SDR Tutorial

Joan Ferreras, Samuel Schmidt, Willie Harrison

Abstract—**A major issue in communication systems has been the difficulty of creating cost effective, easily reconfigurable hardware. Even when reconfigurability is possible, it can be costly and time consuming to reconfigure or upgrade the existing system. This work will provide a concise overview of software defined radio (SDR) development and available frameworks which make fast, reconfigurable high-frequency communication systems a reality. We will explore the SDR landscape and provide details and common pitfalls on several software framework families for implementing communication systems with the goal of easing the learning curve for new users.**

Index Terms—**Software defined radio, GNU RADIO, Software frameworks, Tutorial**

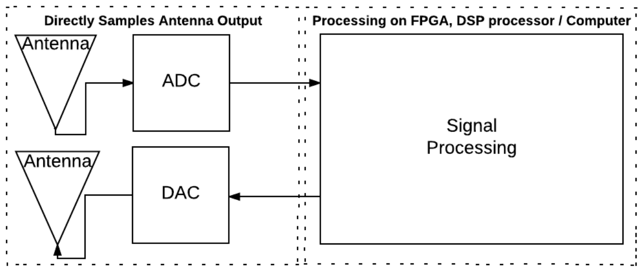

Directly Samples Antenna Output
Processing on FPGA, DSP processor / Computer
Antenna
ADC
Antenna
DAC
Signal Processing

Fig. 1. The ideal software-defined radio.

I. Introduction

SOFTWARE defined radio (SDR) was first introduced to the public by Joseph Mitola in the IEEE National Telesystems Conference of 1992 [1]. SDR is a pervasive radio communication system with added reconfigurability. Among others, SDRs can be used for the prototyping and development of cognitive radio [2], audio processing (as will be shown on this paper), and satellite communication research [3].The goal of SDR is to remove certain hardware components for added reconfigurability, namely the mixers, filters, amplifiers, modulators, and detectors, which are instead implemented in software. Due to advances in electronics, specifically in radio frequency (RF) transceivers and high speed, high resolution digital-to-analog (DAC) and analog-to-digital (ADC) converters, SDRs are replacing traditional fixed-function hardware modems [4]. Different types of commercial SDRs with various specifications have been built [5]. These radios are paired with frameworks such as GNU Radio, MATLAB, REDHAWK, and LabView to carry out research, teaching and easy prototyping of communication systems [6] [7] [8]. For smaller scale/timing sensitive projects, each commercial SDR comes with a C/C++ application program interface (API). For an example, see [9].

Current research on Software Defined Radio focuses on the application of SDR technologies. Notably on software frameworks for SDRs such as the REDHAWK development environment [10]. Although useful from a system-engineering perspective, the work does not provide a clear overview on what is necessary for an inexperienced user to begin research using SDRs. Similar work has also been done on the implementation of software framework for software-defined radio systems [11]. Neither of those equip an inexperienced user with the guidance needed to start implementing communication systems through relevant examples. Furthermore, as noted in [12] SDRs give way to endless educational opportunities but currently there is a deep void in the context of telecommunications systems. Despite the multitude of uses for SDRs, very little has been done to bridge the gap for new researchers and undergraduates to be part of this promising technology. This paper aims to provide a centralized overview of SDR development using GNU Radio and the Universal Software Radio Peripheral (USRP) through simulations, real-channel transmission and solutions to common pitfalls. A basic understanding of digital signal processing [13], experience with a UNIX-like platform, and a background in communication systems [14] is assumed. The GNU Radio software framework, being open-source and one of the most popular in academia, will be used alongside the USRP to give working examples of wireless communication. The GNU Radio tutorials [15], are a good source to obtain an understanding of how SDR facilitates transition from hardware components to software modules. The rest of the paper is organized as follows. In Section I, we have introduced Software Defined Radio and appealed to the need of a research paper which introduces the technology to new-comers. Section II will give background information on the SDR concept and a brief overview of the ideal software radio as well as a comprehensive summary. Section III presents an overview of available SDRs and useful software frameworks. Section IV provides a tutorial for getting started with the GNU Radio framework and the USRP, while Section V presents relevant working examples. Finally, we conclude the paper in Section VI.

Electrical and Computer Engineering Department

II. The SDR concept

SDR originates from a desire to have fast and reconfigurable communication systems. Typically these systems are purpose built for one application, with no or little thought given to upgrading or expanding the system. SDR, however, allows for flexible systems to be deployed. As a result, the user has access to highly reconfigurable radio frequency (RF) and digital signal processing (DSP) capabilities.

The main advantage of using software radios is the reconfigurable RF front end [16]. In Fig. 1, the ideal SDR is presented.

The system directly samples the RF spectrum and expects the signal processing block to account for any aliasing, jitter, and other problems that arise from very fast ADC. After sampling, all functions traditionally performed by analog components are performed digitally. Frequency translation of signals could be done either by bandpass sampling, i.e. under sample the spectrum to intentionally introduce an alias of the desired signal at baseband. Or, if the sampling rate is high enough, the SDR could digitally mix the signals down to baseband. Either solution requires exceptional care to avoid aliasing. The signals would then be resampled to allow the user to select any sample rate and signal bandwidth. Unfortunately, high speed ADC/DACs do not have the necessary resolution required by most Software Defined Radio (SDR) projects to make this a reality.

To overcome this problem, most SDRs incorporate a reconfigurable RF analog front end, Fig. 2.

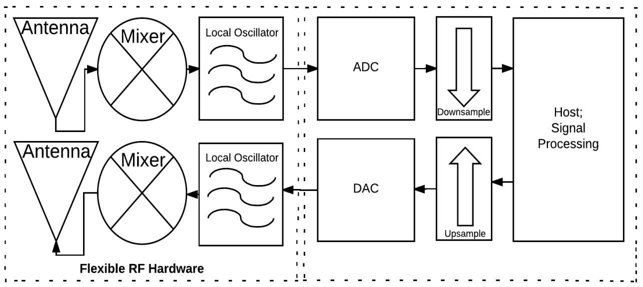


Fig. 2. The real software-defined radio.

This includes phase locked loop (PLL) circuits that generate local oscillator (LO) frequencies used to mix signals up or down in frequency for transmitting or receiving. For the receiver, the signal of interest typically moves through the following operations: variable bandwidth anti-aliasing filter, high speed ADC, and decimation to provide the user the ability to select from many different sampling rates. The transmitter signal path is similar, but traversed in the opposite order. First the signal is interpolated, then passed through a DAC, and finally filtered before mixing up to the carrier frequency.

A. Literature Search

At its infancy, academic research and commercial development of SDR gained traction thanks, in part, to the research done by Joe Mitola in his IEEE Communications Magazine journal submission titled “The software radio architecture”[17]. In contrast, most recent research uses SDR as a tool to a means instead of the focus. As the interest in software radio continues in an upward trend, it is no surprise that software radios have become widely used in academic research.

The literature research presented in this paper will be discussed in two different ways: thematically and methodologically. By thematic analysis, we will examine how researchers have used SDR. Through a methodological review, the team will compare and contrast previous research to the work presented in this paper.

1) Thematic Review: As part of an experimental project [8], The Department of Physics and Engineering Technology at Bloomsburg University of Pennsylvania used the GNU Radio Companion (GRC) along with USRPs in an effort to teach undergraduate students about electronic communications. The abstraction provided by the flowgraphs in the GRC was used with the goal of increasing the abstract thinking of the students. The approach was well received by the students and proved to be successful as a supplement to the lectures. In our paper, we will present the reader with real-channel examples of working communication systems. Furthermore, we encourage the reader to delve into the code presented to get a better understanding on how to simulate and develop their own communication systems.

In [18], SDR is presented as one of the most disruptive technologies at the time of writing. A quick overview of the “ideal” software defined radio is presented with a focus on how pervasive SDR was in the military-industrial and commercial complex. SDR continues to be the most essential technology in industry for fast prototyping of communication systems.

As part of a series of tutorials on instrumentation and measurement, [16] sets out to give an in-depth investigation on the digital signal processing aspect of SDRs. Furthermore, it contrasts the difference between DSP radio and SDR, giving a good overview of the pros and cons of embedded DSP systems in modern radios versus SDR.

One of the major roadblocks with SDR is the difficulty of maintaining required data rates across interprocessor interfaces. In [17], Joe Mitola acknowledges this difficulty while presenting many other problems and introducing the physical architecture of SDRs. This piece of literature is perhaps the most referenced SDR work because it outlines the considerations that must be kept in mind when creating a software radio.

2) Methodological Review: Signal processing engineers are often concerned with dividing the work among several engineers. Flowgraph driven development frameworks like REDHAWK allow engineers to work in tandem in a modular context. In [10], Robert et al. go in depth at describing the REDHAWK SDR framework. Similarly to our paper, they give a brief overview of other available frameworks. Unlike ours, however, the focus of [10] is to describe the REDHAWK environment without real-channel working examples.

In perhaps the work that most resembles ours, Zhang, in his master’s thesis, describes the GNU Radio architecture and the GRC and culminates with working examples [19]. Most interestingly, Zhang showcases a real-time digital video broadcasting system which transfers the information by linking the USRPs through the Ethernet ports. The other examples provided are the FM receiver, GSM Scanner, and OFDM which are available in the GNU Radio standard libraries. It should be noted that the video broadcasting example is a simulation over the ports. Our work, on the other hand, provides both simulations and real-channel implementation using the antennas.

III. Current Developments in commercial SDR

Thanks to advances in hardware such as RF integrated circuits and digital-signal-processing field programmable gate arrays, software-defined radio has become the norm for military tactical radios and commercial cellular devices. Currently, SDRs are widely used in spectrum monitoring, military communications and signal intelligence.

A. Overview of available SDRs

Several families of SDRs have emerged in recent years, a fairly comprehensive list is given here [20]. These radios cover frequencies from Direct Current (DC) to many GHz, with every radio having different specifications, form-factors, and price. The sheer number of options can be extremely daunting, we present an overview of the most used SDRs arranged by transmitting capabilities but encourage the reader to further explore the numerous software radios on the market.

1) Rx only SDR: For applications that do not require transmitting signals, a number of SDR options are available. Two of the most common (and cost efficient) SDR platforms are the RTL-SDR [21] and the Airspy [22]. Possible applications of such an SDR are as a Global Navigation Satellite System (GNSS) receiver or research that requires a multi receiver environment for new security or network schemes [23]. These relatively cheap SDRs provide an excellent platform for a variety of research topics.

Despite its relatively bad RF performance and limited sampling rate, the RTL-SDR has obtained significant support with a number of interesting projects being developed. Some of these projects can be found in [24]. The RTL-SDR has also gained traction as an educational tool due to its low price and high level of support in the SDR [25]. Some common problems encountered with the RTL-SDR is a power spike at DC caused by Inphase and Quadrature (IQ) imbalance, frequency errors due to the poor stability oscillator, and incorrectly calibrated sampling rate. Each of these problems have solutions: [26], [27]. Due to the community support, the RTL-SDR has greatly impacted the SDR community.

The Airspy is more expensive than the RTL-SDR but comes with better RF and digital capabilities. The Airspy has a better tuner IC and oscillator coupled with faster sampling rates and increased resolution digital to analog converter (DAC)/analog to digital converter (ADC). The platform has also strived to solve the problems that plague the RTL-SDR, notably the IQ imbalance has been corrected out of the box.

2) Tx and Rx SDR: Most SDR families provide transmitting and receiving capabilities. Some systems are half-duplex. In other words, they can transmit and receive data, but not simultaneously. Other systems are full-duplex, capable of true simultaneous transmit and receive. We briefly introduce some of these SDR families here:

- Hack RF [28]
- Blade RF [29]
- FMCOMMS2/3/4/5 [30]
- Ettus Research: bus series [31], networked series [32], and embedded series [33].

The Hack RF is a half-duplex SDR, open source, and has strong community support.

On the other hand, the Blade RF is a full-duplex system, open source, and has strong community support.

FMCOMMS is technically only an SDR daughter card as it relies on an FPGA for operation and is not open source but provides a good platform for implementing FPGA oriented designs quickly using this RF Agile Transceiver [34].

Ettus has many different families of SDR and has become the de facto industry standard for SDR. Their SDR platforms are generally the most capable, have a lot of support, and use open source drivers. Their platforms run from embedded systems to large networked machines that stream large amounts of data very quickly [35]. The most common is, perhaps, the bus series. It is relatively cheap, capable of 2x2 multiple input multiple output (MIMO), and commands a large portion of the frequency spectrum (70 - 6000 MHz). Ettus devices communicate with a host processor using the open source USRP Hardware Driver (UHD).

B. Overview of SDR softwares

Generally, the development tools for SDR have been focused on model based design principles. Among these tools, are: MATLAB, REDHAWK, LabView, and GNU Radio. Due to the latter being open-source, it continues to be the most widely used in academic research submitted to IEEE. For those reasons, the examples provided in this paper will be carried out using GNU Radio. It is important to note that the other platforms are as capable and each platform has its strengths and weaknesses as will be discussed. For the more experienced user, options like writing your own code using the windows SDR software package which contains SDR# from airspy [36].

MATLAB is very well established in the academic circles and as such has a large user base with excellent tutorials, numerous examples, and a dedicated support team. Since the addition of Simulink, MATLAB has been able to support model-based design methods that lend themselves to FPGA development, Digital Signal Processing (DSP), and SDR. Simulink currently supports the RTL-SDR, FMCOMMS, and Ettus SDRs [37]. Since Ettus was acquired by National Instruments, LabView is capable to support all Ettus SDRs.

REDHAWK is a digital signal processing SDR development framework. Like GNU Radio, it is free and open source. Developed by the Department of Defense (DoD) in order to push forward rapid prototyping of radio systems and test them, it excels in the processing of real-time software radio applications. REDHAWK's strength lies in its ability to have functionality parts cloned within multiple systems. It employs a middle-ware that allows different objects to work together regardless of its vendor. Its SDR core framework is based on Joint Tactical Radio System (JTRS) software communication architecture. It includes a waveform workshop and a set of tools for rapid development of SDR components and waveform applications. Its open-source nature allows for an ever increasing amount of components and waveform applications.

LabView is another model based development environment which enables the less experienced in programming to prototype and test ideas. Its graphical programming language called "G" is used to generate the dataflow models which remove the need for sequential lines of text code. These dataflow models are known as templates in LabView and allow for prototyping advanced research areas such as multiple input multiple output, RF compressive sampling, spectrum monitoring, coginitive radio and direction finding [38]. A major drawback of LabView is the cost of the software, unlike GNU Radio, it is not free. To offset the cost, followers have developed a LabView interface for RTL-SDR. Nevertheless, the RTL-SDR is limited in usability due to being RX only, having a small frequency range, and having no ability to support FPGA digital signal processing. This is a considerable hindrance from an academia perspective but enables amateur development.

IV. GNU Radio

GNU Radio is an open source project that aims to provide a free SDR development platform. The GNU Radio codebase provides a framework for signal processing, it is mostly a way of abstracting code and gives a model based framework from which very complex systems can be efficiently constructed.

A. Overview

A simple way show how GNU Radio works is to consider testing a filter's frequency response using white noise as an example:

- Generate white noise with variance: σ^2, see Fig. 3
- Filter
- Plot frequency response, see Fig. 4

Each bullet represents a "block" in GNU Radio, that is every operation is contained as a discrete system capable of being easily used across multiple applications. Multiple blocks are connected together in what is called a flowgraph, see Fig. 3.

Initially the data is buffered and then streamed through the blocks. The buffer size is controlled by the GNU Radio scheduler. The scheduler [39] tries to minimize latency and keep data flowing at a constant rate by changing the size of the input/output buffers of each block. Something to note is that each can be contained in its own thread, allowing for parallel execution of blocks and thus better emulation of hardware. The downside of this is extra overhead for the OS to deal with.

The GNU Radio base is written in C++ and Python with everything tied together with the Simplified Wrapper and Interface Generator (SWIG).

1) Installation: The recommended installation utility is PyBOMBS, a package manager written in Python which can be installed using PiP [40]. Building from source is generally not recommended due to the large amount of dependencies GNU Radio requires. Most current Linux operating systems have pre-compiled binaries of GNU Radio available in their respective package managers. This is perhaps the easiest way to install GNU Radio, but the versions in the package managers are typically not the latest release. PyBOMBS will always install the latest stable release and provides an easy way to update GNU Radio and all of its dependencies. The default PyBOMBS configuration is sufficient for most purposes.

GNU Radio and Ettus both provide images of Linux operating systems with GNU Radio and hardware drivers already installed and configured. These are generally meant to be used from portable storage devices like USB Drives. They could also be used with virtual machine (VM) software, but using a VM will hurt the performance of any hardware connected because the VM has to virtualize a communication port.

Windows installers are easily found [41], as are Mac OS X versions [42]. It is worth mentioning that OS X is better supported due to its similarities with Linux.

B. Usage

As GNU Radio provides a framework for signal processing and interfacing with hardware, one can write their own code that strings together GNU Radio signal processing blocks. This is typically done to prototype very simple systems, or to integrate existing functionality provided by GNU Radio (and its super optimized libraries) into another project. The best example of this being done in practice is the 'Global Navigation Satellite System Software Defined Radio' (GNSS SDR) project [43]. The GNU Radio framework is encompassed libraries wrapped within a very well constructor application program interface (API). Since GNU Radio is open source, the GNSS SDR team is able extend the functionality of the available libraries by inserting specialized code and overriding existing functions. As a result, they have created a GNSS software-defined receiver software architecture.

The GNSS SDR has two major abstractions: the control plane [44], and the signal processing block [45]. The control plane is responsible for dynamically activating and deactivating the channels as satellite signals fade in and out during runtime. On the other hand, the signal processing block has the digital signal processing algorithm implementations.

Distinct from the GNU Radio framework, the GNU Radio project provides a graphical interface to development known as the GNU Radio Companion (GRC). When someone mentions they are using GNU Radio, this is typically what is meant. It allows for easy model based design as the data flow can be easily visualized through the flowgraph blocks and can be inspected using the "QT" blocks. Very complex systems can be quickly designed and optimized. The companion auto generates Python code which strings together the blocks and is called by a Python interpreter to run. Thus the companion is able to save the user considerable development time.

When developing a communication system that requires a file to be read, it is useful to set the 'run to completion' flag to true. This ensures that the file is read completely by sending a special character through the blocks to indicate the source block has reached the end of file. This special character is used to avoid instances when a flowgraph keeps running even though the work threads are finished. It stands to reason that a flowgraph can be taken and stripped down to its code in order to be run in systems that do not contain a display or for small scale testing.

1) Extending GNU Radio's base functionality: GNU Radio supports developing code in either the Python [46] or C++ languages. Typically signal processing blocks are written in

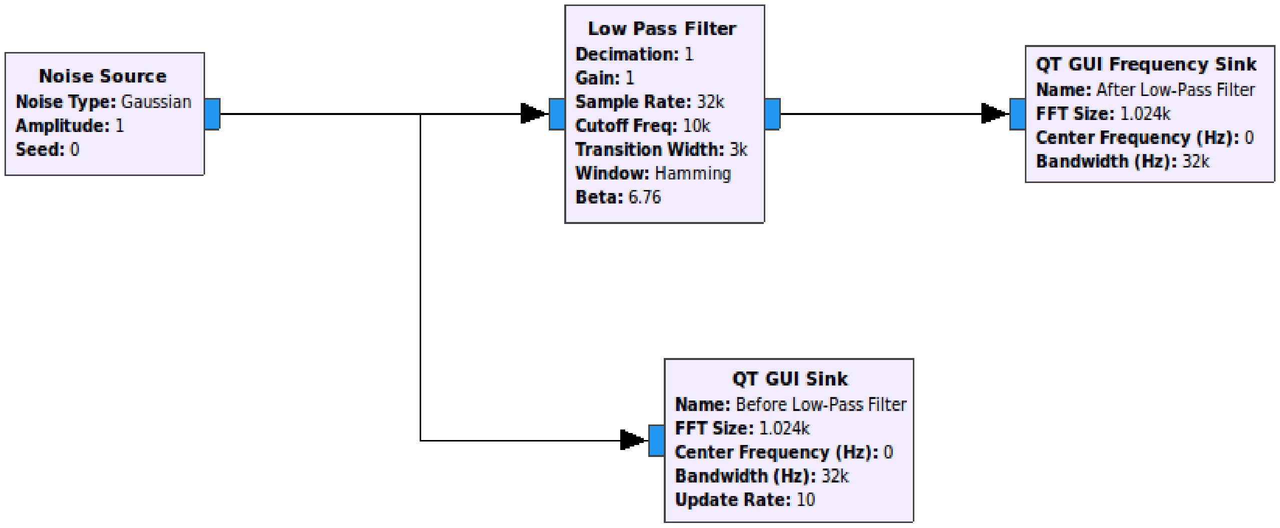


Fig. 3. Creating Gaussian noise and sending it to QT Sinks for observation.

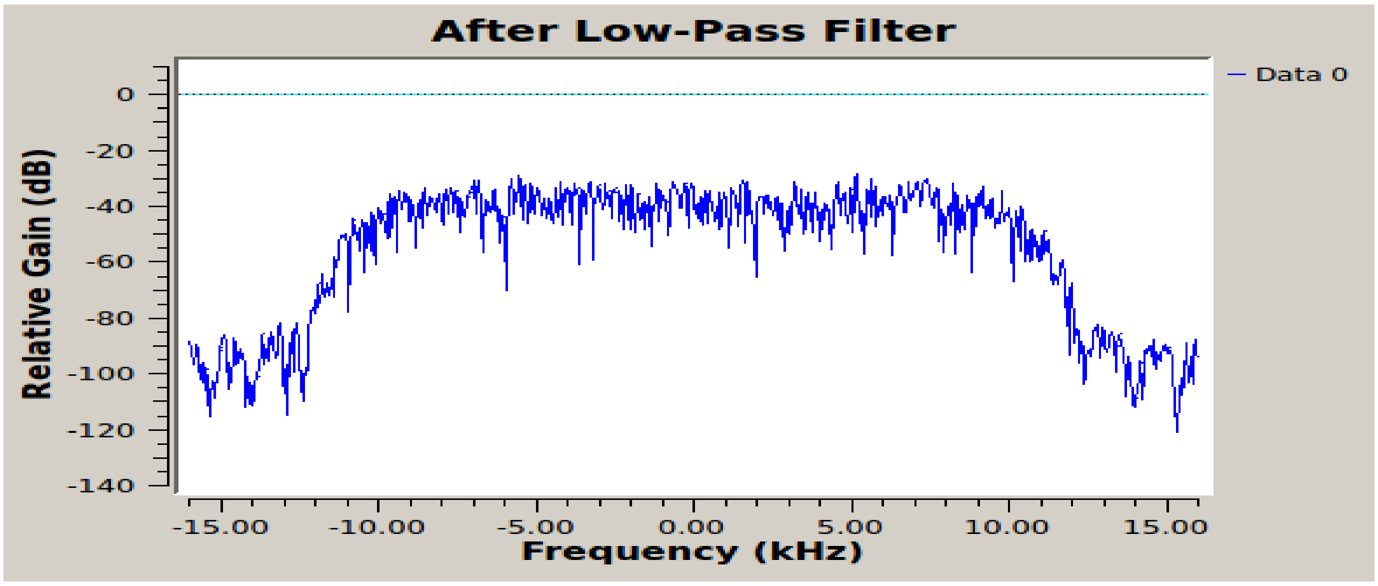


Fig. 4. Frequency response of Gaussian noise after low-pass filtering.

C++ and hierarchical blocks are written in Python. However, Python offers an easy prototyping platform and numerical libraries such as Numpy and Scipy [47] allow for some fast signal processing abilities.

GNU Radio provides a C++ math library known as Vector-Optimized Library of Kernals (VOLK) to help implementing common signal processing tasks such as FIR filters and Fast Fourier Transforms (FFTs). This library is mostly unknown outside of GNU Radio but it has great functionality.

Some notable examples using GNU Radio can be found in [43] and [48]. These projects highlight the great potential GNU Radio has as a research tool. In [43] a fully functioning multi-constellation GNSS receiver is implemented using the GNU Radio framework. It provides an excellent platform for developing tracking and DSP algorithms.

In [48] a fully functioning IEEE 802.11 wifi modem is implemented using GNU Radio. This project demonstrates the capability of GNU Radio to implement existing communication standards completely in software allowing for easier testing to be done.

GNU Radio has improved its capabilities since inception. As a result, it supports orthogonal frequency-division multiplexing (OFDM) encoding. It also has the ability to include tag locations in data streams enabling other blocks to work on set length blocks instead of a continuous stream of data. GNU Radio also supports asynchronous execution of flowgraphs through message passing between blocks which opens the doors for implementing a number of communication protocols such as automatic repeat request (ARQ). It also allows for sending/receiving command messages to blocks, this enables blocks to send debug information to the user and enables the user to change the blocks behavior in real time during execution.

Another noteworthy project is the FOSPHOR Real Time Spectrum Analyser (RTSA) FFT display. This project provides a reference for GNU Radio blocks that target either a GPU via openCL/CUDA, or on some software radios, an FPGA.

For Ettus products with a large enough FPGA, such as the E310 or X series SDRs, some signal processing operations can be offloaded to the FPGA[1]. This allowing performance critical code, such as filtering or demodulation to be free of the performance degradation added by the operating system. For some examples in the literature see: [7], [49], [50].

V. Examples

In this section, we will present two examples which were built using by using and extending GNU Radio blocks. Throughout the examples, we will discuss the data models, roadblocks and solutions, and implementations.

A. Audio Streaming

This example displays the GNU Radio capabilities of fast prototyping by developing simulations using available library modules. Since the code is open-source this allows for developers to alter as needed in order to develop specialized systems. This example will showcase real-channel RF communication by transmitting and receiving live-audio with no perceptible degradation at the receiver.

1) Data Model: In Fig. 5, we present the data model. The data flows to the right. The data model refers to the Flow-graph implementation Fig. 6. Starting from the right on Fig. 5, we see the 'Access Code'. At this part of the data flow, an access code is created. This code is used in order to signal of incoming useful data. The access code is simply a token string defined by a macro in C++. This toke string is appended to the beginning of the message. The specifics of how this is done in code can be seen on the source code for the taggedAccessCode block [51]. Following the 'Access Code', is the 'Packet length' which is used by the receiver to properly decode the incoming data. If the packet length is out of sync, the reconstructed signal will include garbage or be completely unrecognizable. In order to maintain synchronization, a packet number is added to the message. At the receiver, we verify that the packet number is behaving in a nondecreasing manner for debugging purposes. If a packet is lost, the receiver will signal the console and continue to interpret the rest of the incoming packages. The packet type follows the packet number and was added as a way to distinguish between data sent and padding. The padding data is added as a means to flush out the transients within the software radio and are removed before message reconstruction on the receiver. The 'Payload' is the last packet sent. In this example, the data are float data-type representation of audio sampled directly at the audio card by using the 'Audio Source' block as seen in Fig 6.

[1]https://www.ettus.com/sdr-software/detail/rf-network-on-chip

2) Audio Streaming Challenges and Solutions:

- Interpolation and Filtering

  Care needs to be taken when designing the filters as it is *very* easy to make a filter with too many taps. The number of taps dictates how much stop-band attenuation on the signal. In general, more taps requires more signal processing. As a result, the number of taps is directly related to the perceived delay of the signal at the receiver end. The interpolation factor is another major design parameter in this system as higher interpolation rates improve audio quality but reduce the bit rate. Similarly decreasing the interpolation factor increases the bit rate while decreasing the audio quality. The interpolation blocks can be a considerable bottleneck due to the Root Raised Cosine (RRC) filters oversampling denoted as samples/symbol. Typically a good oversampling value for simulations in GNU Radio is two. However, high oversampling rates are needed when transmitting with hardware. Underruns were a major issue in our audio transmitter. An underrun happens when the transmitter sends data at a lower speed than the receiver is expecting. In other words, the receiver is not being fed enough data. A solution to this problem was to slightly oversample at the transmitter side with an interpolation to decimation factor of $\frac{124}{123}$. Using a channel model block to simulate noise can be a good indication of what oversampling is needed for your application.
- Understanding the Demodulator Block

  The demodulator block is known as a hierarchical block. It is a combination of several existing blocks. The blocks included in the demodulator are: Costas loop, frequency locked loop, phase locked loop, polyphase clock sync, symbol mapper. The output of this block are unpacked bytes, only one message bit per byte is sent from the demodulator.

3) Implementation and Evaluation: In Fig. 6, and 7, we showcase the flow-graph implementation of the transmitter and receiver respectively.

4) Evaluation: Filters were designed using both the GNU Radio filter design Graphical User Interface (GUI) which is very similar to Scipy's and MATLAB's windowed FIR design functions.

To stream audio between radios, some kind of encoding standard is typically used. GNU Radio provides implementations of several of these standards. However the documentation of the blocks is somewhat lacking, so a simple Pulse Coded Modulation (PCM) system is used in this example. The audio signal is sent into the flowgraph using the audio source block. This block will capture samples from an audio device, such as a sound card or microphone, at a specified sample rate. The choice of sample rate is limited to common audio sample rates for obvious reasons.

After sampling the audio card at 16 KHz, the rational resampler block is used to interpolate and decimate at a rate of 124/123, this odd re-sampling is done in order to ensure no under-runs happen at the receiver. The float to char block casts a float type to char, this loss of precision usually results in an unintended data representation. That is

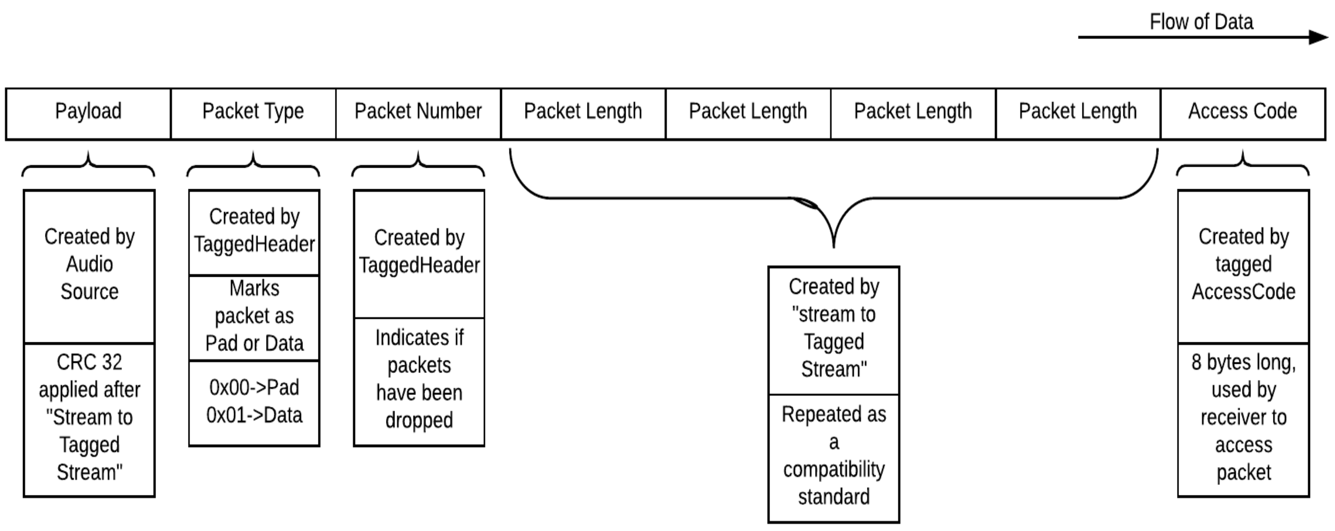
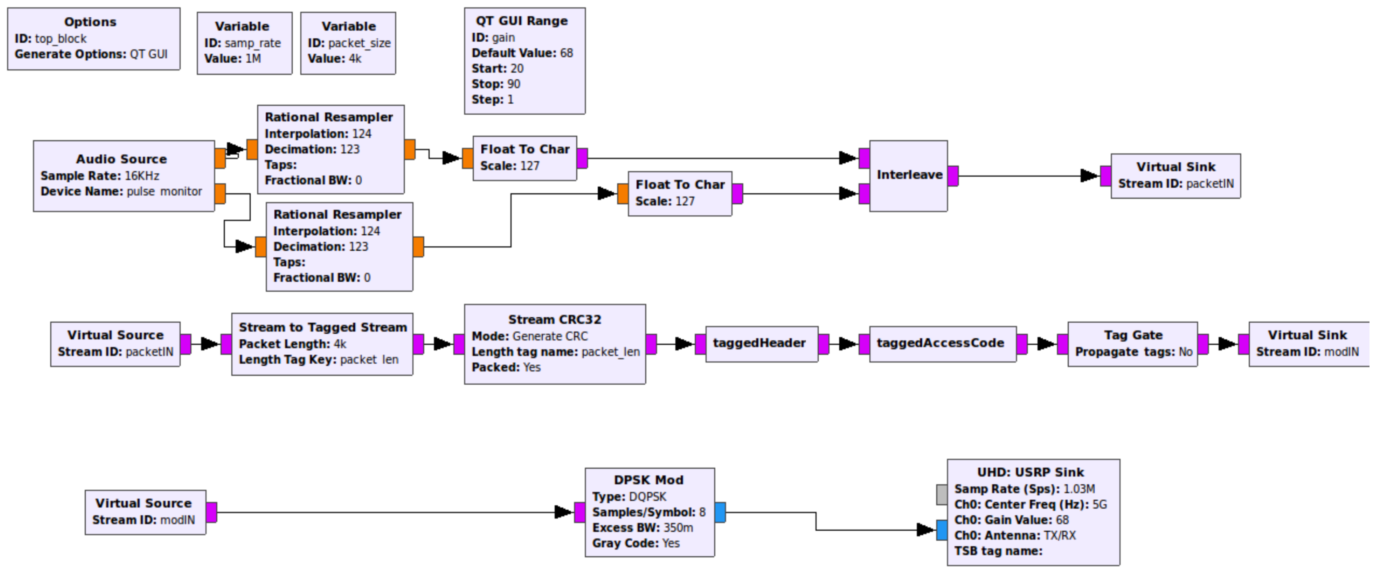

Flow of Data
Payload
Packet Type
Packet Number
Packet Length
Packet Length
Packet Length
Packet Length
Access Code
Created by Audio Source
CRC 32 applied after "Stream to Tagged Stream"
Created by TaggedHeader
Marks packet as Pad or Data
0x00->Pad 0x01->Data
Created by TaggedHeader
Indicates if packets have been dropped
Created by "stream to Tagged Stream"
Repeated as a compatibility standard
Created by tagged AccessCode
8 bytes long, used by receiver to access packet

Fig. 5. Data Model


Options
ID: top_block
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 1M

Variable
ID: packet_size
Value: 4k

QT GUI Range
ID: gain
Default Value: 68
Start: 20
Stop: 90
Step: 1

Audio Source
Sample Rate: 16KHz
Device Name: pulse_monitor

Rational Resampler
Interpolation: 124
Decimation: 123
Taps:
Fractional BW: 0

Rational Resampler
Interpolation: 124
Decimation: 123
Taps:
Fractional BW: 0

Float To Char
Scale: 127

Float To Char
Scale: 127

Interleave

Virtual Sink
Stream ID: packetIN

Virtual Source
Stream ID: packetIN

Stream to Tagged Stream
Packet Length: 4k
Length Tag Key: packet_len

Stream CRC32
Mode: Generate CRC
Length tag name: packet_len
Packed: Yes

taggedHeader

taggedAccessCode

Tag Gate
Propagate tags: No

Virtual Sink
Stream ID: modIN

Virtual Source
Stream ID: modIN

DPSK Mod
Type: DQPSK
Samples/Symbol: 8
Excess BW: 350m
Gray Code: Yes

UHD: USRP Sink
Samp Rate (Sps): 1.03M
Ch0: Center Freq (Hz): 5G
Ch0: Gain Value: 68
Ch0: Antenna: TX/RX
TSB tag name:

Fig. 6. Flow-graph implementation of the audio transmitter in the GRC.

why we set the scale to 127 which essentially multiplies each value by the aforementioned constant integer. The right and left audio samples are interleaved using the interleave block. This prepares to ultimately be modulated since the modulator interprets bytes at a time. We have introduced our own blocks between the interleave and modulator blocks which aid in signal reconstruction at the receiver as explained in section IV.A.1. The latest code implementation can be found at [2].

The modulation scheme used is a simple Differential Phase Shift Keying (DPSK). The documentation of both blocks is excellent, and one of the GNU Radio tutorials gives an excellent overview of the underlying signal processing concepts involved [3].

After demodulation, the bits are repacked to the full eight bits per byte. The left and right channels are then de-interleaved, cast to float and rescaled to $[-1, 1]$. The signal is then played through the laptop speakers using the GRC Audio Sink.

5) GNU Radio Implementation: Include plots at each stage with comments on performance. Also put the graphs here.

The GNU Radio Companion flowgraphs used in this example. The only thing not currently covered is the resampler block in the rx flowgraph. This is done because the USRP will capture samples at 250000 Hz, I didn't want to redesign the filter and its generally better to distribute decimating operations across many stages instead of doing everything in one go. The first resampler resamples the signal to about 88600 Hz, this is close enough to 88200 Hz that the rest of the graph

[2]$https://github.com/samGNSS/NSF_IRES_Tutorial/tree/master/code_dir$

[3]$https://wiki.gnuradio.org/index.php/Guided_Tutorial_PSK_Demodulation$

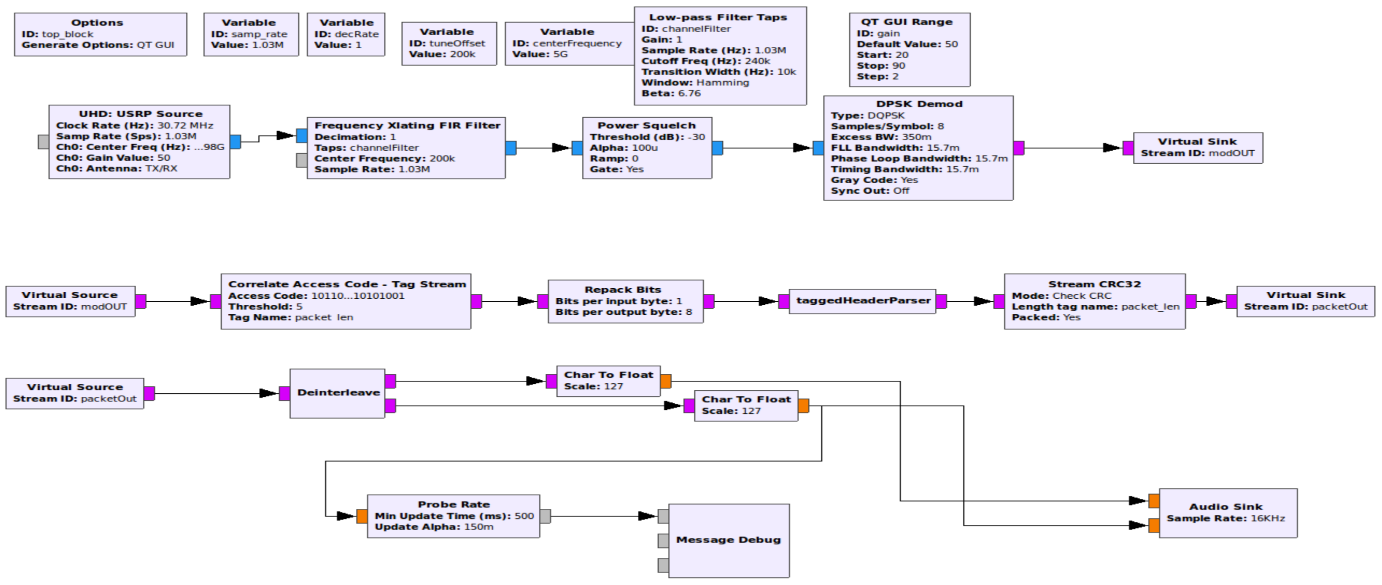


Fig. 7. Flow-graph implementation of the audio receiver in the GRC.

works fine. Take care that the anti-aliasing filter used in the resampler block does not have too many taps. If you wish to design your own filter, the block upsamples before filtering.

B. *Packeted Image Transmission*

This example provides an easy and customizable packetization framework using GNU Radio and written in C++. The current state of GNU Radio does not explicitly have this ability as the packetization blocks were written almost exclusively for OFDM.

Packetization is very important in modern communication schemes. It essentially gives structure to a stream of bits which allows for error detection/correction, passage of metadata such as the size of the packet and the number of the packet in the stream, and eases synchronization of bits at the receiver.

1) System Model: The packet layer implementation given in this tutorial is simple and fairly standard. The packet contains an arbitrarily sized payload, a four byte Cyclic Redunacy check (CRC), a four byte header containing the payload size as two bytes repeated, and finally an eight byte access code.

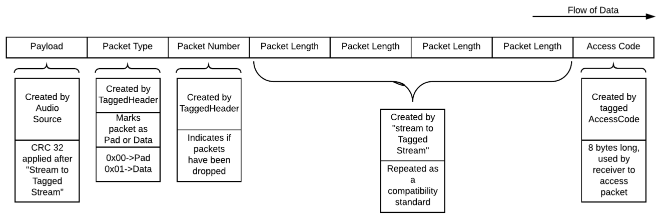

Flow of Data
Payload
Packet Type
Packet Number
Packet Length
Packet Length
Packet Length
Packet Length
Access Code
Created by Audio Source
CRC 32 applied after "Stream to Tagged Stream"
Created by TaggedHeader
Marks packet as Pad or Data
0x00->Pad 0x01->Data
Created by TaggedHeader
Indicates if packets have been dropped
Created by "stream to Tagged Stream"
Repeated as a compatibility standard
Created by tagged AccessCode
8 bytes long, used by receiver to access packet

Fig. 8. Packet

2) Challenges and Solutions:

3) GNU Radio Implementation: In this example, we wrote some code[4] that implements a basic packeted communication scheme. This example makes use of tagged streams in GNU Radio. Prior to tagged streams, GNU Radio was only able to stream data between blocks with no information passed between them. Thus making a physical communication layer extremely difficult to implement.

Add Implementation here when I figure out single page picture

4) Evaluation: Using a raw image transmission, the system is able to function at a noise voltage of 0.5 during simulation but any higher and the bit errors/packet losses become noticeable. This can be seen in Fig. **??**. Due to packet losses the image is completely destroyed and the trailing zeros from the file source are now very noticeable. Error correction can be easily added to the system for increased robustness. The Forward Error Correction (FEC) modules in GNU Radio are lacking documentation but are generally easy to figure out. Using the LDPC encoder and decoder generators as an example. The user defines the generator matrix in systematic form using an alist file[5]. Each row in the a list file denotes the ones location in a column of the generator. Basically alist files are a generic way of working with sparse matrices. The FEC encoders and decoders also support tagging streams.

Tagged streams are normal GNU Radio streams, but with metadata passed in parallel to other blocks. The metadata stores an *absolute*[6] offset value for a stream and marks that location as important. It is trivial to see how this allows for a PHY layer to be implemented. The tagged streams also will not pause flowgraph execution while waiting for samples as some blocks will.

Alternatively, GNU Radio also provides an easy interface for message passing with PDU blocks. These differ from tagged streams, in that the execution of the blocks is asynchronous. Messages are pushed to a global queue and are

[4]$https://github.com/samGNSS/NSF_IRES_Tutorial/tree/master/audio$

[5]http://www.inference.phy.cam.ac.uk/mackay/codes/alist.html

[6]Offsets are calculated based on the total number of samples processed by that block

processed as time allows. The arrival time of any message to any block is *not* guaranteed. While this may not sound ideal, PDU's are excellent for bursty communications. When ever a burst is detected, the samples can be put into a PDU and sent downstream for processing while other parts of the flowgraph are executed. GNU Radio provides an excellent introduction to PDU's and tagged streams in their OFDM example[7].

During real-channel transmission, the received image is perfect but slightly bigger file size than the original image. Transmitted file size is 473.8 kB while the received file size is 474.6 kB. It's worth noting that the distance between transmitter and receiver was 1 meter. The image format choice was JPG as it allowed for partial transmission during the initial testing due to its data structure.

VI. Common Pitfalls and Solutions

In this section, we will discuss the problems that arose at different stages of development of the working examples and their possible solutions. Sometimes it is more desirable to install GNU Radio from source. The reasons for installing from source might differ depending on the user. One common reason is to have the option to install a previous version of GNU Radio that is more compatible with the user's specific installed Linux distribution or the application being built. For our examples, we used the USRP B210 edition together with a Linux machine running Ubuntu long-term support (LTS) 16.04 since it is the most compatible version with GNU Radio. The main issues most people encounter happen during the installation and setup. We will also discuss some general problems one might encounter due to the communication interface used between the software radio and host.

A. Installation and Setup

The steps to achieve installation from source can be found at [52]. The two most common issues after installing from source are the PYTHONPATH and the "old build" problems. The PYTHONPATH error appears when trying to execute GNU Radio Companion from within the terminal. The terminal will display a "PYTHONPATH" error. The solution to this problem is to explicitly declare the PYTHONPATH variable in your $HOME/.bashrc by writing `export PYTHONPATH=/usr/local/lib/python2.7/dist-packages`. GNU Radio does not do well when a machine has a newer version installed on top of an older one. This gives rise to many issues which we bundled together under the "old build" problem-tag. To fix the plethora of issues arising from installing a newer GNU Radio version on top of an old one, it is best to completely uninstall the software. To do this, travel to the build directory within the terminal and `$sudo make uninstall`
After the previous command runs, execute `$ git clean -d -x -f`.

During the setup, it is recommended to test the hardware equipment to avoid possible problems. Testing the software radio and antenna are recommendations made by the manufacturer. In reality, a third piece of equipment, the laptop or PC, should be tested. But in general the computer is only tested if issues with the USB or method of communication used for interfacing are detected. If the hardware is defective, the user will encounter issues in signal reliability when transferring information or the system will not work at all. In order to test the antenna one should look at the spreadsheet given by the manufacturer to confirm optimal frequency operation. It is advisable to work on the recommended frequencies given by the antenna manufacturer. To test the antenna, a network analyzer should be used to verify that the element's reflection parameter coincides with the specified frequencies in the manufacturer's spreadsheet. The main test for the software radio board should be to verify that when GNU Radio tells the board to emit a carrier signal at a specified frequency, we should see a power peak at this frequency using a spectrum analyzer. To test this, simply connect software radio to a SubMiniature version A (SMA) cable and the (SMA) to a Bayonet NeillConcelman (BNC) connector to the spectrum analyzer. When the GNU Radio program is running, one should see a peak at the set frequency.

B. Communication Interface Problems

A common set of problems arises from the interface needed to transfer information between the software radio and the host computer. The 2 types of communication interfaces used are USB and network. The USRP B-series uses the USB communication standard. The USRP X300/X310, N200/N210, and USRP2 are networked devices that work on the 16Gbit/10Gbit Ethernet ports. There is a third type of software radio which acts as a stand-alone Linux SDR device. These software radios run the GNU Radio compiled software themselves without the need of an external computer to run the code. Some such examples of stand-alone software radios are the E100, E110, E310 or E312 USRP series.

If there is a communication problem between the software radio and the host computer, the terminal will output an error related to "USRP no devices found". As a general rule of thumb, it is recommended to check the functionality of the UHD through the terminal with `$uhd_find_devices`. The output will indicate which UHD driver version is installed. Most problems are fixed by following the steps in [52] and ensuring the latest UHD driver is installed.

A networked software radio depends on the host's network interface controller to be 1Gbit capable. It is also important to note that USB3-to-Gigabit adapters do not work well with network SDRs due to being unable to keep up with the high-rate samples of real time applications.

VII. Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Integer nec odio. Praesent libero. Sed cursus ante dapibus diam. Sed nisi. Nulla quis sem at nibh elementum imperdiet. Duis sagittis ipsum. Praesent mauris. Fusce nec tellus sed augue semper porta. Mauris massa. Vestibulum lacinia arcu eget nulla. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Curabitur sodales ligula in libero. Sed dignissim lacinia nunc. Curabitur tortor.

[7]get teh cites

Pellentesque nibh. Aenean quam. In scelerisque sem at dolor. Maecenas mattis. Sed convallis tristique sem. Proin ut ligula vel nunc egestas porttitor. Morbi lectus risus, iaculis vel, suscipit quis, luctus non, massa. Fusce ac turpis quis ligula lacinia aliquet. Mauris ipsum. Nulla metus metus, ullamcorper vel, tincidunt sed, euismod in, nibh.

References

[1] J. Mitola, "Software radios-survey, critical evaluation and future directions," in *[Proceedings] NTC-92: National Telesystems Conference*, May 1992, pp. 13/15–13/23.

[2] S. Soltani, Y. Sagduyu, Y. Shi, J. Li, J. Feldman, and J. Matyjas, "Distributed cognitive radio network architecture, sdr implementation and emulation testbed," in *MILCOM 2015 - 2015 IEEE Military Communications Conference*, Oct 2015, pp. 438–443.

[3] E. Mioso, M. Bonomi, F. Granelli, and C. Sacchi, "An sdr-based reconfigurable multicarrier transceiver for terrestrial and satellite communications," in *2017 IEEE Aerospace Conference*, March 2017, pp. 1–13.

[4] nVidia Corporation, "Whitepaper nvidia sdr (software defined radio) technology the modem innovation inside nvidia i500 and tegra 4i," 2788 San Tomas Expressway Santa Clara, CA 95051, Tech. Rep. Version 3, Final, January 2018. [Online]. Available: http://www.nvidia.com/docs/IO//116757/NVIDIA_i500_whitepaper_FINALv3.pdf

[5] Wikipedia, "List of software-defined radios — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=List%20of%20software-defined%20radios&oldid=771825669, 2017, [Online; accessed 31-March-2017].

[6] M. A. Wickert and M. R. Lovejoy, "Hands-on software defined radio experiments with the low-cost rtl-sdr dongle," in *2015 IEEE Signal Processing and Signal Processing Education Workshop (SP/SPE)*, Aug 2015, pp. 65–70.

[7] E. Grayver, H. S. Green, and J. L. Roberson, "Sdrphy - xml description for sdr physical layer," in *MILITARY COMMUNICATIONS CONFERENCE, 2010 - MILCOM 2010*, Oct 2010, pp. 1140–1146.

[8] S. Vajdic and F. Jiang, "A hands-on approach to the teaching of electronic communications using gnu radio companion and the universal software radio peripheral," in *2016 IEEE Integrated STEM Education Conference (ISEC)*, March 2016, pp. 19–21.

[9] P. Sutton and I. Gomez, "Open source 3gpp lte library," https://https://github.com/srsLTE/srsLTE, 2017.

[10] M. Robert, Y. Sun, T. Goodwin, H. Turner, J. H. Reed, and J. White, "Software frameworks for sdr," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 452–475, March 2015.

[11] S.-L. Tsao, C.-C. Lin, C.-L. Chiu, H.-L. Chou, and M.-C. Wang, "Design and implementation of software framework for software defined radio system," in *Proceedings IEEE 56th Vehicular Technology Conference*, vol. 4, 2002, pp. 2395–2399 vol.4.

[12] M. Bazdresch, "Considerations for the design of a hands-on wireless communications graduate course based on software-defined radio," in *2016 IEEE Frontiers in Education Conference (FIE)*, Oct 2016, pp. 1–5.

[13] A. V. Oppenheim and R. W. Schafer, *Principles of Communication Systems*. Wiley.

[14] R. E. Ziemer and W. H. Tranter, *Discrete-Time Signal Processing*, 7th ed. Pearson, 2014.

[15] GNU Radio, "GNU Radio stuff," millions. [Online]. Available: http://gnuradio.org/

[16] F. Harris and W. Lowdermilk, "Software defined radio: Part 22 in a series of tutorials on instrumentation and measurement," *IEEE Instrumentation Measurement Magazine*, vol. 13, no. 1, pp. 23–32, February 2010.

[17] J. Mitola, "The software radio architecture," *IEEE Communications Magazine*, vol. 33, no. 5, pp. 26–38, May 1995.

[18] M. Cummings and T. Cooklev, "Tutorial: Software-defined radio technology," in *2007 25th International Conference on Computer Design*, Oct 2007, pp. 103–104.

[19] L. Zhang, "Implementation of wireless communication based on software defined radio," July 2013. [Online]. Available: http://oa.upm.es/21618/

[20] Wikipedia, "List of software-defined radios," 2018. [Online]. Available: https://en.wikipedia.org/wiki/List_of_software-defined_radios

[21] RTL-SDR, "RTL-SDR overview," 2014. [Online]. Available: http://sdr.osmocom.org/trac/wiki/rtl-sdr

[22] Air Spy, "Air Spy," 2016. [Online]. Available: http://airspy.com/

[23] Y. L. Sit, B. Nuss, S. Basak, M. Orzol, W. Wiesbeck, and T. Zwick, "Real-time 2d+velocity localization measurement of a simultaneous-transmit ofdm mimo radar using software defined radios," in *2016 European Radar Conference (EuRAD)*, Oct 2016, pp. 21–24.

[24] RTL-SDR, "RTL-SDR news," 2016. [Online]. Available: http://www.rtl-sdr.com

[25] B. Uengtrakul and D. Bunnjaweht, "A cost efficient software defined radio receiver for demonstrating concepts in communication and signal processing using python and rtl-sdr," in *Digital Information and Communication Technology and it's Applications (DICTAP), 2014 Fourth International Conference on*, May 2014, pp. 394–399.

[26] "Removing that center frequency dc spike in gnuradio the easy way," Feb 2017. [Online]. Available: https://www.rtl-sdr.com/removing-that-center-frequency-dc-spike-in-gnuradio-the-easy-way/

[27] "How to calibrate rtl-sdr using kalibrate-rtl on linux," Dec 2015. [Online]. Available: https://www.rtl-sdr.com/how-to-calibrate-rtl-sdr-using-kalibrate-rtl-on-linux/

[28] Hack RF, "Hack RF Documentation," millions. [Online]. Available: https://github.com/mossmann/hackrf/wiki

[29] Blade RF, "Blade RF Overview," millions. [Online]. Available: http://www.nuand.com/

[30] Analog Devices, "FMCOMMS2 Overview," millions. [Online]. Available: http://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/EVAL-AD-FMCOMMS2.html#eb-overview

[31] Ettus Research, "Ettus BUS series," millions. [Online]. Available: https://www.ettus.com/product/category/USRP-Bus-Series

[32] ——, "Ettus Networked series," millions. [Online]. Available: https://www.ettus.com/product/category/USRP-Networked-Series

[33] ——, "Ettus Embedded series," millions. [Online]. Available: https://www.ettus.com/product/category/USRP-Embedded-Series

[34] Analog Devices, "Data Sheet AD9361," 2016. [Online]. Available: http://www.analog.com/media/en/technical-documentation/data-sheets/AD9361.pdf

[35] W. L. I. Agency, "Product category." [Online]. Available: https://www.ettus.com/product/category/USRP-X-Series

[36] "Download." [Online]. Available: https://airspy.com/download/

[37] A. A. Tabassam, F. A. Ali, S. Kalsait, and M. U. Suleman, "Building software-defined radios in matlab simulink - a step towards cognitive radios," in *Computer Modelling and Simulation (UKSim), 2011 UkSim 13th International Conference on*, March 2011, pp. 492–497.

[38] "Using ni labview and ni pxi for cognitive radio prototyping." [Online]. Available: http://sine.ni.com/cs/app/doc/p/id/cs-14843

[39] "Explaining the gnu radio scheduler." [Online]. Available: http://www.trondeau.com/blog/2013/9/15/explaining-the-gnu-radio-scheduler.html

[40] "Pybombs the what, the how and the why." [Online]. Available: https://www.gnuradio.org/blog/pybombs-the-what-the-how-and-the-why/

[41] "Windowsinstall." [Online]. Available: https://wiki.gnuradio.org/index.php/WindowsInstall

[42] "Osx install." [Online]. Available: https://wiki.gnuradio.org/index.php/MacInstall

[43] GNSS SDR, "GNSS SDR," millions. [Online]. Available: http://gnss-sdr.org/

[44] C. Fernndez-Prades, "The control plane," Apr 2016. [Online]. Available: http://gnss-sdr.org/docs/control-plane/

[45] ——, "Signal processing blocks." [Online]. Available: http://gnss-sdr.org/docs/sp-blocks/

[46] "Welcome to python.org." [Online]. Available: https://www.python.org/

[47] "Numpy and scipy documentation." [Online]. Available: https://docs.scipy.org/doc/

[48] people, "IEEE 802.11 implementation in GNU radio," millions. [Online]. Available: https://github.com/bastibl/gr-ieee802-11

[49] S. Heunis, Y. Paichard, and M. Inggs, "Passive radar using a software-defined radio platform and opensource software tools," in *2011 IEEE RadarCon (RADAR)*, May 2011, pp. 879–884.

[50] J. M. Friedt, "Gnuradio as a digital signal processing environment: Application to acoustic wireless sensor measurement and time amp; frequency analysis of periodic signals," in *European Frequency and Time Forum International Frequency Control Symposium (EFTF/IFC), 2013 Joint*, July 2013, pp. 278–281.

[51] samGNSS, "samgnss/nsf_ires_tutorial." [Online]. Available: https://github.com/samGNSS/NSF_IRES_Tutorial/blob/master/code/lib/taggedAccessCode_impl.cc

[52] [Online]. Available: https://kb.ettus.com/Building_and_Installing_the_USRP_Open-Source_Toolchain_(UHD_and_GNU_Radio)_on_Linux