Lab 2 GNURadio Implementation

3 USRP Hardware Implementation

In Laboratory 1, you have already observed a digital communication system employing differential binary phase-shift keying (DBPSK) with a packet based framework that took care of the timing synchronization for you. In this lab, we will revisit the system and begin working towards developing our own packet-like framework.

3.1 Differential Binary Phase Shift Keying

Differential phase shift keying is a non-coherent form of phase shift keying which avoids the need for a coherent reference signal at the receiver. Non-coherent receivers are relatively easy and cheap to build, and hence are widely used in wireless communications.

3.1.1 Simulation

Download the file dbpsk_simulation.grc from the website and run it. The simulation is very straightforward – see Figure 1 below.

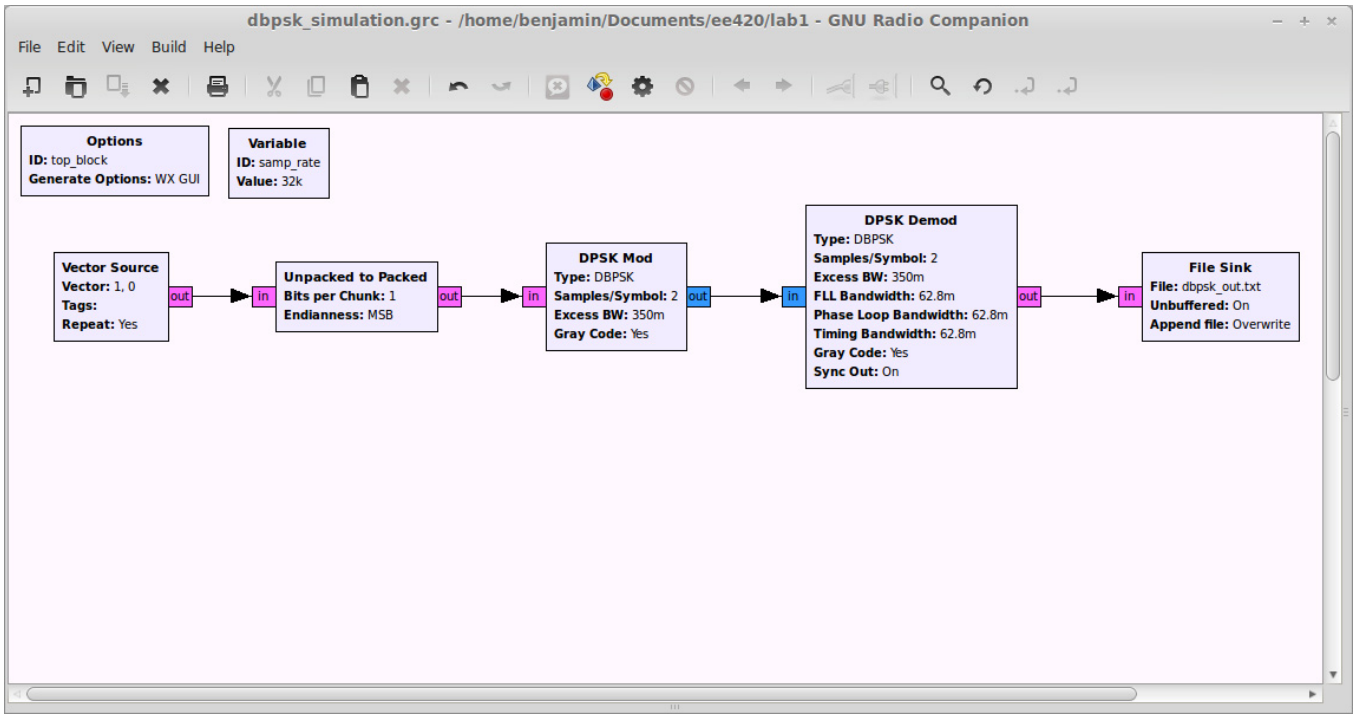


Figure 1: DBPSK Simulation.

The purpose of the blocks and their important parameters are as follows (for the other parameters you can probably just leave the default values):

- **Vector Source:** Generates vectors of data
    - Output Type: Byte – the DBPSK modulator is expecting byte inputs.
    - Vector: Input vector (interpreted as a list of integers, then truncated to bytes when output) – in this case we use the vector [1, 0] because it is easy to observe.
    - Repeat: Setting this to Yes (or True in python) will cause the block to constantly output repeated copies of the input vector. Setting it to No (or False in python) outputs only one copy of the vector.
    - Vec Length: Leave a 1 – this defines the vector length of the output. Setting it to 1 outputs a stream of individual bytes. *Note: the output vector length need NOT be the same as the Vector field above – i.e. [0,1] is the same as [0,1,0,1] when repeat is on and Vec Length is 1.*
- **Unpacked to Packed:** Converts a stream of unpacked bytes to packed bytes
    - Bits per Chunk: The number of bits to take from each input byte – set to 1 in this case because our input is only 0's or 1's.
- **DPSK Mod:** This block expects the input to be packed bytes (i.e. 'A' = 01000001 in binary). It then breaks each byte apart into individual bits which it then modulates and outputs the complex (baseband) representation of
    - Type: DBSPK – this is the type of modulation we want.
    - Samples/Symbol: 2 – this is needed to accurately decode DBPSK.
- **DPSK Demod:** This block expects complex numbers in which it then demodulates into bytes that are either 0x00 or 0x01.
    - Type: DBSPK – this is type of modulation we want.
    - Samples/Symbol: 2 – this is needed to accurately decode DBPSK.
- **File Sink:** Outputs data to a file
    - File: Path to output file – can be absolute or relative path. *Note: the relative path is relative to the directory that the python script is running from, which, if you are running gnuradio-companion, is the directory from which you ran the gnuradio-companion command from.*

Observe the output in the output file. Is the output data the same as the input data? How can you tell? Try changing the values in your input vector (feel free to increase the length of the input vector). Is the output still the same as the input? Again, explain how you know.

3.1.2 Over the Air (OTA) Transmission

Take the dbpsk_simulation.grc file and break it into two files: one for the transmitter chain and the other for the receiver chain. Now add a USRP sink to the end of the transmitter chain and a USRP source to the beginning of the receiver chain. (You can look back to source files from Lab 1 if you are unclear how to do this).

Run the transmitter and receiver. Again, observe the output in the output file. Is the output data the same as the input data? Are there any discrepancies? If so, what could have caused these errors?

3.2 USRP In-phase/Quadrature Representation

Understanding how a signal is represented can greatly enhance one's ability to analyze and design baseband digital communication systems. As mentioned in Laboratory 1, a bandpass signal can be represented by the sum of its in-phase (I) and quadrature (Q) components. Moreover, the data input to the USRP board is complex, which includes I and Q. Since I and Q play a very important role in digital communications, you are now going to observe I and Q data on the transmitter and receiver sides.

3.2.1 Observing In-phase and Quadrature Data

For the transmitter side, you can download and use IQ_siggen.grc.

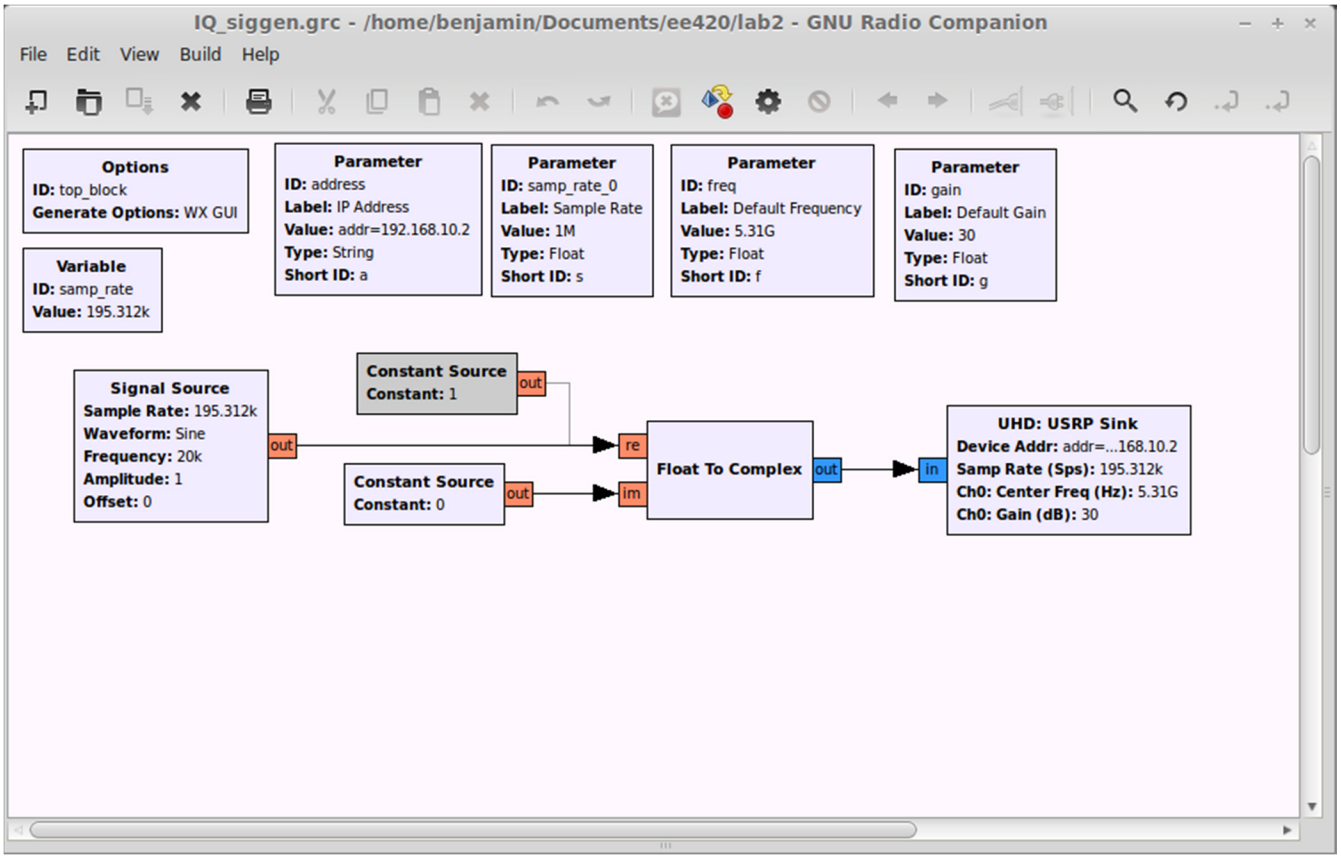


Figure 2: The structure of IQ_siggen.grc. Since the USRP Transmitter block requires the complex input, the *Float To Complex* block converts real and imaginary inputs to a complex valued signal. Note: The Constant Source with the dark gray background means that it is "unplugged" and therefore doesn't affect the flow graph.

For the receiver side, you can download IQ_observe.grc.

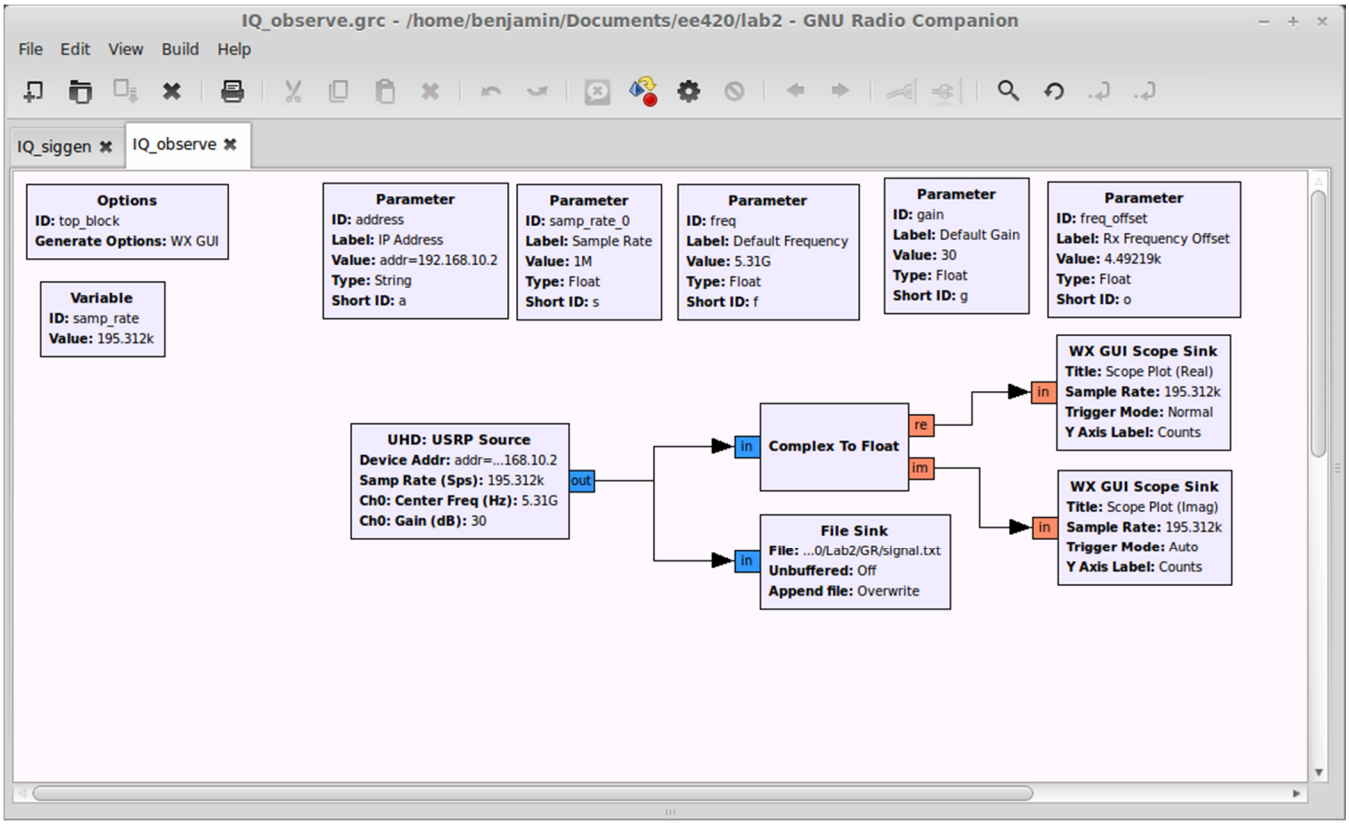


Figure 3: The structure of IQ_observe.grc. Since the USRP receiver block produces the complex output, the *Complex to Float* block converts the complex-valued signal to real and imaginary components, which you can observe using two WX GUI Scope Sinks.

Specify the following six sets of values in IQ_siggen.grc.

Set Index	Real	Imaginary
Set 1	0	0
Set 2	0	1
Set 3	1	0
Set 4	0	sine
Set 5	sine	0
Set 6	sine	sine

This can be done by unplugging and plugging in the source blocks to the corresponding real/imag inputs. Record the corresponding output in IQ_observe.grc by saving screenshots of the plots of the WX GUI Scope Sinks. Is the output the same as the input you have specified? You can also compare/verify your observations by running IQ_sim.grc and observing the output there.

4 Open-ended Design Problem: Frame Synchronization

4.1 Frame Synchronization

Frame synchronization is the process in the telecommunications transmission system used to align the digital channel (time slot) at the receiving end with the corresponding time slot at the transmission end as it occurs. For example, you transmit a series of frames; however there may be gaps in between frame transmissions. At the receiver side you need to know where each frame actually starts in order to decode the data. Thus, you will need to implement frame synchronization to do this.

Frame synchronization involves the following steps: In the first step, the transmitter injects a fixed length symbol pattern, called a marker, into the beginning of each frame to form a marker and frame pair, which is known as a packet. Packets are then converted from symbols into a waveform and transmitted through the channel. The receiver detects the arrival of packets by searching for the marker, removes the markers from the data stream, and recovers the transmitted messages. Marker detection is the most important step for frame synchronization.

In this section you are going to use gnuradio to implement a system that uses a 13-bit barker code for frame synchronization and eventually incorporate the USRP into it. In the end, you need to transmit "Hello world" from end to end using two USRPs.

4.2 Barker Code

A Barker code is sequence of N values of +1 and –1:

$$a_j \text{ for } j = 1, 2, \ldots, N$$

such that:

$$\left| \sum_{j=1}^{N-v} a_j a_{j+v} \right| \leq 1$$

for all $1 \leq v < N$.

Barker codes are commonly used for frame synchronization in digital communication systems. Barker codes have a length of at most 13 and possess low correlation side lobes. A correlation side lobe is the correlation of a code word with a time-shifted version of itself. An example of autocorrelation function of Barker-7 code is shown in Fig. 4 below. It is obvious from this figure that the side lobes are low.

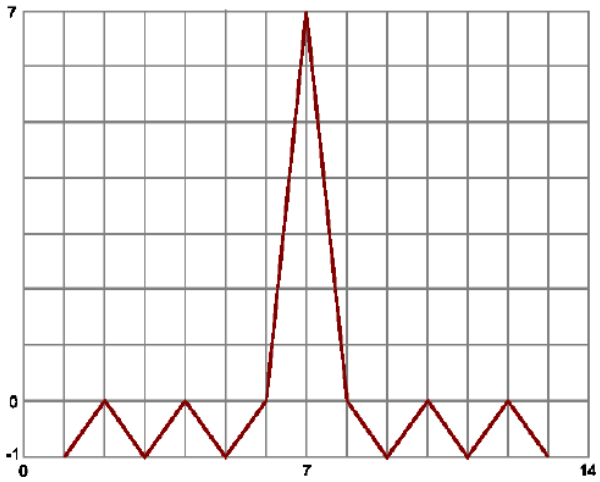


Figure 4: Autocorrelation function of Barker-7 code, which has low side lobes.

4.3 Simulation

In this section you will use GNUradio to implement frame synchronization. This will be done by using correlation at the receiver to search for the 13-bit barker sequence listed below.

Bipolar 13-bit Barker sequence:

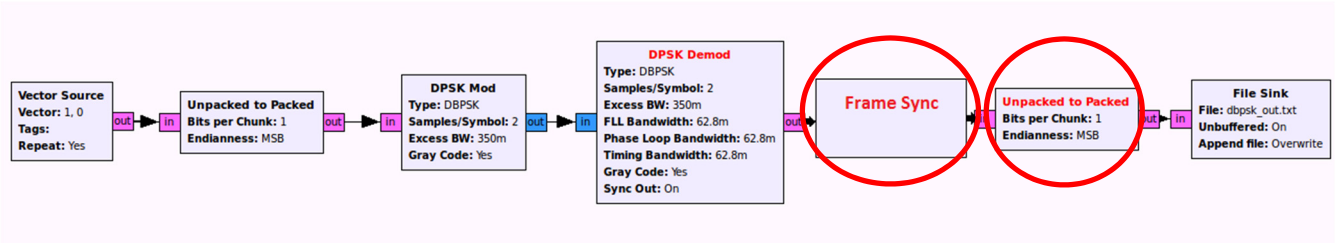
[-1, -1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, -1]

Your task is to transmit packets that consist of the Barker Sequence + Data, and on the receiver side detect the barker sequence, remove it, and recover the Data.

Download the files: frame_sync_sim.py, frame_sync.py, and string_to_list.py.

The file frame_sync_sim.py is the main file that you will run and contains the top_block that connects everything together. All of the blocks that you will need are already instantiated and connected together. All you will need to do here is assign the variable input_vector to be the barker code + data. In order to make your life easier, you awesome TA has written a function in the file string_to_list.py called conv_string_to_1_0_list which, as the name implies, takes in a string, such as "Hello World", and converts it to a list of 1's and 0's such as [1, 1, 0, 0, 1, 0, 1 …etc.]. This list can then be fed into a vector source.

The block diagram that corresponds to the flow graph can be seen in the figure below.



You will notice that this flow graph is very similar to the flow graph from part 3 except for the two new blocks between the *DBPSK Demod* and the *File Sink*. The first of these blocks is called *Frame Sync*. Your task is to implement the internals of this block so that it does frame synchronization by searching for the barker code *using correlation*. The second block technically is optional; however it is recommended that you use it. Remember, the output of the DBPSK modulator is a stream of bytes that are either 0's or 1's – this is helpful since the barker code is also a sequence of 0's and 1's; therefore it makes doing the correlation straight forward. However, after you have found the barker code and remove it, you will want to convert the following data back to packed bytes before passing it to the *File Sink* so that it is easier to read.

A template for the *Frame Sync* block can be found in frame_sync.py. It is your job to fill in the missing correlation code in the general_work function.

A few things to think about:

- Unipolar vs Bipolar: The Barker code correlation properties only hold when it is in bipolar form, however the DBPSK mod/demod blocks input/output unipolar data.
- Finite input buffer: When doing the correlation to find the barker sequence, how do you make sure you don't walk off the end of the input buffer?
- While you are searching for the barker sequence what are you outputting? After you have found the barker sequence what do you do? How do you know how long your data is? How are you keeping track of the number of items that you have output?

4.4 USRP Simulation

Now that your simulation is working, break frame_sync_sim.py into two files: one for a transmit chain and the other for the receive chain similar to what you did in section 3.1.2. You can get the python code for the *USRP Sink* and *USRP Source* blocks from the auto-generated python code that gnuradio-companion created from the .grc files you made in section 3.1.2. *Hint: It may be helpful to pad some 0's after your data.*

Prove that you are able to transmit a packet consisting of the barker sequence + data and that you are able to fully recover the data that you transmitted. Are there any errors? Calculate your average packet success rate.

How do you know how many packets have been transmitted? *Hint: You can create a very long vector in python by multiplying your input vector by an integer number of times you want it to repeat and then pass that into your vector source – i.e.* blocks.vector_source_b(input_vector*100, True,…)).

4.5 Extra Credit: Packets

You are only allowed to do the extra credit once everything else works.

Modify your system above so that your packets have a sequence number and can be variable length.

The easiest way to do this is to replace the vector source with a *Message Source*, and then use the function msgq.insert_tail(packet) to pass your packets to the *Message Source*.

Prove that your system works with variable length packets and calculate your average packet success rate. Does this rate depend on the length of the packet?

Some things to think about:

- How do you know your packet has been received correctly?
- How does the receiver know the length of the packet?
- What if the receiver find's the start of the packet but cannot decode the length of the packet?