Lab 1: Introduction to the SDR

Dennis Joosens[1,*] **and Maarten Weyn**[1]

[1]Electronics-ICT, Faculty of Applied Engineering, University of Antwerp, Belgium
[*]corresponding. (dennis.joosens@uantwerpen.be)

ABSTRACT

A Software Defined Radio (SDR) is a communication radio that uses software to implement the algorithms necessary for digital communication. In this lab, you will be designing and implementing an SDR using Universal Software Radio Peripheral (USRP) hardware and the GNU Radio software.
The purpose of this introductory laboratory exercise is to ensure that the students have a working installation of GNU Radio on their computers and know how to connect to the USRP software defined radio.

1 Introduction

The Wireless Innovation Forum (WINNF) defines Software Defined Radio as: *"Radio in which some or all of the physical layer functions are software defined"*. SDR refers to the technology wherein software modules running on a generic hardware platform are used to implement radio functions. By combining the USRP hardware with the GNU Radio software you can create a flexible and functional SDR platform for rapid prototyping of wireless signals including physical layer design, record and playback, signal intelligence, algorithm validation, and more.

1.1 *USRP Hardware*

The Universal Software Radio Peripheral (USRP) software defined radio products are designed for RF applications from DC to 6 GHz, including multiple antenna (MIMO) systems. Example application areas include white spaces, mobile phones, public safety, spectrum monitoring, radio networking, cognitive radio, satellite navigation, and amateur radio. The USRP connects to a host PC creating a software defined radio.

For receiving:

- The incoming signals at the SMA connector inputs are mixed down using a direct-conversion receiver to baseband I/Q components.
- The I/Q data will be sampled by an analog-to-digital converter (ADC).
- The digitized I/Q data follows parallel paths through a digital down-conversion (DDC) process that mixes, filters, and decimates the input signal to a user-specified rate.
- The down-converted samples are passed to the host computer.

For transmission:

- Baseband I/Q signal samples are synthesized by the host computer and fed to the USRP at a specified sample rate over USB.
- The USRP hardware interpolates the incoming signal to a higher sampling rate using a digital up-conversion (DUC) process.
- Then converts the signal to analog with a digital-to-analog converter (DAC).
- The resulting analog signal is then mixed up to the specified carrier frequency and transmit via the SMA connectors.

The transmitting or receiving operations are summarized in Figure (1).

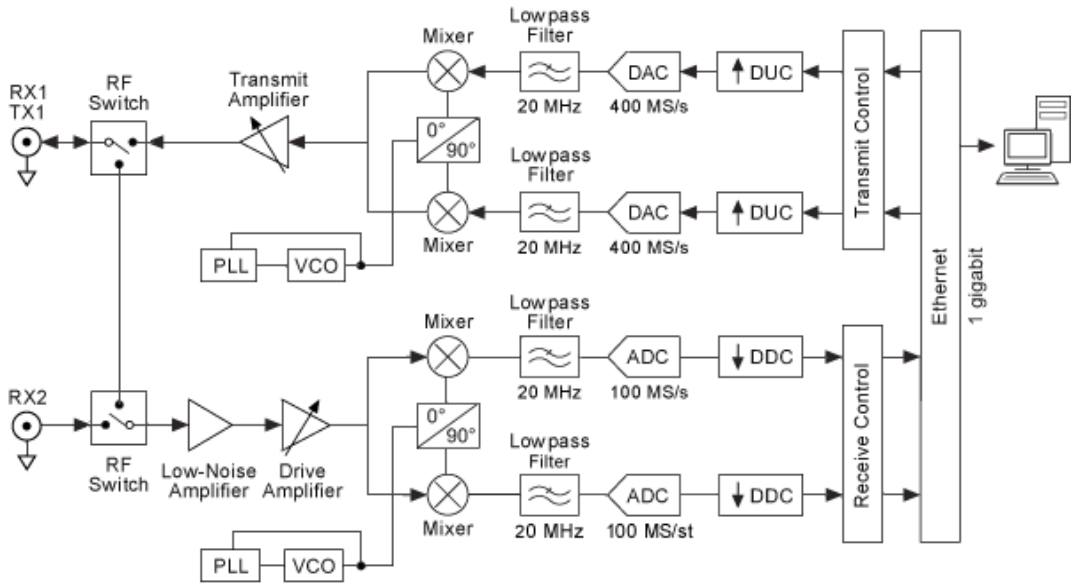

Figure 1. Typical Block Diagram of a USRP

In our lab, we will use the USRP B200 and B210 hardware, see Figures (2), (3) and (4). The B(us) series USRP covers RF frequencies from 70 MHz to 6 GHz, has a Spartan-6 FPGA, and USB 3.0 connectivity. This platform enables experimentation with a wide range of signals including FM and TV broadcast, cellular, Wi-Fi, and more. The USRP B200 features one receive and one transmit channel in a bus-powered design. The USRP B210 extends the capabilities of the B200 by offering a total of two receive and two transmit channels, incorporates a larger FPGA, GPIO, and includes an external power supply. Both use an Analog Devices RFIC (AD9361) to deliver a cost-effective RF experimentation platform, and can stream up to 56 MHz of instantaneous bandwidth over a high bandwidth USB 3.0 bus on select USB 3.0 chipsets (with backward compatibly to USB 2.0).

Figure 2. Front View of a USRP B210 Software Defined Radio

Figure 3. Front View of a USRP B2x0 Software Defined Radio

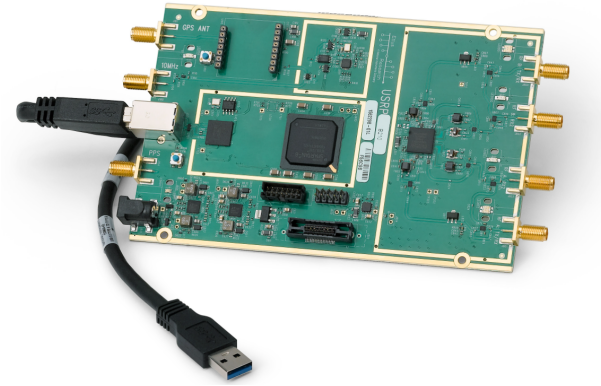

Figure 4. PCB of a USRP B210 Software Defined Radio

1.2 *GNU Radio Software*

GNU Radio[1][2] is a free & open-source software development toolkit that provides signal processing blocks to implement software radios. It can be used with readily-available low-cost external RF hardware to create software-defined radios, or without hardware in a simulation-like environment. It is widely used by hobbyist, in academic and commercial environments to support both wireless communications research and real-world radio systems.

GNU Radio performs all the signal processing. You can use it to write applications to receive data out of digital streams or to push data into digital streams, which is then transmitted using hardware. GNU Radio has filters, channel codes, synchronisation elements, equalizers, demodulators, decoders, and many other elements (called blocks) which are typically found in radio systems. More importantly, it includes a method of connecting these blocks and then manages how data is passed from one block to another. Extending GNU Radio is also quite easy; if you find a specific block that is missing, you can quickly create and add it. These are call out-of-tree modules and can be built by writing Python code.

Since GNU Radio is software, it can only handle digital data. Usually, complex baseband samples are the input data type for receivers and the output data type for transmitters. Analog hardware is then used to shift the signal to the desired center frequency. That requirement aside, any data type can be passed from one block to another - be it bits, bytes, vectors, bursts or more complex data types.

GNU Radio applications are primarily written using the Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ using processor floating point extensions, where available. Thus, the developer is able to implement real-time, high-throughput radio systems in a simple-to-use, rapid-application-development environment.
GNU Radio is a framework that enables users to design, simulate, and deploy highly capable real-world radio systems. It is a highly modular, ”flowgraph”-oriented framework that comes with a comprehensive library of processing blocks that can be readily combined to make complex signal processing applications.

[1] https://wiki.gnuradio.org
[2] https://www.gnuradio.org/

2 IQ Data

The in-phase/quadrature (I/Q) data is a complex representation of the transmitted signal, for that it can not only give information about the amplitude of the signals but also the phase information. The IQ data can be represented as two channels of data that are orthogonal with each other as shown in Figure (5).

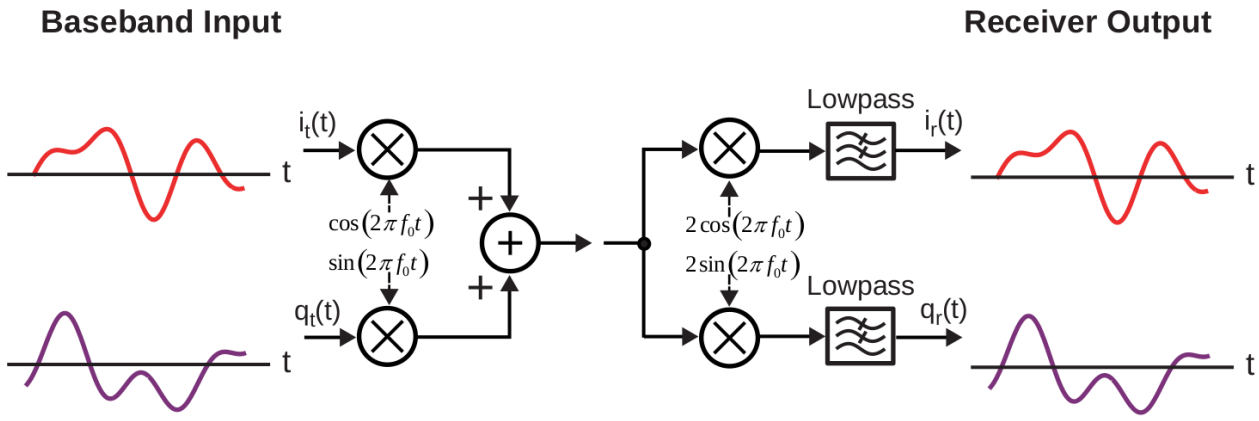

Figure 5. In-phase and Quadrature channels

The two channels at the transmitter side can be written as follows:

$$\begin{aligned} i(t) &= i_t(t)\cos(2\pi f_o t) \\ q(t) &= q_t(t)\sin(2\pi f_o t) \end{aligned} \tag{1}$$

which leads to,

$$y(t) = i_t(t)\cos(2\pi f_o t) + q_t(t)\sin(2\pi f_o t) \tag{2}$$

at the receiver side the two channels I and Q can be recovered using the mathematical product-to-sum identity. [3]

$$\begin{aligned} i_r(t) &= [i_t(t)\cos(2\pi f_o t) + q_t(t)\sin(2\pi f_o t)]2\cos(2\pi f_o t) \\ &= [i_t(t) + i_t(t)\cos(4\pi f_o t)] + [q_t(t)\sin(4\pi f_o t) + q_t(t)\sin(0)] \\ &= [i_t(t) + i_t(t)\cos(4\pi f_o t)] + [q_t(t)\sin(4\pi f_o t)] \end{aligned} \tag{3}$$

$$\begin{aligned} q_r(t) &= [i_t(t)\cos(2\pi f_o t) + q_t(t)\sin(2\pi f_o t)]2\sin(2\pi f_o t) \\ &= [i_t(t)\sin(4\pi f_o t) - i_t(t)\sin(0)] + [q_t(t) - q_t(t)\cos(4\pi f_o t)] \\ &= [i_t(t)\sin(4\pi f_o t)] + [q_t(t) - q_t(t)\cos(4\pi f_o t)] \end{aligned} \tag{4}$$

by using a low pass filter the I and Q channels can be recovered at the receiver.

[3] using the mathematical product-to-sum identities
$\cos(x)\times\cos(y) = \frac{1}{2}(\cos(x-y)+\cos(x+y))$
$\sin(x)\times\sin(y) = \frac{1}{2}(\cos(x-y)-\cos(x+y))$
$\sin(x)\times\cos(y) = \frac{1}{2}(\sin(x+y)+\sin(x-y))$
$\cos(x)\times\sin(y) = \frac{1}{2}(\sin(x+y)-\sin(x-y))$

3 Lab Procedure

1. Creating a flowgraph
   During this exercise you will generate a cosine wave using GNU Radio.

   - Open a new flowgraph in GNU Radio.
   - Two boxes will be in the flowgraph, first ”options” box, which is used to set global parameters for the flowgraph. The second one is ”variable”, which you can give it a name and use it in your flowgraph (default variable is the sample rate[4]).
   - Set Generate Options to QT GUI.
   - Add another ***Variable*** block, call it **freq** and set it to **1000**.
   - Add a ***Signal Source*** and set the **Output Type** to float numbers, **Waveform** to **cosine**, **Frequency** to freq, **Amplitude** to 1 and finally **Offset** to **0**.
   - Add a ***Throttle*** and set **Type** to **Float**.
   - Add a ***QT GUI Sink*** and set ”Type” to Float and ”FFT Size” to 1024.
   - Change the frequency of the Signal Source from 1k to 2.5k, 5k, 16k and 100k and observe the time and frequency domain.
   - **Question 1:** What is the difference and why?

   Eventually, you should have a flowgraph as shown in Figure(6) and when you run the flowgraph you will get a result as shown in Figure (7).

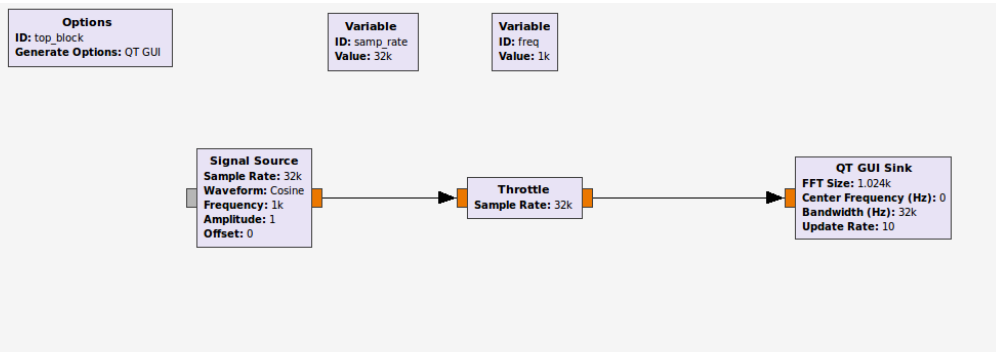

Figure 6. signal source GRC

[4]Think of the sample rate as being used to calculate the discrete step size from one sample to the next within a DSP operation. Sample rate also refers to the rate at which samples pass through the flowgraph. If there is no rate control, hardware clock or throttling mechanism, the samples will be generated, pass through the flowgraph and be consumed as fast as possible (i.e. the flowgraph will be CPU bound). Only a block that represents some underlying hardware with its own clock (e.g. USRP, sound card), or the Throttle Block, will use ”Sample Rate” to set that hardware clock, and therefore have the effect of applying rate control to the samples in the flowgraph.

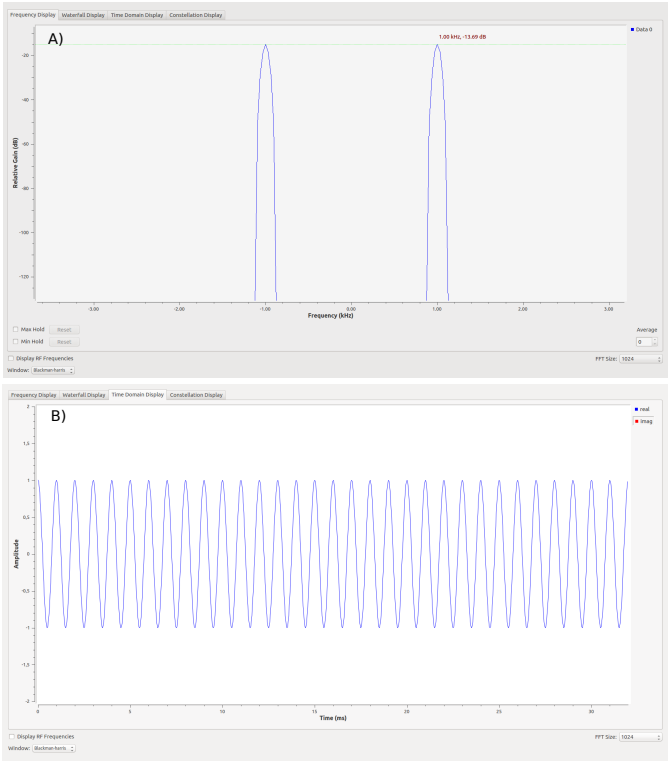

Figure 7. A) The frequency domain of the signal , B) The time domain of the signal

2. Complex signal with noise
   In this exercise, you will generate a complex cosine wave contaminated with noise.

   - Open a new flowgraph in GNU Radio.
   - Set **Generate Options** to **QT GUI**.
   - Set **samp_ rate** variable to **64000**.
   - Add another ***Variable*** block, call it **freq** and set it to **2000**.
   - Add a ***Signal Source*** and set the **Output Type** to **complex** numbers, **Waveform** to **Cosine**, **Frequency** to **freq**, **Amplitude** to **1** and finally **Offset** to **0**.
   - Add a ***Noise Source*** and set the **Output Type** to **complex** numbers, **Noise Type** to **Gaussian** and **Amplitude** to **0.1**.
   - Add an ***Add*** block.
   - Add a ***Throttle*** block and set **Type** to **Complex**.
   - Add a ***QT GUI Sink*** and set **Type** to **Complex** and **FFT Size** to **1024**.
   - **Question 2:** How many frequency peaks you can see, and why?
   - **Question 3:** How can you mitigate the noise effect?

At the end you should have a flowgraph as shown in Figure(8) and when you run the flowgraph you will get a results as shown in Figure (9).

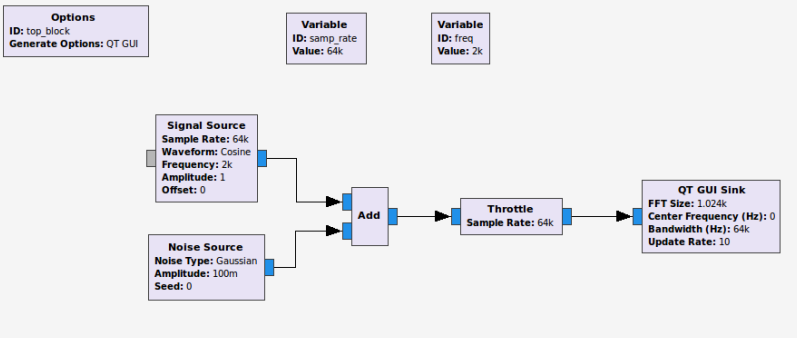

Figure 8. Complex signal with noise GRC

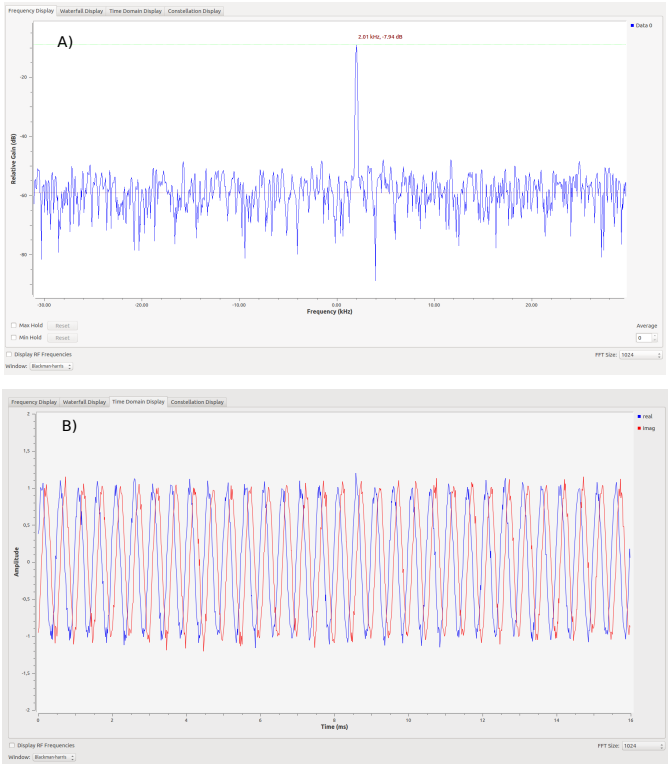

Figure 9. A) The frequency domain of the signal , B) The time domain of the signal

3. Connecting the USRP
   This exercise is to transmit and receive a complex cosine wave using the USRP.

   - Open a new flowgraph in GNU Radio.
   - Set **Generate Options** to **QT GUI**.
   - Set **samp_ rate** variable to **1e6**.
   - Add another ***Variable*** block, call it **freq** and set it to **20e3**.
   - Add another ***Variable*** block, call it **carrier_freq** and set it to **1e9**.
   - Add a ***Signal Source*** and set the **Output Type** to **complex numbers**, **Waveform** to **Cosine**, **Frequency** to **freq**, **Amplitude** to **1** and finally **Offset** to **0**.
   - Add a ***UHD: USRP sink*** and set the **Input Type** to **complex float32**, **center Freq** to **carrier_freq** and **Gain Value** to **1**.
   - Add a ***UHD: USRP Source*** and set the **Output Type** to **complex float32**, **center Freq** to **carrier_freq** and **Gain Value** to **0**.
   - Add ***QT GUI Sink*** and set **Type** to **Complex** and **FFT Size** to **1024**.
   - Add a filter to get the right signal with less noise.
   - **Question 4**: Which filter do you use and what are the parameters you will choose?

   At the end you should have a flowgraph as shown in figure(10).

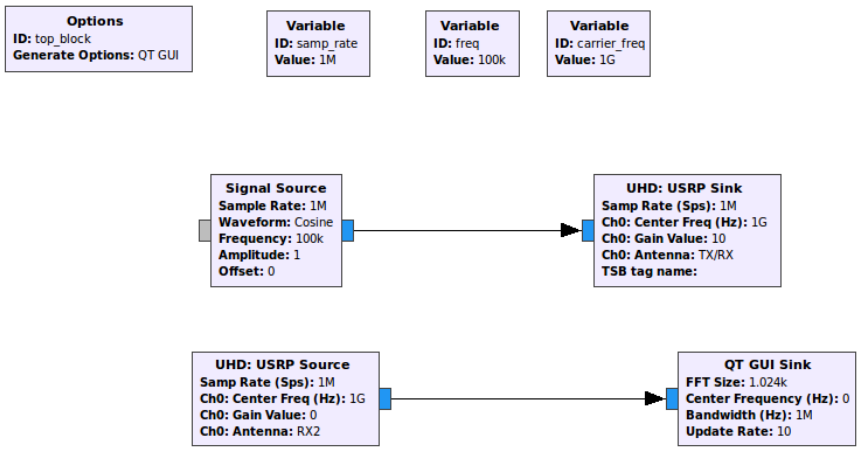

Figure 10. Signal source GRC