemporary processors. Digitization may take place at other points in the traditional radio architecture: after the IF (Intermediate Frequency) filter or after the demodulator at the baseband stage. Traditional radios use no digitization or baseband digitization. IF digitization is the solution currently implemented in SDRs.

2.2.1 RF Digitization

In RF digitization, an analog-digital converter (A/D) digitizes the radio waves collected at the antenna. Signal processing software extracts the information from the digital samples. In this manner, A/D converters, general purpose processors and signal processing software replace the whole hardware based radio chain.

This approach is highly flexible and ideal because the same piece of equipment may be used for any new frequency, standard and application with simple software upgrades but is limited by the present state of the art of A/D converters and the limitations on computational capacity of present processors. Present A/D converters are limited in speed and resolution at high frequencies such as GHz. Moreover, when A/D converters are placed right after the antenna, sampling is done over signals with very different strengths: the dynamic range of the signals may vary from μvolts to volts. Current A/D resolutions are not able to cover such dynamic ranges. Nevertheless, significant research efforts are taking place to surmount this
problem. Recently, Analog Devices (http://www.analog.com/) announced that they have developed a technique to eliminate IF digitization (called Othello) and the Othello-G radio chip shrinks a complete quad-band radio for cellular telephones to a mere 1.5 square centimeters – 30 percent smaller than anything available on the market. The Othello-G radio design also uses approximately 75 percent fewer components than its predecessor, leading to significant reductions in bill-of-materials (BOM) costs for cellular handset designers. (http://www.analog.com/en/press/0,2890,3%255F%255F96822,00.html, Apr. 2006)

Speed and power consumption are also a tradeoff in A/D converters. Fast A/D converters exhibit higher power consumption than slower ones. If power consumption is very high, the A/D converter could dissipate too much heat and overheat the device. This issue is particularly critical in mobile devices, where cooling systems cannot be installed and the battery life is an extremely limiting factor. In fact, for mobile devices, the A/D power consumption should be within the range of 50 to 150 mw. Currently, there are two trends in the A/D research. Some researchers direct their efforts to achieve high speeds. Others focus on reducing the power consumption. Another problem concerns computational capacity. When placing an A/D right after the antenna, the converter will be required to digitize the whole band (from baseband to several GHz). Such wideband receivers are still not a viable option with cost effective solutions. Also, the software must filter the samples to select the targeted signals. Such filtering has enormous computational cost that can be only achieved using multiple processors.
2.2.2  **IF Digitization**

To surmount the present problems of RF digitization, SDR designers place A/D converters after the IF stage. This design requires an RF front-end, which consists of a RF filter, a RF/IF converter and an IF filter. The RF front-end selects and converts the signal to IF as in traditional radios. Before demodulation, an A/D converter digitizes the signal. Signal processing software module then extracts the information.

There are two are the main advantages of this configuration. Firstly, current A/D converters can achieve enough speed and resolution at IF frequencies. Secondly, this design requires less computational resources because the tunable RF filter of the front-end limits the number of received channels which reduces the burden of software channel selection.

2.2.3  **Baseband Digitization**

Digitization at baseband level is common in traditional transceivers. Information extracted from the analog signal and baseband sampling is used in subsequent stages to profit from signal processing techniques such as music equalization. This is a common practice in widely used devices such a stereo music equipment, etc. Because none of the radio functions for information extraction is carried out in software, radios using baseband digitization are not considered to be software defined radios, but traditional equipment with a software component to fine tune the demodulated signal, which are termed Software Controlled Radio (SCR).
Chapter 3
GNU RADIO

3.1 GNU Radio Basics

GNU Radio is an open source software toolkit which provides a library of signal processing blocks and the glue to tie these blocks together for building and deploying software defined radios.

Using GNU Radio, a radio can be built by creating a graph where the vertices are signal processing blocks and the edges represent the data flow between them. The GNU Radio components are connected as shown in Figure 3.1. The signal processing blocks are implemented in C++ and the graphs are constructed and run in Python. Conceptually, a signal processing block processes an infinite stream of data flowing from its input ports to its output ports. A block's attributes include the number of input and output ports it has as well as the type of data that flows through each. Some blocks have only output ports or input ports. Input and output ports serve as data sources and sinks in the graph. For instance, there are sources that read from a file or ADC, and sinks that write to a file, digital-to-analog converter (DAC) or graphical display. More than 100 blocks are currently implemented in GNU Radio. Using a generic RF front end and few other hardware components like the ADC and DAC, GNU Radio code implements radio functionalities as shown in Figure 3.2.
Figure 3.1  Block diagram of GNU Radio components

Figure 3.2  Transmit and receive paths
3.2 GNU Radio: A “Hello World” Example Application

“Hello World” example application in GNU Radio would be the generation of the US dial tone and playing it using a PC’s audio device (Table 3.1). The dial tone is generated by two sine waves, one on the left channel and the other on the right channel of the audio device.

Table 3.1 Dial tone Python code

```python
#!/usr/bin/env python
from gnuradio import gr
from gnuradio import audio
def build_graph():
    sampling_freq = 48000
    ampl = 0.1
    fg = gr.flow_graph()
    src0 = gr.sig_source_f(sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f(sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink(sampling_freq)
    fg.connect((src0, 0), (dst, 0))
    fg.connect((src1, 0), (dst, 1))
    return fg

if __name__ == '__main__':
    fg = build_graph()
    fg.start()
    raw_input('Press Enter to quit: ')
    fg.stop()
```
3.2.1 Code Walkthrough

A flow graph is created to hold the blocks and connections between them. The two sine waves are generated by the `gr.sig_source_f` function calls. The `f` suffix indicates that the source produces floating numbers. One sine wave is at 350 Hz, and the other is at 440 Hz. Together, they sound like the US dial tone. The audio sink takes one or more streams of floats in the range of -1 to +1 as its input, and writes its input to the sound card. The three blocks are connected together using the `connect()` method of the flow graph. The `connect()` method takes two parameters, the source endpoint and the destination endpoint, and creates a connection from the source to the destination. An endpoint has two components: a signal processing block and a port number. The port number specifies which input or output port of the specified block is to be connected. In the general form, an endpoint is represented as a python tuple like this: `(block, port_number)`. When `port_number` is zero, the block may be used alone. For instance the following two expressions are equivalent:

```python
fg.connect ((src1, 0), (dst, 1))

fg.connect (src1, (dst, 1))
```

The `start()` method forks one or more threads to run the computation described by the graph and returns control immediately to the caller. In this case, the program waits for any keystroke.

The graphical user interfaces (GUI) for GNU Radio applications, such as the soft oscillograph and the soft spectrum analyzer are built using wxPython.
3.2.2 Summary

1. Contains a library of signal processing blocks written in C++
2. The processing blocks include signal sources, signal sinks, and filters.
3. The processing blocks are glued together using Python
4. A flow graph, with vertices as signal processing blocks and edges as data flow between vertices, defines the modeled system.
5. The processing block attributes define the number of input & output ports and type of data flowing through them

3.3 USRP (Universal Software Radio Peripheral)

The Universal Software Radio Peripheral, (USRP) is a device which allows the creation of a software defined radio using any computer with an USB 2.0 port. Various plug-on daughterboards allow the USRP to be used on different radio frequency bands. Presently, daughterboards operating from DC (logical zero) to 2.9 GHz are available. The entire schematics design of the USRP is open source. A typical setup of the USRP board consists of one mother board and up to four daughter boards, as shown in Figure 3.3.
The USRP consists of a motherboard containing up to four 12-bit, 64M sample/sec ADCs, four 14-bit, 128M sample/sec DACs, a million gate, Field Programmable Gate Array (FPGA) and a programmable USB 2.0 controller. Each fully populated USRP motherboard supports four daughterboards, two for receiving and two for transmitting. RF front ends are implemented on the daughterboards.
One USRP can simultaneously receive and transmit on two antennas in real time. All sampling clocks and local oscillators are fully coherent, thus allowing the creation of MIMO (multiple input, multiple output) systems. In the USRP, high sampling rate processing takes place in the FPGA, while lower sampling rate processing occurs in the host computer. The two onboard digital downconverters (DDCs) mix, filter, and decimate (from 64 M Samples/s) incoming signals in the FPGA. Two digital upconverters (DUCs) interpolate baseband signals to 128 MS/s before translating them to the selected output frequency. The DDCs and DUCs combined with the high sampling rates also greatly simplify analog filtering requirements. Daughterboards mounted on the USRP provide flexible, fully integrated RF front-ends. The USRP accommodates up to two RF transceiver daughterboards (or two transmit and two receive) for RF I/O.

### 3.3.1 USRP Features

USRP includes the following features.

1. Four 64 MS/s 12-bit analog to digital Converters and four 128 MS/s 14-bit digital to analog Converters

2. Four digital downconverters with programmable decimation rates and two digital upconverters with programmable interpolation rates

3. High-speed USB 2.0 interface (480 Mb/s) and capable of processing signals up to 16 MHz wide

4. Modular architecture supports wide variety of RF daughterboards

5. Auxiliary analog and digital I/O support complex radio controls such as RSSI and AGC

6. Fully coherent multi-channel systems (MIMO capable)
3.3.2 A/D and D/A Converters

The 4 high-speed 12-bit AD converters can sample at a rate of 64M samples per second. In theory, it could digitize a band as wide as 32MHz. The AD converters can bandpass-sample signals of up to about 150MHz. If an IF signal larger than 32MHz is sampled, it introduces aliasing. The higher the frequency of the sampled signal, the more the SNR degrades by jitter. The recommended upper limit is 100MHz. The full range on the ADCs is 2V peak-to-peak, and the input is 50 ohms differential, i.e. 40mW, or 16dBm. A programmable gain amplifier (PGA) is used before the ADCs to amplify the input signal, and to utilize the entire input range of the ADCs, in case the signal is weak. The PGA range is up to 20dB. Other sampling rates can be used if needed. The available rates are all submultiples of 128, such as 64, 32 etc.

At the transmitting path, there are 4 high-speed 14-bit DA converters. The DAC clock frequency is 128 MS/s, so Nyquist frequency is 64MHz. However, staying below about 50MHz or so will make filtering easier. Therefore, a useful output frequency range is DC to about 50MHz. The DACs can supply an output of 1V peak to a 50 ohm differential load, or 10mW (10dBm). There is also a PGA used after the DAC, providing up to 20dB gain. The PGAs on both RX and TX paths are programmable.

In principle, the 4 input and 4 output channels use real samplings. However, there will be more flexibility (and bandwidth) if a complex (IQ) sampling is used. For instance, the input and output channels can be paired up, resulting in 2 complex inputs and 2 complex outputs.
3.3.3 Daughter Boards

There are four slots on the motherboard, which can be used to plug in up to 2 RX daughter boards and 2 TX daughter boards. The daughter boards are used to hold the RF receiver interface and the RF transmitter.

There are slots for 2 TX daughter boards, labeled TXA and TXB, and 2 corresponding RX daughter boards, RXA and RXB. Each daughter board slot has access to 2 of the 4 high-speed AD/DA converters (DAC outputs for TX, ADC inputs for RX). This allows each daughter board which uses real (not IQ) sampling to have 2 independent RF sections and 2 antennas (4 total for the system). If complex IQ
sampling is used, each board can support a single RF section, for a total of 2 for the whole system. There are also two SMA connectors on each daughter board. Normally they are used to connect the input or output signals.

There exist several kinds of daughter boards available now:

1. Basic daughter boards. A basic board with nothing fancy on it. Two SMA connectors are used to connect external tuners or signal generators. This can be treated as an entrance or an exit for the signal without affecting it. Some form of external RF front end is required

2. DBSRX daughter boards. This is a receive-only board with RF frequency ranging from 800MHz to 2.4GHz.

3. Transceiver daughter boards, where several transceiver boards are available.
   
   RFX400   -- 400-500 MHz Transceiver, 100+mW output
   RFX900   -- 800-1000MHz Transceiver, 200+mW output
   RFX1200 -- 1150 MHz - 1450 MHz Transceiver, 200+mW output
   RFX1800 -- 1.5-2.1 GHz Transceiver, 100+mW output
   RFX2400 -- 2.3-2.9 GHz Transceiver, 20+mW output

3.3.4 FPGA

Understanding the FPGA operation is one of the important steps for GNU Radio users. As shown in the figure above, all the ADCs and DACs are connected to the FPGA. The FPGA plays a key role in the GNU Radio system. Basically what the FPGA does is to perform high bandwidth math, and to reduce the data rates to something that can be easily transferred over a USB 2.0 interface. The FPGA connects to an USB 2.0 interface chip, the Cypress FX2. Everything (FPGA circuitry and USB 2.0 Microcontroller) is programmable over the USB 2.0 bus.

The standard FPGA configuration includes digital down converters (DDC) implemented with cascaded integrator-comb (CIC) filters. CIC filters are very
high-performance filters using only adds and delays. The FPGA implements 4 digital down converters (DDC), which allows 1, 2 or 4 separate RX channels. At the RX path, there are 4 ADCs and 4 DDCs. Each DDC has two inputs I and Q. Each of the 4 ADCs can be routed to either the I or the Q input of any of the 4 DDCs, which allows for having multiple channels selected out of the same ADC sample stream.

The digital up converters (DUCs) on the transmit side are actually contained in the AD9862 CODEC chips, not in the FPGA. The only transmit signal processing blocks in the FPGA are the interpolators. The interpolator outputs can be routed to any of the 4 CODEC inputs.

The multiple RX channels (1, 2, or 4) must all be the same data rate (i.e. same decimation ratio). The same applies to the TX channels, where each channel must be at the same data rate, which may be different from the RX rate.

The MUX is like a router or a circuit switch. It determines which ADC (or constant zero) is connected to each DDC input. As shown in Figure 3.5, there are 4 DDCs, each having two inputs. The MUX is controlled using usrp.set_mux() method in Python.
Figure 3.5  MUX block diagram showing the USRP using a Basic Rx and TvRx

MUX Values

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Q</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>I</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Q</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>I</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Q</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Each 4-bit I field is either 0, 1, 2, or 3.
Each 4-bit Q field is either 0, 1, 2, 3 or 0xf (input is const zero).
All Q's must be 0xf or none of them may be 0xf.

Each input specified with (I0, Q0, I1 ... I3, Q3), i.e denoting which ADC is connected to it by using 4 bits (0, 1, 2, 3 or 0xf). Therefore a 32-bit integer would be enough for all 8 inputs to know which ADC is connected. Of course an integer in hexadecimal system will be more convenient if we want to use the set_mux() method. For most real sampling applications, the Q input of each DDC is constant zero. So quite often the modification of standard configuration of the FPGA is not required.
The digital down converter (DDC), first, down converts the signal from the IF band to the base band. Second, it decimates the signal so that the data rate can be adapted by the USB 2.0 and is reasonable for the computers' computing capability. Figure 3.6 shows the block diagram of the DDC. The complex input signal (IF) is multiplied by the constant frequency (usually also IF) exponential signal. The resulting signal is also complex and centered at 0. Then the signal is decimated with a factor of N.

The decimator can be treated as a low pass filter followed by a downsampler. Consider the decimation factor to be D. The digital spectrum, the low
pass filter selects the band \([-p/D, p/D]\), and the downsampler spreads the spectrum in 
\([-p/D, p/D]\) to \([-p, p]\). Thus, the bandwidth of the digital signal of interest is narrowed
down by a factor of \(D\). The USB 2.0 interface can sustain about 32MB/sec across it.
All samples sent over the USB 2.0 interface are in 16-bit signed integers in IQ format,
i.e. 16-bit I and 16-bit Q data (complex), resulting in 8M complex samples/sec across
the USB 2.0. This provides a maximum effective total spectral bandwidth of about
8MHz by Nyquist criteria. Of course a much narrower range can be selected by
changing the decimation rate. For example, the bandwidth of a FM station is generally
200kHz. Therefore the decimation factor can be set to 250, then the data rate across
the USB 2.0 is \(64MHz / 250 = 256kHz\), which is well suited for the 200kHz
bandwidth without losing any spectral information. The IF frequency of the DDC can
be set using usrp.set_rx_freq() method and the decimation factor can be set using the
usrp.set_decim_rate() method in Python. The decimation rate must be in the range \([1, 256]\).

Note that when there are multiple channels (up to 4), the channels are
interleaved. For example, with 4 channels, the sequence sent over the USB 2.0
interface would be I0 Q0 I1 Q1 I2 Q2 I3 Q3 I0 Q0 I1 Q1, etc. The USRP can operate
in full duplex mode. When in duplex mode, the transmit and the receive sides are
completely independent of one another. The only consideration is that the combined
data rate over the USB 2.0 bus must be 32MBps or less.

Finally when the I/Q complex signal enters the computer via the USB 2.0
interface, that's where the software takes over the processing

At the TX path, the story is pretty much the same, except that it happens
in inverse. A baseband I/Q complex signal needs to be sent to the USRP board from
the computer. The digital up converter (DUC) interpolates the signal, up converts it to
the IF band, and finally sends it through the DAC.

3.4 GNU Radio Installation

The list of available packages includes:

1. [gnuradio-core] The main library, contains the underlying runtime system and most of the hardware independent signal processing blocks.

2. [gnuradio-examples] Simple examples that exercise GNU Radio.

3. [gr-audio-alsa] Support for sound cards using the ALSA (Preferred on GNU/Linux).

4. [gr-audio-jack] Support for sound cards using the JACK (experimental: GNU/Linux).

5. [gr-audio-oss] Support for sound cards using the Open Sound System.


7. [gr-audio-osx] Support for OSX audio

8. [gr-audio-windows] Support for audio-sink in windows.

9. [gr-wxgui] GUI framework built on wxPython, including blocks for displaying realtime FFT and oscilloscope.

10. [gr-gsm-fr-vocoder] GSM 06.10 13kbit/sec voice encoder/decoder.

11. [gr-radio-astronomy] Radio astronomy application files

12. [gr-trellis] Implementation of trellis-based encoding and decoding algorithms

14. [usrp] The non-GNU Radio specific part of the USRP code base, containing the host libs, firmware and FPGA code.

15. [gr-usrp] The glue that ties the usrp library into GNU Radio.

### 3.4.1 Building GNU Radio

External dependencies are listed below:

With the exception of SDCC, the following GNU/Linux distributions are known to come with all required dependencies pre-packaged: Ubuntu 6.06, SuSE 10.0 (the pay version, not the free download), Fedora Core 2, 3, 4, 5, and 6. Other distributions may work too. The required packages may be contained on your installation CD/DVD, or may be loaded over the Internet. The specifics vary depending on particular GNU/Linux distributions.

On systems using pkgsrc (e.g. NetBSD/Dragonfly), “build metapackages/gnuradio”, will build a previous release and force installation of the dependencies. Then pkg_delete the gnuradio and usrp packages, which will leave the dependencies.

See the wiki at http://gnuradio.org/trac/wiki for more details.

Installing from the SVN repository:

The new repository organization simplifies a lot of the build system. You no longer need to go into the individual directories and compile separately.

To checkout the latest code from the development trunk, enter this on the command line:

$ svn co http://gnuradio.org/svn/gnuradio/trunk gnuradio
To instead checkout the latest stable release code, enter this on the command line:

```
$ svn co http://gnuradio.org/svn/gnuradio/branches/releases/3.0 gnuradio
```

after ensuring that the dependencies specified in the top-level README are met. Most GNU/Linux systems come packaged with the earlier listed dependencies. You may need to install them off your install CD/DVD or over the Internet.

To compile, there are 5 steps. Start by cd'ing to the gnuradio directory, then complete the following commands:

```
$ ./bootstrap         # Do NOT perform this step if you are building from a tarball.
$ ./configure
$ make
$ make check
$ sudo make install
```

This will perform all configuration checks and select for build, test, and installation all components that pass.

Brief instructions for installing the tarball packages in Fedora Core:

**[gnuradio-core]:**

Prerequisites:

1. The "autotools"
   - autoconf 2.57 or later
   - automake 1.7.4 or later
   - libtool 1.5 or later
If your system has automake-1.4, there's a good chance it also has automake-1.7 or later. Check your install disk and/or (on GNU/Linux) try:

```
$ man update-alternatives
```

for info on how some distributions support multiple versions.

2. **pkgconfig 0.15.0 or later**
   http://www.freedesktop.org/Software/pkgconfig
   From the web site: pkgconfig is a system for managing library compile/link flags that works with automake and autoconf. It replaces the ubiquitous *-config scripts you may have seen with a single tool.

3. **FFTW 3.0 or later (http://www.fftw.org)**
   When building FFTW, you must use the --enable-single and --enable-shared configure options. This builds the single precision floating point version which we use. You should also use either the --enable-3dnow or --enable-sse options if you're on an Athlon or Pentium respectively.

4. **Python 2.3 or later**  http://www.python.org
   Python 2.3 or later is now required. If your distribution splits python into a bunch of separate RPMS including python-devel or libpython, you'll most likely need those too.

   ```
   ./configure --enable-unicode=ucs4
   make clean
   make
   make install
   ```

5. **Numeric python library**  http://numeric.scipy.org
   This library provides a high performance array type for Python.
   http://sourceforge.net/project/showfiles.php?group_id=1369&package_id=1351

   GNU Radio uses the Smart Pointer library. Fedora Core 2 has a package for this library, boost-devel-1.31.0-7. Otherwise download the source and follow the build instructions, which are a bit different from the normal ./configure && make

7. **cppunit 1.9.14 or later.** (http://cppunit.sourceforge.net)
   Unit testing framework for C++.

Some of the other utilities which may be required are:

8. **SWIG – Simplified Wrapper and Interface Generator**
   http://www.swig.org/
These versions are known to work: 1.3.23, 1.3.24, 1.3.25, 1.3.27, 1.3.28, 1.3.29

9. SDCC – Small Device C Compiler
   http://sdcc.sourceforge.net/
   Use version 2.4.0 or later.
   This includes a C compiler and linker for the 8051. It's required to build
   the firmware for the USRP. If you don't have a USRP, ignore this.

Optional, but nice to have:

    Use version 2.5.2.7 or later. Again, almost all systems have this available.
    As a last resort, build it from source, although it is not recommended.
    http://www.wxpython.org

    Also, as noted on the website, GNU Radio experiences bugs in certain
    versions of g++ 3.3.x on the x86 platform. If you are using g++ 3.3 and make check
    fails, please either upgrade to 3.4 or downgrade to 3.2. Both are known to work.

[gnuradio-examples]

    Set your PYTHONPATH environment variable so that the GNU Radio
    toolkit and optional packages can be found by python. PYTHONPATH should include
    the path of the local site-packages directory.

    If the above packages were installed using the default prefix (/usr/local)
    and you are using python 2.3, this should work:

    $ export PYTHONPATH=/usr/local/lib/python2.4/site-packages

    You may want to add this to your ~/.bash_profile or similar file.
Once PYTHONPATH is set, you should be able to run any of the examples for which you have the required i/o devices.

To ensure that your setup is sane, try the following command:

$ python

>>> from gnuradio import gr

If this works, your PYTHONPATH is set correctly.

**[gr-audio-alsa] & [gr-audio-oss]**

These two packages are audio packages which are needed to interface with the audio device on your computer.

**[gr-howto-write-a-block]:**

The documentation with descriptions of how to write signal processing blocks for GNU Radio

If you have gotten doxygen installed and provided the --enable-doxygen configure option, the build process creates documentation for the class hierarchy, etc. Point your browser at gnuradio-core/doc/html/index.html

The online version can be found at: [http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html](http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html). The online documentation for GNU Radio with descriptions of all of the modules, class hierarchy and file list can be found at: [http://www.gnu.org/software/gnuradio/doc/](http://www.gnu.org/software/gnuradio/doc/)

43
[gr-wxgui]:

GUI framework built on wxPython, includes blocks for displaying realtime FFT and oscilloscope. The modules wxPython and Numerical Python also need to be installed.

http://wiki.wxpython.org/index.cgi/Getting_Started is a good place to look for information about wxPython and its installation guidelines.

A short summary of the instructions is as below:

a. Install Python
(http://www.python.org/download/releases/2.4.4/) (This is the version I used

b. Install wxPython: (requires glib and gtk+ libraries installed -
http://www.wxpython.org/download.php#prerequisites)

Download the source code of the last wxPython release: wxPython website. The website has listings for different platforms. There is no separate version for Fedora Core 3, but instead use the version for Fedora Core 2 and it works.

The default installation happens in “/usr/lib/python<version>/site-packages/”. But the Python installation happens in “/usr/local/lib/python<version>/site-packages/”. This may prevent the “wx” module from being accessed correctly.

To solve the wx module access problem, in “/usr/local/lib/python<version>/site-packages/” create the path file “wx.pth” and in that file give the full path to the wxPython installation i.e
“/usr/lib/python<version>/site-packages/wx-<unicode-version>/”. This will enable python to find the wx module.

c. To test the installation with a small program in python:

```python
import wx
app = wx.PySimpleApp()
frame = wx.Frame(None, -1, "Hello World")
frame.Show(1)
app.MainLoop()
```

Here is what you should get with wx:

![Hello World Frame](image)

After importing wxPython GUI, we instantiate a new wxPySimpleApp and a new wxFrame. A frame in wxPython is a window with its titlebar, reduction and close buttons, etc... We make this Frame appear by "showing" it. Eventually, we start the application's MainLoop whose role is to handle the events.

d. Install Numerical Python: Numerical Python adds a fast array facility to the Python language (http://numeric.scipy.org/).
Download it from http://sourceforge.net/projects/numpy. The install happens within the python installation directory, hence requires no changes to the path file.

To verify, look under “/usr/local/lib/python<version>/site-packages/” and ensure that the files “Numeric.pth” and “wx.pth” exist and contain valid paths.


e. Install NumArray:
   http://www.stsci.edu/resources/software_hardware/numarray
   Download it from Sourceforge Numarray Download Page.

[usrp]:

The non-GNU Radio specific part of the USRP code base, which contains the host libs, firmware and FPGA code.

[gr-usrp]:

The glue that ties the usrp library into GNU Radio. The USRP hardware needs to be setup and checked for correct operation.

http://comsec.com/wiki?UsrpInstall gives a quick walkthrough to check the working of USRP board.

USRP hardware schematics and associated files can be obtained with the following command:

$ svn co http://gnuradio.org/svn/usrp-hw/trunk usrp-hw
3.4.2 Running Examples

To run examples, “cd” into the directory “gnuradio-examples/python/audio”.

You should find some *.py files which can be run as executables. Running “./dialtone.py”, should produce a dial tone of frequency 32KHz.

Also in the "usrp" subdirectory. Running ./usrp_oscope.py, should bring up something that looks like an oscilloscope. Grab the corner of the window and resize it so you can read the labels on the buttons. It looks pretty dull until you get the triggering working -- set it from "Pos" to "Auto". Then you'll start seeing a bunch of noise on the screen -- a red line and a green line. If you attach a piece of wire to the inner conductor of your "RX-A" input on your "RXA" daughterboard, the green line will start to wiggle a lot.

A more interesting example is testing the FM reception. Even without a tuner, the ADC's on the USRP should be able to digitize a narrowband signal in the range up to about 200MHz, by adjusting the built-in digital downconverter (DDC). Find out a FM station with strong broadcast signal in your area, and try to receive it. For instance,

```
./usrp_wfm_rcv.py --freq 104.5
```

The argument is the station's center frequency, in megahertz. A window will pop up which will show the signal at various stages of processing; and the radio station should be audible on your computer's speakers.
3.4.3 Digital Communication Examples (/gnuradio-examples/python/digital)

GNU Radio uses the universal TUN/TAP drivers to tunnel the packets from the USRP via USB 2.0 to the Linux kernel.

3.4.3.1 TUN/TAP Adapters

TUN/TAP provides packet reception and transmission for user space processes. It can be seen as a simple Point-to-Point or Ethernet device, which, instead of receiving packets from physical media, receives the packets from user space process and instead of sending packets via physical media writes the packets to the user space process.

In order to use the driver, a program has to open /dev/net/tun and issue a corresponding ioctl() to register a network device with the kernel. A network device will appear as tunXX or tapXX, depending on the options chosen. When the program closes the file descriptor, the network device and all corresponding routes will disappear. Depending on the type of device chosen, the userspace program has to read/write IP packets (with tun) or Ethernet frames (with tap). Which one being used depends on the flags given to the ioctl(). The package from http://vtun.sourceforge.net/tun contains two simple examples on how to use tun and tap devices. Both programs work like a bridge between two network interfaces. br_select.c - bridge based on select system call. br_sigio.c - bridge based on async IO and SIGIO signal. The best example is VTun (http://vtun.sourceforge.net).

3.4.3.2 Universal TUN/TAP Device Driver FAQs

The following FAQs are adapted from http://vtun.sourceforge.net/tun/faq.html.
1. What platforms are supported by TUN/TAP driver?

Currently driver has been written for 3 Unices:

- Linux kernels 2.2.x, 2.4.x
- FreeBSD 3.x, 4.x, 5.x
- Solaris 2.6, 7.0, 8.0

2. What is TUN/TAP driver used for?

As mentioned above, the main purpose of TUN/TAP driver is tunneling, which is used by VTun (http://vtun.sourceforge.net). Another interesting application using TUN/TAP is pipsecd (http://perso.enst.fr/~beyssac/pipsec/), an userspace IPSec implementation that can use complete kernel routing (unlike FreeS/WAN).

3. How does Virtual network device actually work?

Virtual network device can be viewed as a simple Point-to-Point or Ethernet device, which instead of receiving packets from a physical media, receives the packets from user space program and instead of sending packets via physical media sends the packets to the user space program. Let's say that you configured IPX on the tap0, then whenever the kernel sends an IPX packet to tap0, it is passed to the application (VTun, for example). The application encrypts, compresses and sends it to the other side over TCP or UDP. The application on the other side decompresses and decrypts the data received and writes the packet to the TAP device, where the kernel handles the packet like it came from real physical device.

4. What is the difference between TUN driver and TAP driver?
TUN works with IP frames, and TAP works with Ethernet frames. This means that you have to read/write IP packets when you are using tun and Ethernet frames when using tap.

5. What is the difference between BPF and TUN/TAP driver?

BFP is an advanced packet filter, which can be attached to existing network interface. BPF does not provide a virtual network interface. A TUN/TAP driver does provide a virtual network interface and it is possible to attach BPF to this interface.

6. Does TAP driver support kernel Ethernet bridging?

Yes. Linux and FreeBSD drivers support Ethernet bridging.

3.4.3.3. Communication example using TUN/TAP (tunnel.py)

This program provides a framework for building your own MAC protocols. It creates a "TAP" interface in the kernel, typically gr0, and sends and receives Ethernet frames through it. The Linux 2.6 kernel includes the tun module, so you don't have to build it. You may have to "modprobe tun" if it is not loaded by default. If /dev/net/tun does not exist, try "modprobe tun".

To run this program you'll need to be root or running with the appropriate capability to open the tun interface. You'll need to fire up two copies on different machines. Once each copy is running you'll need to ifconfig the gr0 interface to set the IP address.

This will allow two machines to talk to eachother, but anything beyond the two machines depends on your networking setup. As an example,
On machine A:

$ su

# ./tunnel.py --freq 423.0M --bitrate 500k

# # in another window on A, also as root...

# ifconfig gr0 192.168.200.1

On machine B:

$ su

# ./tunnel.py --freq 423.0M --bitrate 500k

# # in another window on B, also as root...

# ifconfig gr0 192.168.200.2

Now, on machine A you should be able to ping machine B:

$ ping 192.168.200.2

and you should see some output for each packet in the tunnel.py window if you used the -v option.

Likewise, on machine B:

$ ping 192.168.200.1

This example now uses a carrier sense MAC protocols, so it is now possible to ssh between the machines, browse the web, etc.
Chapter 4
GNU RADIO COMPANION

4.1 Introduction

GNU Radio programming has a steep learning curve, due to its command line interface. To aid beginners, Josh Blum of Johns Hopkins University, has developed a graphical interface for GNU Radio. This GUI termed GNU Radio Companion (GRC), allows users to interact with GNU Radio signal blocks in a manner similar to Labview or Simulink. The entire interface is completely designed with GNU Radio in mind, and encompasses over 150 blocks from the GNU Radio Project. Blocks are manually integrated into GRC via descriptive python definitions. The definitions are very flexible, and allow multiple GNU Radio blocks to be grouped into a single GRC super-block.

4.2 Components

1. **Flow Graph:** A flow graph is an interconnection of signal processing blocks. GRC provides a scrollable window to place and connect various signal blocks

2. **Signal Blocks:** Signal blocks perform all of the processing in a flow graph. For example: A signal block can be a filter, an adder, a source, or a sink. GRC represents signal blocks as rectangular blocks. Each block has a label indicating the name of the block and a list of parameters.

3. **Parameters:** Parameters influence the function of a signal block. For example, a parameter can be a sampling rate, a gain,
or a flag. Most parameters for a signal block are displayed below its label.

4. Sockets: Sockets are the inputs and outputs of a signal block. Each signal block has certain sockets associated with it. For example, an adder has two input sockets and one output socket. GRC represents a socket as a small rectangle attached to the signal block. The socket has a label indicating its function. Labels are usually named "in" or "out". Some labels are named "vin" or "vout" to indicate a vector type. Sockets are also colored to indicate their data type. Blue for complex, Red for float, Green for int, Yellow for short, and Purple for byte.

5. Connections: A connection joins an input and an output socket. GRC represents connections by drawing a line between the two sockets. Connections must be between matching data types, including vectors.

6. Variables: A variable holds a number that is available to all elements in the flow graph. Variables serve two purposes: First, parameters can use a variable as a way to share values. For example, if all parameters for sampling rate use the samp_rate variable, changing the samp_rate variable once is easier than modifying every parameter. Second, variables can also have a range (min and max) associated with them. Variables with ranges can be dynamically changed while the flow graph is running.

4.3 Installation

4.3.1 Dependencies

Any version of GNU Radio should be fine. If a block does not exist in a particular version, GRC will ignore the missing block. Installing GNU Radio will take care of most of GRC's dependencies. GRC needs wx support to run flow graphs, and gtk2 for its main GUI. The python XML module is essential, as all flow graphs and user preferences are saved to and loaded from xml. The package names for pyxml vary among software distributions.
In summary, a list of the required packages include:

1. gnuradio with gr-wxgui
2. python-gtk2 (>=2.6)
3. python-xml and/or pyxml

The software can be downloaded from http://www.joshknows.com/download/grc/. More information regarding updates to the GRC projects can be found at http://www.joshknows.com/?key=grc.

4.4 Using GRC

4.4.1 Running the interface

In the src folder, execute Run.py with the python interpreter. Saved flow graphs can be passed as the first argument to Run.py.

4.4.2 Adding a block

Select a block from the signal block tree menu. Double click the block or click the add button. A new signal block will be placed on the screen.

4.4.3 Moving a block

Left click a block and drag it around the flow graph. If you drag a block near a border that can be scrolled, wiggle the block to advance the scroll bar.

4.4.4 Rotating a block

With a block selected, select rotate right or left from the tool bar. Short-cut keys: right or left
4.4.5 Deleting a block

With a block being selected, select delete from the tool bar. Short-cut keys: delete

4.4.6 Connecting blocks

Left click on one socket and then left click on another socket. A connection will only be created between an input socket and an output socket, and the input socket may have no existing connections. An output socket may have unlimited connections. Red lines surrounding the connection indicate that the data types of the sockets do not match.

4.4.7 Modifying Parameters

Double click on a block or select it and choose properties from the edit menu. A dialog containing all the parameters for the signal block will appear. Some parameters are set via a drop down menu, and most must be typed in as characters. These parameters are usually numeric, representing sampling rates, gains, and amplitudes. Numeric parameters may contain mathematical expressions with variables.

4.4.8 Numeric Expressions

A numeric expression may contain any number of variables, numbers, and operators. There must be an operator between every pair of numbers/variables. Possible operators are + - * / ^. Numbers can be integers, decimals, floating-point, and
complex. Python's built in floating point and imaginary formats are used. Floating point numbers end in an e followed by a signed integer. Imaginary numbers end in a j. Matching bracket pairs are allowed: (), [], {}. Variables are denoted by a leading '$' character.

4.4.8.1 Valid Expressions

Following are sample valid expressions:

1. 32e3 + $offset
2. 4*(2.3^1j + 2)
3. 32e3j + $num^-4
4. ([32e3 - 1.5e-4] * 0.5j)^3e-1j

4.4.8.2 Using Variables

The variable window has 4 columns: variable name, default value, minimum value, and maximum value. Choose ‘add’ to create a new variable entry. Choose ‘remove’ to delete a selected variable entry. Any value in an entry can be edited by clicking on the input field in the selected row. Invalid changes are ignored. Variable names are case-sensitive and alpha-numeric, and they may contain hyphens. The first character must be a letter. The default value for a variable may be any string.

4.4.8.3 Using Variables with Ranges

A variable with a range may be dynamically changed while the flow graph is running. The default, minimum, maximum, and step size values must be of type float or int. To make a variable with a range, simply type a number into the min and/or the max input field and the other fields will be filled in automatically with default
values. The min must be less than or equal to the default, and the max must be greater than or equal to the default. To remove the range from a variable, just clear one of the input cells for min, max, or step and all cells will be cleared.

4.4.9 Flow Graph Validation

1. Connections are between input and output sockets of the same data type. Different data types have different colors. Therefore, input and output colors must match. Invalid connections are highlighted red.

2. All sockets must be connected (except for the optional sockets). Disconnected sockets cause their signal block to have a red label.

3. All signal block parameters must be valid. For instance, numerical expressions can be parsed. Invalid parameters have red labels and cause their signal block to have a red label.

4.4.10 Running a Flow Graph

If a flow graph is valid, all parameters are valid and all sockets are connected. Choose run from the tool bar or press F5. A wx window will appear with any sliders or graphs that were added. To stop the flow graph, close the wx window, press stop in the tool bar, or press F7. Flow graphs can be run without the interface by invoking the python interpreter on FlowGraphApp.py, and passing the saved flow graph as the first argument to FlowGraphApp.py.

4.4.11 Options

The options menu allows a user to adjust the window size, set user preferences, and view USRP information.

1. The window size dialog allows the user to set the size of the flow graph in pixels.
2. The user preferences dialog stores per-user preferences. These preferences can alter the flow graph validation, or set a default flow graph when the program starts.

3. The USRP diagnostics dialog queries the USRP and selected daughterboard for information.

4.5 GRC Snapshots

The usability of the GRC interface is illustrated in the following two sections. The first section illustrates filtering a noisy channel and the second models a QAM Modulator/Demodulator system.

4.5.1 Filtering A Noisy Channel

The noisy channel is generated using a cosine signal source and a Gaussian noise source. This generated noisy channel is connected to two FFT sinks, (which act as oscilloscopes to observe the noisy channel) one with a band pass filter (BPF) and the other without a filter.
Figure 4.1  Channel Noise Generation

Each block’s parameters can be modified either by double clicking on the block or by selecting the block and choosing the ‘Params’ option from the ‘Edit’ menu. The BPF blocks parameters are set in Figure 4.2..
Figure 4.2 Channel Noise Parameters
Since the FFT sinks are oscilloscopes, when the created flow graph is initiated, the noisy channel signal waveforms can be observed on the sinks. The outputs are shown in Figure 4.3. It can be seen that the FFT-BPF sink has a more defined source signal component around the 5 kHz frequency due to the presence of the BPF which eliminates the noise components.
4.5.2 QAM Modulation / Demodulation

Similar to the simple noisy channel model above, a complete modulator/demodulator system also can be designed with the same ease. A QAM modulator/demodulator system is illustrated in Figure 4.4.
Chapter 5
MAC PROTOCOL IMPLEMENTATION FRAMEWORK

5.1 Motivation

The features of GNU Radio allow it to be used as a prototyping environment to prototype new MAC protocols, which helps in evaluating the designed protocols in real protocol stacks and bridge the gap between simulation and emulation environments. Although the existing hardware (i.e., contemporary processors and RF front end) is not yet capable of accurately emulating the MAC/PHY protocol functionalities, it provides for very useful exercises in implementing them in GNU Radio.

5.2 Using QualNet Code

The ability of implementing a function as a processing block allows for easy inclusion of new functionality into GNU Radio. One idea is to use existing, developed simulation code for the protocol from QualNet and use wrapper functions to make them as processing blocks and to integrate them with GNU Radio as shown in Figure 5.1. Such wrapper functions serve as plug-in points for future protocols to be made part of GNU Radio.

The steps involved in integrating existing simulation code of a desired MAC protocol with GNU Radio are:

1. Strip out the functional pieces from the QualNet code.
2. Write a wrapper function to handle the interfacing between QualNet function calls and GNU Radio function calls.

3. Include the available *.c and *.h files from QualNet into the Makefile in GNU Radio source directories.

4. The required QualNet object files should be included into the libgnuradio.so file using the ‘libtool’ package along with other GNU Radio functions.

5. Recompile GNU Radio using the modified Makefile

Figure 5.1  GNU Radio MAC protocol framework using wrapper functions
5.3 Drawing Parallels

Certain QualNet functionalities need to be emulated in GNU Radio to implement the MAC protocol. Examples of corresponding functions in QualNet and GNU Radio include:

1. NetworkLayerHasPacketToSend $\leftrightarrow$ os.read()
2. ReceivePacketFromPhy $\leftrightarrow$ phy_rx_callback()
3. ReceivePhyStatusChangeNotification $\leftrightarrow$ carrier_sensed()
4. MessageSend() $\leftrightarrow$ send_pkt()
5. Message_Alloc $\leftrightarrow$ make_pkt()

These functions can be modified or completely rewritten to suit the protocol being implemented.

To emulate the scheduling of MAC protocol events, GNU Radio supports real time scheduling, if available in the kernel. For instance, the Linux kernel supports real time scheduling, which can be accessed using the “sched_setscheduler” command that sets both the scheduling policy and the associated parameters for the process identified by pid. Table 5.1 gives an excerpt of the communication example (i.e. tunnel.py) where real time scheduling is enabled if available and also the underlying C++ function, which implements the real time scheduling functionality.

The Linux scheduler offers three different scheduling policies, one for normal processes and two for real-time applications. A static priority value sched_priority is assigned to each process and this value can be changed only via system calls. Conceptually, the scheduler maintains a list of runnable processes for each possible sched_priority value, and sched_priority can have a value in the range of...
0 to 99. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. The scheduling policy determines for each process, where it will be inserted into the list of processes with equal static priority and how it will move inside this list.

**SCHED_OTHER** is the default universal time-sharing scheduler policy used by most processes, and **SCHED_FIFO** and **SCHED_RR** are intended for special time-critical applications that need precise control over the way in which runnable processes are selected for execution. Processes scheduled with **SCHED_OTHER** must be assigned the static priority 0, and processes scheduled under **SCHED_FIFO** or **SCHED_RR** can have a static priority in the range of 1 to 99.
### Table 5.1  Real-time scheduling support in tunnel.py

**Tunnel.py:**

```
........
r = gr.enable_realtime_scheduling()
if r == gr.RT_OK:
    realtime = True
else:
    realtime = False
    print "Note: failed to enable realtime scheduling"
........
```

The function which gets called:

```c
gr_enable_realtime_scheduling()
{
    int policy = SCHED_FIFO;
    int pri = (sched_get_priority_max (policy) - sched_get_priority_min (policy)) / 2;
    int pid = 0;  // this process

    struct sched_param param;
    memset(&param, 0, sizeof(param));
    param.sched_priority = pri;
    int result = sched_setscheduler(pid, policy, &param);
    if (result != 0){
        if (errno == EPERM)
            return RT_NO_PRIVS;
        else {
            perror ("sched_setscheduler: failed to set real time priority");
            return RT_OTHER_ERROR;
        }
    }
    //printf("SCHED_FIFO enabled with priority = %d\n", pri);
    return RT_OK;
}
```

//printf("SCHED_FIFO enabled with priority = %d\n", pri);
return RT_OK;
5.4 Packet Processing Blocks

5.4.1 Packet Received From PHY Layer

The PHY layer passes up the received packet to the MAC layer using the callback function. The phy_rx_callback function is used to process the incoming packets. The packet’s CRC is checked and then it is written to the TUN descriptor. Excerpt from the communication example (tunnel.py) defining the callback function is listed in Table 5.2. The received packet’s data is stored in the “payload” parameter and the result of CRC check is stored in the “ok” parameter, which is a boolean value.

Table 5.2 Packet receive code snippet

```python
def phy_rx_callback(self, ok, payload):
    """
    Invoked by thread associated with PHY to pass received packet up.

    @param ok: bool indicating whether payload CRC was OK
    @param payload: contents of the packet (string)
    """
    if self.verbose:
        print "Rx: ok = %r  len(payload) = %4d" % (ok, len(payload))
    if ok:
        os.write(self.tun_fd, payload)
```

5.4.2 Network Layer Has Packet To Send

The packet to be sent is stored in the variable “payload” and the channel is sensed before transmitting. When implementing CSMA, if the channel is busy, an
exponential backoff delay is used and then the process is repeated until the packet is transmitted using the send_pkt() function. Table 5.3 shows the CSMA functionality of the GNU Radio MAC layer along with the exponential backoff.

Table 5.3  Packet send code snippet

```python
def main_loop(self):
    
    Main loop for MAC.
    Only returns if we get an error reading from TUN.
    
    min_delay = 0.001  # seconds

    while 1:
        payload = os.read(self.tun_fd, 10*1024)
        if not payload:
            self.fg.send_pkt(eof=True)
            break

        if self.verbose:
            print "Tx: len(payload) = %4d" % (len(payload),)

        delay = min_delay
        while self.fg.carrier_sensed():
            sys.stderr.write('B')
            time.sleep(delay)
            if delay < 0.050:
                delay = delay * 2  # exponential back-off

        self.fg.send_pkt(payload)
```

The packet send code snippet demonstrates the use of exponential backoff in the MAC layer of the GNU Radio software.
5.4.3 Packet Format

The payload to be sent is packetized using the make_pkt() function. Given an access code & the payload, the make_pkt() function generates the packet, and, if required, adds sufficient padding such that each packet ends up being a multiple of 512 bytes when sent across the USB 2.0. Since, we send 4-byte samples across the USB 2.0 (16-bit I and 16-bit Q), we want to pad so that after modulation the resulting packet is a multiple of 128 samples. The parameters used to create the packet to send are:

* ptk_byte_len: length in bytes of packet, not including padding.
* samples_per_symbol: samples per bit (1 bit / symbol with GMSK)

The packet being sent out has the following format:

Access Code : Length : Payload : CRC

5.4.4 Packet Modulation and Demodulation

Packets to be sent are enqueued by calling send_pkt(). The output is a complex modulated signal at baseband. The modulation function requires the following parameters:

* fg: flow graph

* modulator: instance of modulator class (gr_block or hier_block)

* access_code: AKA sync vector (string of 1's and 0's between 1 and 64 long)

* msgq_limit: maximum number of messages in message queue

* pad_for_usrp: If true, packets are padded such that they end up a multiple of 128 samples
When demodulating a packet, the input is the complex modulated signal at baseband. Demodulated packets are sent to the rx_callback() handler. The required parameters for the demodulation function are:

- **fg**: flow graph
- **demodulator**: instance of demodulator class (gr_block or hier_block)
- **access_code**: AKA sync vector (string of 1's and 0's)
- **callback**: function of two args: ok, payload (ok: bool; payload: string)
- **threshold**: detect access_code with up to threshold bits wrong (-1 -> use default)

### 5.4 Related Work

#### 5.4.1 Emulab Based GNU Radio Testbed [15]

University of Utah and the GNU Radio team are developing and deploying a programmable, low-cost, high-performance, and open software radio platform to support innovative research and education projects in wireless networking. The platform includes both hardware and software components and has been made available to the wireless research and education communities. Specifically, they have incorporated sixteen USRPs into Utah’s Emulab testbed, which is used by over a thousand researchers and students worldwide. Emulab provides its users with both hardware and software: Beyond providing "raw" access to the USRPs, Emulab preinstalls GNU Radio and custom software that allows the USRPs to be used as IP network devices. The Emulab SDR installation makes it possible for researchers and educators to "borrow" the highly programmable wireless technology that is required for advancing the wireless state of the art.
5.4.2 Evaluating A GNU Radio Testbed [16]

Stefan Valentin, Holger von Malm and Holger Karl of the University of Paderborn, studied the performance and implementation cost of a wireless testbed based on GNU Radio. They implemented a complete transceiver chain and measured the performance of the testbed running this chain. They concluded that, the performance achieved by the implemented testbed was low and that the system was not suitable for real time testbeds in the megabit range. However, the testbed programmer is supported by a comprehensive programming environment, which results in low preparation and implementation time and low programming complexity.

5.4.3 Flexible MAC/PHY Multihop Testbed [17]

Hydra is a flexible wireless network testbed being developed at the University of Texas, Austin. They focus on networks that support multiple wireless hops and networks, that take advantage of sophisticated PHY techniques, such as OFDM and MIMO. Hydra nodes consist of a flexible RF front-end and a general purpose machine with a software based MAC and PHY. Using the frameworks of Click modular router, the GNU Radio, Hydra facilitates prototyping of cross layer designs that require custom MAC and PHY protocols. The early results have shown Hydra to be a capable prototyping tool for wireless network research.
6.1 Tunnel.py Command Line Options

The tunnel.py example has an array of options, which can be accessed from the command line. There are 2 groups of options

a. Basic
   Modulation scheme (GMSK, DQPSK, DBPSK),
   Frequency of transmission (Depending on the daughter board),
   Bitrate,
   Tx and Rx subdevice (Depending on the side of the USRP),
   Tx amplitude, etc.

b. Expert
   Carrier Threshold,
   TUN device to use,
   Tx and Rx frequency,
   Interpolation rate,
   Samples per symbol,
   FPGA decimation rate,
   BW Time product,
   Logging flow graph data, etc.
6.2 Matlab GUI For Running The Program

Figure 6.1 Snapshot of the Matlab GUI interface for tunnel.py
The Matlab GUI in Figure 6.1 lists all the options which are in the GNU Radio communication example (tunnel.py) and also lists all the default values of parameters. After selecting the required options, the “Run Example” button runs the tunnel.py example. (The “Frequency” option is mandatory.)

The Run_Example callback function listed in Table 6.1, reads all the options and generates the command to run the tunnel.py example.

### Table 6.1 Callback function of the matlab interface

```matlab
function Run_Example_Callback(hObject, eventdata, handles)
    Command_line_args = '';%--------------------------- Basic Options ----------------------------
    Modulation_val = get(handles.Modulation,'Value');
    if Modulation_val == 1
        modulation_val = get(handles.modulation,'Value');
        switch modulation_val
            case 1,
                modulation_str = 'gmsk';
            case 2,
                modulation_str = 'dbpsk';
            case 3,
                modulation_str = 'dqlsk';
            otherwise,
                modulation_str = 'gmsk';
        end
        Command_line_args = sprintf('%s -m %s',Command_line_args,modulation_str);
    end
    Freq_val = get(handles.Freq,'Value');
    if Freq_val == 1
        freq_val = get(handles.freq,'String');
        freq_str = freq_val;
    else
        freq_str = '2.5G';
    end
```
% The selected value is added to the command line options string.
Command_line_args = sprintf('%s -f %s',Command_line_args,freq_str);
% Similarly all the options are read and the command line options string is generated.
% If the Log option is selected, depending on the modulation scheme selected, the appropriate Graph GUI is called

LogEnable_val = get(handles.LogEnable,'Value');
if LogEnable_val == 1
    Command_line_args = sprintf('%s --log',Command_line_args);

% Start the Graphs GUI
% Based on the Modulation type, start the appropriate graph GUI
switch modulation_val
    case 1,
        run GNURadio_Graphs_GUI_GMSK;
    case 2,
        run GNURadio_Graphs_GUI_DBPSK;
    case 3,
        run GNURadio_Graphs_GUI_DQPSK;
    otherwise,
        run GNURadio_Graphs_GUI_GMSK;
end
end

disp('Running Tunnel.py with the following selected options')

Final_Run_Command = '/root/GNURadio/GNURadio3.0/gnuradio/gnuradio-
examples/python/digital/tunnel.py';

Final_Run_Command = sprintf('%s %s',Final_Run_Command, Command_line_args);
% Run the tunnel.py example with the selected options
unix(Final_Run_Command);

6.3 The Logging Option

If the “Log” option is selected, it enables each of the processing blocks in
the flow graph to also write to a file_sink. For example, if the modulation method is
GMSK, the gmsk.py file defines the logging procedure. The code for the loggin of
GMSK modulator/demodulator blocks is listed in Table 6.2. The
modulator/demodulator blocks are connected to file sinks. The file sink block requires
2 parameters to be specified, the data type and filename.
Table 6.2 The logging using file sinks

```python
# /////////////////////////////////////////////////////////////////////////////
#                              GMSK modulator
# /////////////////////////////////////////////////////////////////////////////
def _setup_logging(self):
    print "Modulation logging turned on."
    self._fg.connect(self.nrz,
        gr.file_sink(gr.sizeof_float, "nrz.dat"))
    self._fg.connect(self.gaussian_filter,
        gr.file_sink(gr.sizeof_float, "gaussian_filter.dat"))
    self._fg.connect(self.fmmod,
        gr.file_sink(gr.sizeof_gr_complex, "fmmod.dat"))

# /////////////////////////////////////////////////////////////////////////////
#                            GMSK demodulator
# /////////////////////////////////////////////////////////////////////////////
def _setup_logging(self):
    print "Demodulation logging turned on."
    self._fg.connect(self.fmdemod,
        gr.file_sink(gr.sizeof_float, "fmdemod.dat"))
    self._fg.connect(self.clock_recovery,
        gr.file_sink(gr.sizeof_float, "clock_recovery.dat"))
    self._fg.connect(self.slicer,
        gr.file_sink(gr.sizeof_char, "slicer.dat"))
```

The captured data is in the format of 32-bit complex numbers. This needs to be converted to a readable format, a column vector. This is done by the function `read_complex_binary()` (Table 6.3).

This function takes 3 parameters, the filename and a start count and end count of lines to be read from the file.
### Table 6.3  Read_complex_binary function

```matlab
%% usage: read_complex_binary (filename, [s_count], [e_count])
%%
%%  open filename and return the contents as a column vector,
%%  treating them as 32 bit complex numbers

function v = read_complex_binary (filename, s_count, e_count)
    m = nargchk (1,3,nargin);
    if (m)
        % usage (m);
    end
    if (nargin < 3)
        s_count = 2
        e_count = Inf;
    end
    f = fopen (filename, 'rb');
    if (f < 0)
        v = 0;
    else
        t = fread (f, [s_count, e_count], 'float');
        fclose (f);
        v = t(1,:) + t(2,:)*i;
        [r, c] = size (v);
        v = reshape (v, c, r);
    end
```

The Matlab Graph GUI interface uses this calculated column vector to plot the captured data. Since the captured data results in very large files, the data is read in batches and then plotted. Each modulation technique uses a different number of processing blocks.
6.4 Supported Modulation Techniques

6.4.1 GMSK

By clicking the Plot Log Data / Refresh button, reads data from the log file in steps of N samples and plots the data onto the axes. N is configurable, and can be set to the required number of samples to analyze.
Figure 6.2  Initial screen for GMSK based logging

When the GMSK based logging is initiated, the GUI in Figure 6.2 is shown as the initial screen.
Figure 6.3  Plots of logged GMSK data

Figure 6.3 shows the dot plots of 5000 samples of the logged GMSK data. The callback function for the GMSK GUI (Table 6.4) defines the actions to be performed when the ‘Plot Log Data / Refresh’ button is clicked. When the ‘Plot Log Data / Refresh’ button is pressed for the first time, two persistent counters are initialized and used to parse through the log file in steps of the parameter LOG_JUMP_SIZE.
Table 6.4 Callback function for GMSK GUI

% The callback function activates and refreshes the axes with the appropriate data.
function refresh_Callback(hObject, eventdata, handles)

% The LOG_JUMP variable defines the read batch size of the log files. This can be changed as required
LOG_JUMP_SIZE = 300;

persistent S_LOG_SIZE;
persistent E_LOG_SIZE;

% The start and end line numbers are initialized.
if isempty(S_LOG_SIZE) & isempty(E_LOG_SIZE)
    S_LOG_SIZE = 2;
    E_LOG_SIZE = LOG_JUMP_SIZE;
end

% The file path of the log file is set.
EX_PATH = '/root/GNURadio/GNURadio3.0/gnuradio/gnuradio-
examples/python/digital/gmsk_dat_files/';

% The binary file is read into a vector
fmmod_data = read_complex_binary(strcat(EX_PATH,'fmmod.dat'), S_LOG_SIZE, E_LOG_SIZE);
% The correct axes is made the current_axes
axes(handles.fmmod_axes);
% The data is plotted on the current axes. The plot function can be changed to represent the data as required.
plot(real(fmmod_data), imag(fmmod_data));
% The process is repeated for all the processing blocks of the modulation scheme

% The start and end line numbers of the log file are updated
S_LOG_SIZE = E_LOG_SIZE + 1;
E_LOG_SIZE = E_LOG_SIZE + LOG_JUMP_SIZE;
6.4.2 DQPSK

The DQPSK modulation scheme uses a different flow graph with different number and type of processing blocks. Figure 6.4 shows a snapshot of the log data using DQPSK modulation scheme.

![DQPSK Log Graphs](image)

**Figure 6.4** Plots of logged DQPSK data
6.4.3 DBPSK

The DBPSK modulation scheme is very similar to the DQPSK scheme. Figure 6.5 shows a snapshot of the log data using DBPSK modulation scheme.

Figure 6.5  Plots of logged DBPSK data
Chapter 7

CONCLUSION AND FUTURE WORK

In this thesis, I explore GNU Radio with the objective of using it for a wireless network testbed, which will be primarily used for prototyping MAC protocols, which require dynamic use of the radio spectrum. In the process, I discovered that GNU Radio is a sophisticated & rich programming environment, but lacks good references for beginners. Anyone starting, required extensive Googling and discussion list mailings to get to the stage of creating exciting new radios. I wrote my GNU Radio Installation Notes, with the intention of it helping beginners get kick-started on their own fascinating Software Radio journey. From there, I developed a framework for MAC protocol implementations in GNU Radio.

From all the experimentation and exploring, I can conclude that, GNU Radio along with USRP, is a low cost platform with immense implementation flexibility and support for modular development and easier sharing of new blocks in the development community. Also, the GNU Radio Companion (GRC) is an excellent GUI tool, to test drive the capabilities of GNU Radio without having to write any code.

Even though the hardware components currently used pose a constraint in the terms of processing speed and data transfer rates, the situation can be circumvented by using faster ways of interfacing the hardware with the software components, such as PCI Xpress or the like, and by using faster CPUs.
As future work, the testbed can be improved to better support a wider range of protocols. Also, the GNU Radio components can be further improved for better performance and more capabilities. Aside from this, the documentation also can be made more comprehensive to serve as reference guides for Software Radio programmers.
Chapter 8

BIBLIOGRAPHY


Appendix A

CODE – MATLAB INTERFACE – TUNNEL.PY

function varargout = GNURadio_Tunnel_Log_tool(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
 gui_State = struct('gui_Name', mfilename, ...
   'gui_Singleton', gui_Singleton, ...
   'gui_OpeningFcn', @GNURadio_Tunnel_Log_tool_OpeningFcn, ...
   'gui_OutputFcn', @GNURadio_Tunnel_Log_tool_OutputFcn, ...
   'gui_LayoutFcn', [], ...
   'gui_Callback', []);
if nargin && ischar(varargin{1})
  gui_State.gui_Callback = str2func(varargin{1});
end
if nargout
  [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
  gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before GNURadio_Tunnel_Log_tool is made visible.
function GNURadio_Tunnel_Log_tool_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to GNURadio_Tunnel_Log_tool (see VARARGIN)

% --- Outputs from this function are returned to the command line.
function varargout = GNURadio_Tunnel_Log_tool_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% Get default command line output from handles structure
varargout{1} = handles.output;

%-------------------------------------------------------------------------
% --- Executes on button press in Run_Example.
function Run_Example_Callback(hObject, eventdata, handles)
% hObject    handle to Run_Example (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

Command_line_args = ";"

%--------------------------- Basic Options -----------------------------
Modulation_val = get(handles.Modulation,'Value');

if Modulation_val == 1
    modulation_val = get(handles.modulation,'Value');
    switch modulation_val
        case 1,
            modulation_str = 'gmsk';
        case 2,
            modulation_str = 'dbpsk';
        case 3,
            modulation_str = 'dqpsk';
        otherwise,
            modulation_str = 'gmsk';
    end
    Command_line_args = sprintf('%s -m %s',Command_line_args,modulation_str);
end

Freq_val = get(handles.Freq,'Value');
if Freq_val == 1
    freq_val = get(handles.freq,'String');
    freq_str = freq_val;
else
    freq_str = '2.5G';
end
Command_line_args = sprintf('%s -f %s',Command_line_args,freq_str);

Bitrate_val = get(handles.Bitrate,'Value');
if Bitrate_val == 1
    bitrate_val = get(handles.bitrate,'String');
    Command_line_args = sprintf('%s -r %s',Command_line_args,bitrate_val);
end

USRP_Tx_val = get(handles.USRP_Tx,'Value');
if USRP_Tx_val == 1
    usrp_tx_val = get(handles.usrp_tx,'String');
    Command_line_args = sprintf('%s -T %s',Command_line_args,usrp_tx_val);
end
USRP_Rx_val = get(handles.USRP_Rx,'Value');
if USRP_Rx_val == 1
  usrp_rx_val = get(handles.usrp_rx,'Value');
  Command_line_args = sprintf('%s -R %s',Command_line_args,usrp_rx_val);
end

Tx_Amplitude_val = get(handles.Tx_Amplitude,'Value');
if Tx_Amplitude_val == 1
  tx_amplitude_val = get(handles.tx_amplitude,'Value');
  Command_line_args = sprintf('%s -tx-amplitude=%s',Command_line_args,tx_amplitude_val);
end

Rx_Gain_val = get(handles.Rx_Gain,'Value');
if Rx_Gain_val == 1
  rx_gain_val = get(handles.rx_gain,'Value');
  Command_line_args = sprintf('%s --rx-gain=%s',Command_line_args,rx_gain_val);
end

MinMaxRxGain_val = get(handles.MinMaxRxGain,'Value');
if MinMaxRxGain_val == 1
  Command_line_args = sprintf('%s --show-rx-gain-range',Command_line_args);
end

Verbose_val = get(handles.Verbose,'Value');
if Verbose_val == 1
  Command_line_args = sprintf('%s -v',Command_line_args);
end

%--------------------------- Expert Options ---------------------------

Carrier_Thresh_val = get(handles.Carrier_Thresh,'Value');
if Carrier_Thresh_val == 1
  carrier_thresh_val = get(handles.carrier_thresh,'Value');
  Command_line_args = sprintf('%s -c %s',Command_line_args,carrier_thresh_val);
end

TUN_File_val = get(handles.TUN_File,'Value');
if TUN_File_val == 1
  tun_file_val = get(handles.tun_file,'Value');
  Command_line_args = sprintf('%s --tun-device-filename=%s',Command_line_args,tun_file_val);
end

Tx_Freq_val = get(handles.Tx_Freq,'Value');
if Tx_Freq_val == 1
  tx_freq_val = get(handles.tx_freq,'Value');
  Command_line_args = sprintf('%s -tx-freq=%s',Command_line_args,tx_freq_val);
end

Rx_Freq_val = get(handles.Rx_Freq,'Value');
if Rx_Freq_val == 1
  rx_freq_val = get(handles.rx_freq,'Value');
end
Command_line_args = sprintf('%s -rx-freq=%s',Command_line_args,rx_freq_val);
end

Interp_Rate_val = get(handles.Interp_Rate,'Value');
if Interp_Rate_val == 1
    interp_rate_val = get(handles.interp_rate,'String');
    Command_line_args = sprintf('%s -i %s',Command_line_args,interp_rate_val);
end

Samples_Per_Symbol_val = get(handles.Samples_Per_Symbol,'Value');
if Samples_Per_Symbol_val == 1
    samples_per_symbol_val = get(handles.samples_per_symbol,'String');
    Command_line_args = sprintf('%s -S %s',Command_line_args,samples_per_symbol_val);
end

Decim_Rate_val = get(handles.Decim_Rate,'Value');
if Decim_Rate_val == 1
    decim_rate_val = get(handles.decim_rate,'String');
    Command_line_args = sprintf('%s -d %s',Command_line_args,decim_rate_val);
end

BW_Time_val = get(handles.BW_Time,'Value');
if BW_Time_val == 1
    bw_time_val = get(handles.bw_time,'String');
    Command_line_args = sprintf('%s --bt=%s',Command_line_args,bw_time_val);
end

XS_BW_val = get(handles.XS_BW,'Value');
if XS_BW_val == 1
    xs_bw_val = get(handles.xs_bw,'String');
    Command_line_args = sprintf('%s --excess-bw=%s',Command_line_args,xs_bw_val);
end

Costas_Alpha_val = get(handles.Costas_Alpha,'Value');
if Costas_Alpha_val == 1
    costas_alpha_val = get(handles.costas_alpha,'String');
    Command_line_args = sprintf('%s --costas-alpha=%s',Command_line_args,costas_alpha_val);
end

Clock_Gain_val = get(handles.Clock_Gain,'Value');
if Clock_Gain_val == 1
    clock_gain_val = get(handles.clock_gain,'String');
    Command_line_args = sprintf('%s --gain-mu=%s',Command_line_args,clock_gain_val);
end

Clock_Freq_val = get(handles.Clock_Freq,'Value');
if Clock_Freq_val == 1
    clock_freq_val = get(handles.clock_freq,'String');
    Command_line_args = sprintf('%s --freq-error=%s',Command_line_args,clock_freq_val);
end
Clock_Omega_val = get(handles.Clock_Omega,'Value');
if Clock_Omega_val == 1
    clock_omega_val = get(handles.clock_omega,'String');
    Command_line_args = sprintf('%ss --omega-relative-limit=%s',Command_line_args,clock_omega_val);
end

Clock_Mu_val = get(handles.Clock_Mu,'Value');
if Clock_Mu_val == 1
    clock_mu_val = get(handles.clock_mu,'String');
    Command_line_args = sprintf('%ss --mu=%s',Command_line_args,clock_mu_val);
end

FUSB_Size_val = get(handles.FUSB_Size,'Value');
if FUSB_Size_val == 1
    fusb_size_val = get(handles.fusb_size,'String');
    Command_line_args = sprintf('%ss -B %s',Command_line_args,fusb_size_val);
end

FUSB_Num_val = get(handles.FUSB_Num,'Value');
if FUSB_Num_val == 1
    fusb_num_val = get(handles.fusb_num,'String');
    Command_line_args = sprintf('%ss -N %s',Command_line_args,fusb_num_val);
end

GrayCoding_val = get(handles.GrayCoding,'Value');
if GrayCoding_val == 1
    Command_line_args = sprintf('%ss --no-gray-code',Command_line_args);
end

%-------------------------------- Log Option --------------------------------

LogEnable_val = get(handles.LogEnable,'Value');
if LogEnable_val == 1
    Command_line_args = sprintf('%ss --log',Command_line_args);
    %Start the Graphs GUI
    %Based on the Modulation type, start the appropriate graph GUI
    switch modulation_val
    case 1,
        run GNURadio_Graphs_GUI_GMSK;
    case 2,
        run GNURadio_Graphs_GUI_DBPSK;
    case 3,
        run GNURadio_Graphs_GUI_DQPSK;
otherwise,
    run GNURadio_Graphs_GUI_GMSK;
end
end

disp('Running Tunnel.py with the following selected options')

    Final_Run_Command = '/root/GNURadio/GNURadio3.0/gnuradio/gnuradio-examples/python/digital/tunnel.py ';
    Final_Run_Command = sprintf('%s %s',Final_Run_Command, Command_line_args);

unix(Final_Run_Command);
Appendix B

CODE – MATLAB GRAPH GUI

GMSK

function varargout = GNURadio_Graphs_GUI_GMSK(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name', mfilename, ...
    'gui_Singleton', gui_Singleton, ...
    'gui_OpeningFcn', @GNURadio_Graphs_GUI_GMSK_OpeningFcn, ...
    'gui_OutputFcn', @GNURadio_Graphs_GUI_GMSK_OutputFcn, ...
    'gui_LayoutFcn', [], ...
    'gui_Callback', []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end
if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before GNURadio_Graphs_GUI_GMSK is made visible.
function GNURadio_Graphs_GUI_GMSK_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to GNURadio_Graphs_GUI_GMSK (see VARARGIN)

% Choose default command line output for GNURadio_Graphs_GUI_GMSK
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes GNURadio_Graphs_GUI_GMSK wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = GNURadio_Graphs_GUI_GMSK_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

%--------------------------------------------------------
%Put code for the creation of axes on the gui here
%--------------------------------------------------------

%-------------------RefreshGraphs---------------------
% --- Executes on button press in refresh.
function refresh_Callback(hObject, eventdata, handles)
% hObject    handle to refresh (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
    disp('Refreshing Graphs with new Log Data');
    LOG_JUMP_SIZE = 5000;

    persistent S_LOG_SIZE;
    persistent E_LOG_SIZE;

    if isempty(S_LOG_SIZE) & isempty(E_LOG_SIZE)
        S_LOG_SIZE = 2;
        E_LOG_SIZE = LOG_JUMP_SIZE;
    end

    EX_PATH = '/root/GNURadio/GNURadio3.0/gnuradio/gnuradio-examples/python/digital/gmsk_dat_files/';
    fmmod_data = read_complex_binary(strcat(EX_PATH,'fmmod.dat'), S_LOG_SIZE, E_LOG_SIZE);
    axes(handles.fmmod_axes);
    plot(real(fmmod_data), imag(fmmod_data),'.');

    nrz_data = read_complex_binary(strcat(EX_PATH,'nrz.dat'), S_LOG_SIZE, E_LOG_SIZE);
    axes(handles.nrz_axes);
    plot(real(nrz_data), imag(nrz_data) ,'.');

    gaussian_data = read_complex_binary(strcat(EX_PATH,'gaussian_filter.dat'), S_LOG_SIZE, E_LOG_SIZE);
    axes(handles.gaussian_axes);
    plot(real(gaussian_data), imag(gaussian_data));

    fmdemod_data = read_complex_binary(strcat(EX_PATH,'fmdemod.dat'), S_LOG_SIZE, E_LOG_SIZE);
    axes(handles.fmdemod_axes);
    plot(real(fmdemod_data), imag(fmdemod_data) ,'.');

    clock_data = read_complex_binary(strcat(EX_PATH,'clock_recovery.dat'), S_LOG_SIZE, E_LOG_SIZE);
    axes(handles.clock_axes);
plot(real(clock_data), imag(clock_data), '.');

slicer_data = read_complex_binary(strcat(EX_PATH,'slicer.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.slicer_axes);
plot(real(slicer_data), imag(slicer_data), '.');

S_LOG_SIZE = E_LOG_SIZE + 1;
E_LOG_SIZE = E_LOG_SIZE + LOG_JUMP_SIZE;

%% usage: read_complex_binary (filename, [count])
%%
%% open filename and return the contents as a column vector,
%% treating them as 32 bit complex numbers
function v = read_complex_binary (filename, s_count, e_count)
m = nargchk (1,3,margin);
if (nargin < 3)
    s_count = 2
    e_count = Inf;
end

f = fopen (filename, 'rb');
if (f < 0)
v = 0;
else
t = fread (f, [s_count, e_count], 'float');
cfclose (f);
v = t(1,:) + t(2,:)*i;
[r, c] = size (v);
v = reshape (v, c, r);
end

DQPSK

function varargout = GNURadio_Graphs_GUI_DQPSK(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name', mfilename, ... 'gui_Singleton', gui_Singleton, ... 'gui_OpeningFcn', @GNURadio_Graphs_GUI_DQPSK_OpeningFcn, ... 'gui_OutputFcn', @GNURadio_Graphs_GUI_DQPSK_OutputFcn, ... 'gui_LayoutFcn', [], ... 'gui_Callback', []);
if nargin && ischar(varargin{1})
gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT
function GNURadio_Graphs_GUI_DQPSK_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to GNURadio_Graphs_GUI_DQPSK (see VARARGIN)

% Choose default command line output for GNURadio_Graphs_GUI_DQPSK
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes GNURadio_Graphs_GUI_DQPSK wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = GNURadio_Graphs_GUI_DQPSK_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

%--------------------------------------------------------
%Put code for the creation of axes on the gui here
%--------------------------------------------------------

%-----------------RefreshGraphs---------------------
% --- Executes on button press in refresh.
function refresh_Callback(hObject, eventdata, handles)
% hObject    handle to refresh (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

disp('Refreshing Graphs with new Log Data');
LOG_JUMP_SIZE = 5000;
persistent S_LOG_SIZE;
persistent E_LOG_SIZE;
if isempty(S_LOG_SIZE) & isempty(E_LOG_SIZE)
    S_LOG_SIZE = 2;
    E_LOG_SIZE = LOG_JUMP_SIZE;
end

EX_PATH = '/root/GNURadio/GNURadio3.0/gnuradio/gnuradio-examples/python/digital/dqpsk_dat_files/';
%--------Modulation Log Data--------

b2c_data = read_complex_binary(strcat(EX_PATH,'.bytes2chunks.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.b2c_axes);
plot(real(b2c_data), imag(b2c_data) ,'.');

grayenc_data = read_complex_binary(strcat(EX_PATH,'graycoder.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.grayenc_axes);
plot(real(grayenc_data), imag(grayenc_data) ,'.');

diffenc_data = read_complex_binary(strcat(EX_PATH,'diffenc.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.diffenc_axes);
plot(real(diffenc_data), imag(diffenc_data) ,'.');

c2s_data = read_complex_binary(strcat(EX_PATH,'chunks2symbols.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.c2s_axes);
plot(real(c2s_data), imag(c2s_data) ,'.');

rrc_data = read_complex_binary(strcat(EX_PATH,'rrc_filter.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.rrc_axes);
plot(real(rrc_data), imag(rrc_data) ,'.');

%--------Demodulation Log Data--------

pre_data = read_complex_binary(strcat(EX_PATH,'prescaler.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.pre_axes);
plot(real(pre_data), imag(pre_data) ,'.');

agc_data = read_complex_binary(strcat(EX_PATH,'agc.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.agc_axes);
plot(real(agc_data), imag(agc_data) ,'.');

costas_data = read_complex_binary(strcat(EX_PATH,'costas_loop.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.costas_axes);
plot(real(costas_data), imag(costas_data) ,'.');

rrc2_data = read_complex_binary(strcat(EX_PATH,'rrc_filter.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.rrc2_axes);
plot(real(rrc2_data), imag(rrc2_data) ,'.');

clock_data = read_complex_binary(strcat(EX_PATH,'clock_recovery.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.clock_axes);
plot(real(clock_data), imag(clock_data) ,'.');

diffdec_data = read_complex_binary(strcat(EX_PATH,'diffdec.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.diffdec_axes);
plot(real(diffdec_data), imag(diffdec_data) ,'.');

slicer_data = read_complex_binary(strcat(EX_PATH,'slicer.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.slicer_axes);
plot(real(slicer_data), imag(slicer_data) ,'.');

graydec_data = read_complex_binary(strcat(EX_PATH,'gray_decoder.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.graydec_axes);
plot(real(graydec_data), imag(graydec_data) ,'.');
unpack_data = read_complex_binary(strcat(EX_PATH,'unpack.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.unpack_axes);
plot(real(unpack_data), imag(unpack_data) ,'.');

S_LOG_SIZE = E_LOG_SIZE + 1;
E_LOG_SIZE = E_LOG_SIZE + LOG_JUMP_SIZE;

%%% usage: read_complex_binary (filename, [count])
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% open filename and return the contents as a column vector,
%%% treating them as 32 bit complex numbers
function v = read_complex_binary (filename, s_count, e_count)
  m = nargchk (1,3,nargin);
  if (nargin < 3)
    s_count = 2
    e_count = Inf;
  end

  f = fopen (filename, 'rb');
  if (f < 0)
    v = 0;
  else
    t = fread (f, [s_count, e_count], 'float');
    fclose (f);
    v = t(1,:) + t(2,:)*i;
    [r, c] = size (v);
    v = reshape (v, c, r);
  end

function varargout = GNURadio_Graphs_GUI_DBPSK(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
 gui_State = struct('gui_Name', mfilename, ...
 'gui_Singleton', gui_Singleton, ...
 'gui_OpeningFcn', @GNURadio_Graphs_GUI_DBPSK_OpeningFcn, ...
 'gui_OutputFcn', @GNURadio_Graphs_GUI_DBPSK_OutputFcn, ...
 'gui_LayoutFcn', [], ...
 'gui_Callback', []);
if nargin && ischar(varargin{1})
  gui_State.gui_Callback = str2func(varargin{1});
end

if nargin
  [varargout{1:nargin}] = gui_mainfcn(gui_State, varargin{:});
else
  gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

% --- Executes just before GNURadio_Graphs_GUI_DBPSK is made visible.
function GNURadio_Graphs_GUI_DBPSK_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to GNURadio_Graphs_GUI_DBPSK (see VARARGIN)

% Choose default command line output for GNURadio_Graphs_GUI_DBPSK
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes GNURadio_Graphs_GUI_DBPSK wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = GNURadio_Graphs_GUI_DBPSK_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

%--------------------------------------------------------
%Put code for the creation of axes on the gui here
%--------------------------------------------------------

%-----------------RefreshGraphs---------------------
% --- Executes on button press in refresh.
function refresh_Callback(hObject, eventdata, handles)
% hObject    handle to refresh (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

disp('Refreshing Graphs with new Log Data');
LOG_JUMP_SIZE = 5000;
persistent S_LOG_SIZE;
persistent E_LOG_SIZE;
if isempty(S_LOG_SIZE) & isempty(E_LOG_SIZE)
    S_LOG_SIZE = 2;
    E_LOG_SIZE = LOG_JUMP_SIZE;
end
EX_PATH = '/root/GNURadio/GNURadio3.0/gnuradio/gnuradio-examples/python/digital/dbpsk_dat_files/';
%--------Modulation Log Data--------

b2c_data = read_complex_binary(strcat(EX_PATH,'.bytes2chunks.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.b2c_axes);
plot(real(b2c_data), imag(b2c_data) ,'.');

symmap_data = read_complex_binary(strcat(EX_PATH,'graycoder.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.symmap_axes);
plot(real(symmap_data), imag(symmap_data) ,'.');

diffenc_data = read_complex_binary(strcat(EX_PATH,'diffenc.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.diffenc_axes);
plot(real(diffenc_data), imag(diffenc_data) ,'.');

c2s_data = read_complex_binary(strcat(EX_PATH,'chunks2symbols.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.c2s_axes);
plot(real(c2s_data), imag(c2s_data) ,'.');

rrc_data = read_complex_binary(strcat(EX_PATH,'rrc_filter.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.rrc_axes);
plot(real(rrc_data), imag(rrc_data) ,'.');

%--------Demodulation Log Data--------

pre_data = read_complex_binary(strcat(EX_PATH,'prescaler.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.pre_axes);
plot(real(pre_data), imag(pre_data) ,'.');

agc_data = read_complex_binary(strcat(EX_PATH,'age.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.agc_axes);
plot(real(agc_data), imag(agc_data) ,'.');

costas_data = read_complex_binary(strcat(EX_PATH,'costas_loop.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.costas_axes);
plot(real(costas_data), imag(costas_data) ,'.');

rrc2_data = read_complex_binary(strcat(EX_PATH,'rrc_filter.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.rrc2_axes);
plot(real(rrc2_data), imag(rrc2_data) ,'.');

clock_data = read_complex_binary(strcat(EX_PATH,'clock_recovery.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.clock_axes);
plot(real(clock_data), imag(clock_data) ,'.');

diffdec_data = read_complex_binary(strcat(EX_PATH,'diffdec.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.diffdec_axes);
plot(real(diffdec_data), imag(diffdec_data) ,'.');

slicer_data = read_complex_binary(strcat(EX_PATH,'slicer.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.slicer_axes);
plot(real(slicer_data), imag(slicer_data) ,'.');

symmap2_data = read_complex_binary(strcat(EX_PATH,'symbol_mapper.dat'), S_LOG_SIZE, E_LOG_SIZE);
axes(handles.symmap2_axes);
plot(real(symmap2_data), imag(symmap2_data) ,'.');
unpack_data = read_complex_binary(strcat(EX_PATH,'unpack.dat'), S_LOG_SIZE, E_LOG_SIZE);
 axes(handles.unpack_axes);
 plot(real(unpack_data), imag(unpack_data) ,'.');
 S_LOG_SIZE = E_LOG_SIZE + 1;
 E_LOG_SIZE = E_LOG_SIZE + LOG_JUMP_SIZE;

%% usage: read_complex_binary (filename, [count])
%%
%%% open filename and return the contents as a column vector,
%%% treating them as 32 bit complex numbers
function v = read_complex_binary (filename, s_count, e_count)
 m = nargchk (1,3,nargin);
 if (nargin < 3)
    s_count = 2
    e_count = Inf;
 end

 f = fopen (filename, 'rb');
 if (f < 0)
    v = 0;
 else
    t = fread (f, [s_count, e_count], 'float');
    fclose (f);
    v = t(1,:) + t(2,:)*i;
    [r, c] = size (v);
    v = reshape (v, c, r);
 end