بسم الله الرحمن الرحيم

EMERGENCY COMMUNICATION SYSTEM DEVELOPMENT

Document Version	Description	Edited By	Release Date
V1.0	Initial Release	Raja Mourad	3/1/2023

Written By:

Raja Mourad

LAST UPDATED: January 3, 2023

TABLE OF CONTENTS

1 Introduction 6
2 Requirements 6
3 System Overview 8
3.1 System Setup 9
4 Understanding Wireless Range Calculations 9
4.1 Power and dbm Calculations 9
4.2 Path Loss 10
5 Intervention Units 11
5.1 IUs System Architecture 11
5.2 IUs System Components 11
5.3 NRF24 Signal Interpretation 12
6 Scouting Units 13
7 Base Station 13
7.1 Base Station Components 13
7.2 Base Station Software Packages 14
7.3 Communicating with IUs 15
7.3.1 Building the Receiver and Decoder 15
7.3.2 Building the Transmitter 23
8 How to Use 28

List of Figures

Figure 2-1 System Planned GUI 8
Figure 3-1 System Architecture 8
Figure 3-2 System Setup 9
Figure 5-1 Intervention unit basic architecture 11
Figure 5-2 An Enhanced Shockburst packet with payload (0 - 32 bytes) 12
Figure 5-3 Packet control field 12
Figure 7-1 Gnuradio NRF24 demodulator and decoder (V1.02) 15
Figure 7-2 Low pass filter response to TX packets being received 16
Figure 7-3 Binary slicer and multiply constant output 18
Figure 7-4 Before a NRF packet is received 18
Figure 7-5 After NRF packets are received 18
Figure 7-6 After NRF packet is received zoomed in 19
Figure 7-7 NRF24 packet received 19
Figure 7-8 Decode NRF24 python block flowchart (V1.02) 21
Figure 7-9 Gnuradio NRF24 Transmitter (V1.02) 23
Figure 7-10 NRF24 Packet Generator block flowchart (V1.02) 24
Figure 7-11 Packed TX output byte stream 25
Figure 7-12 Unpacked TX output byte stream 26
Figure 7-13 GFSK modulation block output 27
Figure 8-1 Arduino Nano configuration in Arduino IDE 28
Figure 8-2 Successful connection to NRF24 29
Figure 8-3 Gnuradio GUI TX menu 30
Figure 8-4 Gnuradio GUI RX menu 30

List of Tables

Table 2-1 EmerComm Software Requirements 6
Table 5-1 Intervention unit system components (for prototyping) 11
Table 7-1 Base Station Components 13
Table 7-2 Soapy HackRF source block parameters 15
Table 7-3 Low pass filter block parameters 16
Table 7-4 Decode NRF24 python block parameters 20
Table 7-5 NRF24 Packet Generator block parameters 24
Table 7-6 Soapy hackRF sink block parameters 27

SOURCE CODE VERSION CONTROL

Version	**Features**
V1.0	Initial release: Gnuradio script (Base Station) to receive packets from NRF24 with 2 Mbps data air rate and configurable communication channel.
V1.01	Text messages received from NRF24 through gnuradio are now decoded and the payload is extracted.
V1.02	Base station can send text data back to NRF24 modules

1 INTRODUCTION

This article demonstrates the development process of an emergency communication system (EmerComm). This system is intended to be used where secured and independent communication channels are requested. It communicates via a Control Station (CS) with multiple Intervention Units (IU) and multiple Scouting Units (SU).

The Control Station (CS) is responsible for:

- Sending position and altitude commands to scouting units.
- Sending text and voice messages to intervention units.
- Receiving position, images, and video livestream from SU.
- Receiving position, text, and voice information from IU.

2 REQUIREMENTS

The system software requirements are listed in the requirements tracking sheet. In short, the following requirements shall be met.

Table 2-1 EmerComm Software Requirements

Req. ID	Description	Field	Status
Req_001	The EmerComm system consists of 3 parts: Control station "CS", Scouting units "SU", and intervention units "IU".	General	
Req_002	SU shall send its location (Lat, Long, Alt) periodically to the CS.	SU	
Req_003	SU shall be able to send images or video to the CS on request.	SU	
Req_004	SU shall change its location by a CS command.	SU	
Req_005	SU shall use an SDR for its communication with the CS.	SU	
Req_006	A voice communication channel shall be established to make a streaming voice communication channel between the IUs and IU-CS (broadcasting).	General	
Req_007	IU shall be able to send/receive text message to/from the CS.	IU	
Req_008	IU shall send its location (Lat, Long, Alt) periodically to the CS.	IU	

Req_009	IU may use any Mid-Range communication module to make the communication with the CS.	IU	
Req_010	All communication packages shall be encrypted.	General	
Req_011	The AES standard shall be used for the encryption.	General	
Req_012	IU may have its own interaction hardware "Input/Output" or it may connect to a mobile phone via Bluetooth to do it.	CS	
Req_013	The CS application shall be developed to be run on windows computer, with an SDR unit connected for the communication.	CS	
Req_014	The CS application main dashboard consists of: radio module, units Listing module, mapping module, chatting module.	CS	
Req_015	The radio module used to listen and calling the IUs.	CS	
Req_016	The units Listing module used to show the on-range units with their info (Name, details, status).	CS	
Req_017	The map module shall be used to locate units (SU, IU) on map using pins.	CS	
Req_018	Distinct pins shall be used for SU and IU.	CS	
Req_019	SU pin shall be movable to send the unit a command to change its location.	CS	
Req_020	The map module may also use the images provided by the SUs instead of the map to locate the units.	CS	
Req_021	The chat module shall be used to send/receive text message between the CS and IU.	CS	
Req_022	The CS shall have the ability to send text message to a specific IU or to all (broadcasting).	CS	

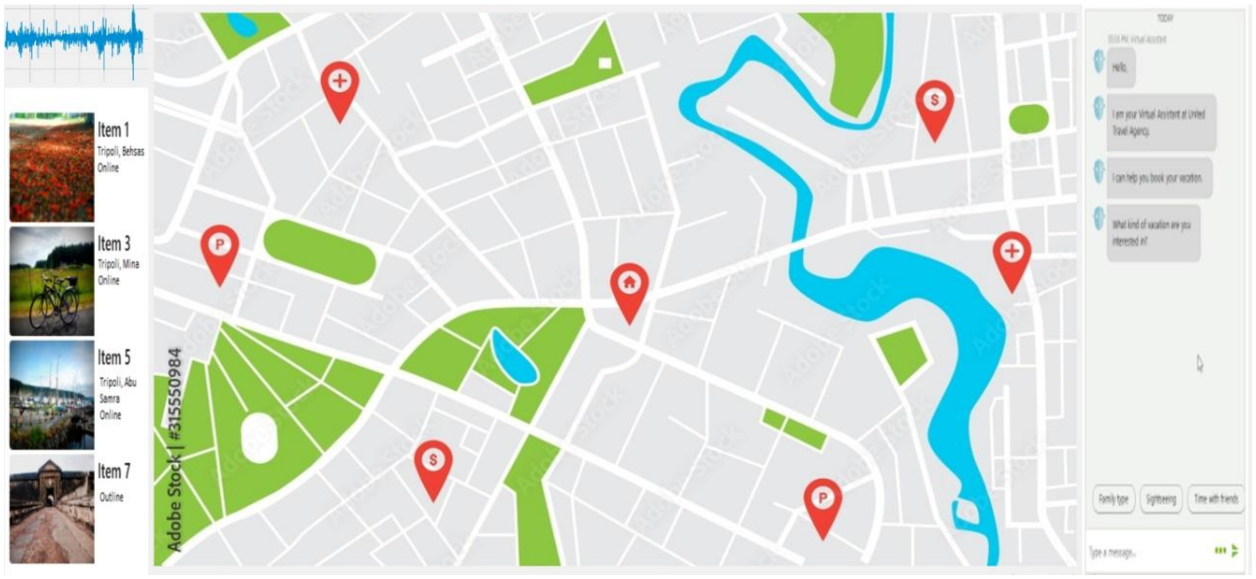


Figure 2-1 System Planned GUI

The planned Graphical User Interface shall consist of a text interface between the intervention units and the bas station. In addition, it should view, in real time, the current position of each IU and each SU. Numbers of connected devices should clearly be displayed with the name of each device.

3 SYSTEM OVERVIEW

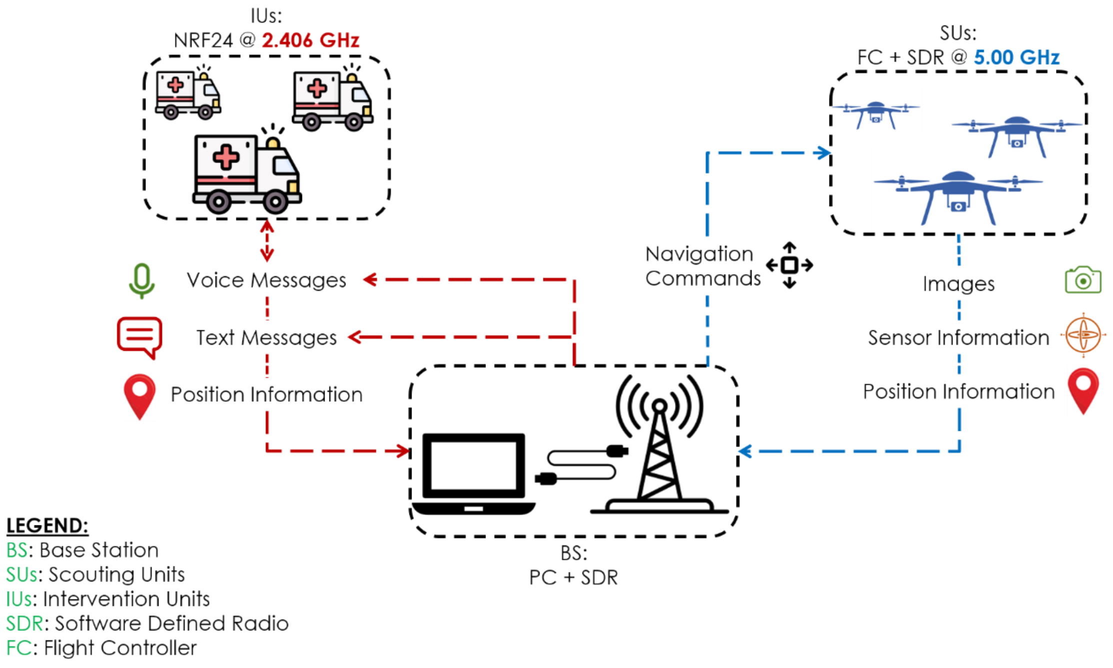


Figure 3-1 System Architecture

The system, as mentioned before, is composed of 3 units: base station, intervention units, and scouting units. The communication channels between the BS and the IUs are held somewhere between 2.4 and 2.5 GHz where this band is very efficient for digital communication protocols.

3.1 System Setup

The system is assembled on a lab-scaled level as shown in Figure 3-2. Each unit is described with a separate section below in the document.

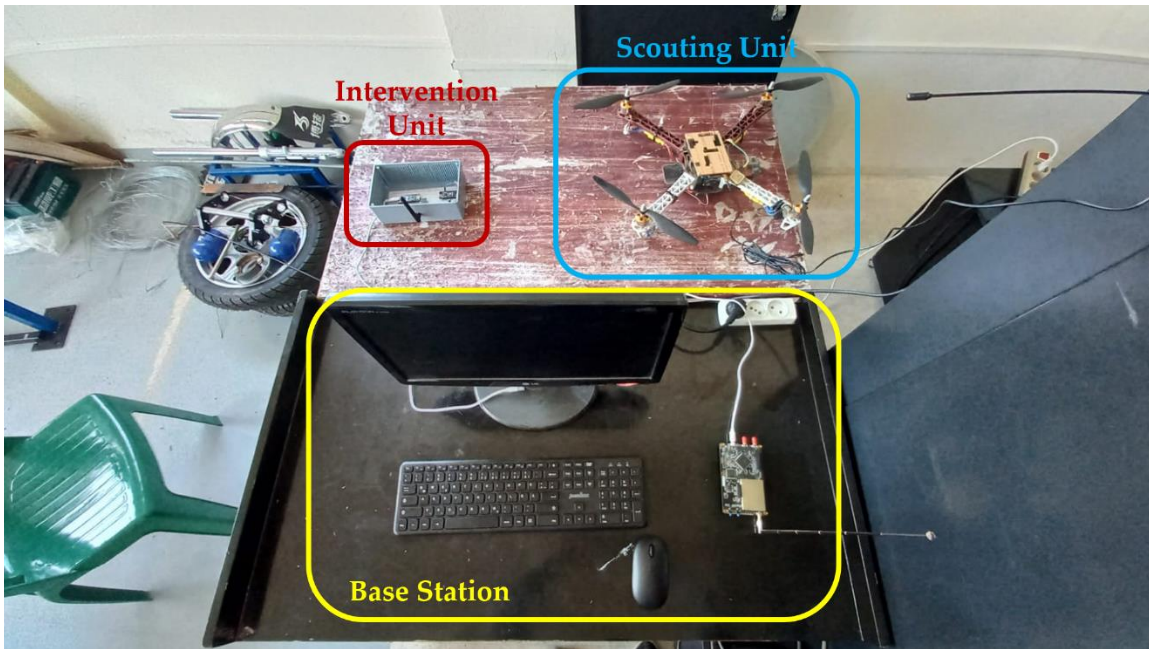


Figure 3-2 System Setup

4 Understanding Wireless Range Calculations

One of the key calculations in any wireless design is range, the maximum distance between transmitter and receiver for normal operation.

4.1 Power and dbm Calculations

RF power is most commonly expressed and measured in decibels with a milliwatt reference, or dBm. A decibel is a logarithmic unit that is a ratio of the power of the system to some reference. A decibel value of 0 is equivalent to a ratio of 1. Decibel-milliwatt is the output power in decibels referenced to 1 mW.

Since dBm is based on a logarithmic scale, it is an absolute power measurement. For every increase of 3 dBm there is roughly twice the output power, and every increase of 10 dBm represents a tenfold increase in power. 10 dBm (10 mW) is 10 times more powerful than 0 dBm (1 mW), and 20 dBm (100 mW) is 10 times more powerful than 10 dBm. You can convert between mW and dBm using the following formulas:

$$P(dBm) = 10.\log_{10}\big(P(mW)\big)$$

$$P(mW) = 1\,mW.10^{\left(\frac{P(dBm)}{10}\right)} \tag{4-1}$$

4.2 Path Loss

Path loss is the reduction in power density that occurs as a radio wave propagates over a distance. The primary factor in path loss is the decrease in signal strength over distance of the radio waves themselves. Radio waves follow an inverse square law for power density: the power density is proportional to the inverse square of the distance. Every time you double the distance, you receive only one-fourth the power. This means that every 6-dBm increase in output power doubles the possible distance that is achievable.

Besides transmitter power, another factor affecting range is receiver sensitivity. It is usually expressed in –dBm. Since both output power and receiver sensitivity are stated in dBm, you can use simple addition and subtraction to calculate the maximum path loss that a system can incur:

$$\begin{aligned}&Maximum\ path\ loss\\ &\qquad = transmit\ power - receiver\ sensitivity\ +\ gains - losses\end{aligned} \tag{4-2}$$

Gains include any gains resulting from directional transmit and/or receive antennas. Antenna gains are usually expressed in dBi referenced to an isotropic antenna. Losses include any filter or cable attenuation or known environmental conditions. This relationship can also be stated as a link budget, which is the accounting of all gains and losses of a system to measure the signal strength at the receiver:

$$Received\ power\ =\ transmit\ power\ +\ gains - losses \tag{4-3}$$

The goal is to make the received power greater than the receiver sensitivity.

In free space (an ideal condition), the inverse square law is the only factor affecting range. In the real world, however, the range also can be degraded by other factors:

• Obstacles such as walls, trees, and hills can cause significant signal loss.

• Water in the air (humidity) can absorb RF energy.

• Metal objects can reflect radio waves, creating new versions of the signal. These multiple waves reach the receiver at different times and destructively (and sometimes constructively) interfere with themselves. This is called multipath.

5 INTERVENTION UNITS

5.1 IUs System Architecture

Each intervention unit is supposed to send the location, text, and voice messages to the CS and shall receive the same from the CS. (The architecture is changeable upon the change of requirements).

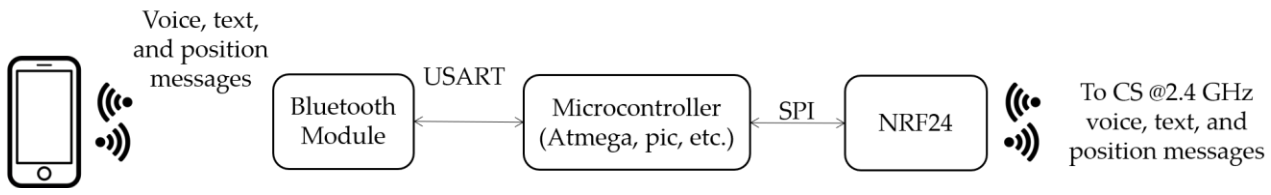

Voice, text, and position messages
Bluetooth Module
USART
Microcontroller (Atmega, pic, etc.)
SPI
NRF24
To CS @2.4 GHz voice, text, and position messages

Figure 5-1 Intervention unit basic architecture

Each intervention unit shall connect to an operator's cell phone via Bluetooth interface. The user shall send text and voice data to the unit which will be processed by a microcontroller and fed to NRF24 module in order to be transmitted by air.

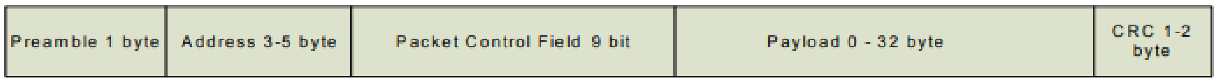
5.2 IUs System Components

Table 5-1 Intervention unit system components (for prototyping)

Component	Description
Nano-RF24	• Microcontroller: ATmega328P-MU QFN32 • Bootloader: Newest 1.8.8 • Wireless: Nrf24L01+ 2.4G • BLE chip: TI CC2540 • Work channel: 2.4G – 2.528G • Transmission distance: ~100m • Architecture: AVR • Input voltage: USB power supply, Vin 6-12V • Operating Voltage: 5V • Flash Memory: 32KB of which 2KB used by bootloader • SRAM: 2KB • Clock Speed: 16MHz

5.3 NRF24 Signal Interpretation

The first step to establish a communication channel between the base station and the IUs is to detect and decode the signal from the NRF24L01 module and be able to replay a signal back to it. This module modulates its signal using Gaussian Frequency Shift Keying (GFSK) with user-defined channel (2.400 to 2.528 GHz separated at 1 MHz @ 1 Mbps and 2 MHz @ 2 Mbps), air rate (1 or 2 Mbps), and antenna power. The module uses Enhanced **Shockburst packet format** for data transmission which contains a preamble field, address field, PCF field, payload field, and CRC field.



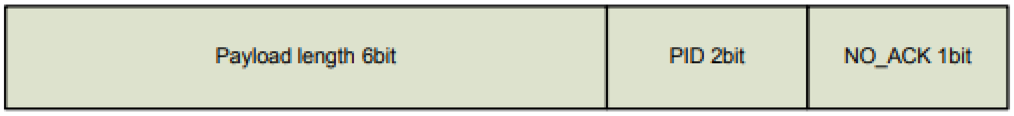
Preamble 1 byte	Address 3-5 byte	Packet Control Field 9 bit	Payload 0 - 32 byte	CRC 1-2 byte

Figure 5-2 An Enhanced Shockburst packet with payload (0 - 32 bytes)

The preamble (1-byte long) is a bit sequence used to detect the 0 and 1 levels in the receiver. It is either 01010101 or 10101010. This is done to ensure there enough transitions in the preamble to stabilize the receiver.

The address is for the receiver. An address ensures that the correct packet is detected by the receiver. It can be configured as 3 to 5 bytes long. When choosing the address, one shall avoid using repetition in characters nor cyclic codes (e.g., 000FFFFFFF or 0101010101) which will raise the packet-error-rate.

The Packet Control Field (PCF) is a 9-bit long sequence which gives information about the payload. This field is only used if the Dynamic Payload Length function is enabled.



Payload length 6bit	PID 2bit	NO_ACK 1bit

Figure 5-3 Packet control field

Payload length gives the information about the user-defined message length sent by the transmitter. It can be either static or dynamic.

The 2-bit PID (Packet Identification) field is used to detect if the received packet is new or retransmitted. PID prevents the receiver from presenting the same payload more than once to the. The PID field is incremented at the transmitter side for each new packet received through the SPI. The PID and CRC fields are used by the receiver device to determine if a packet is retransmitted or new. When several data packets are lost on the link, the PID fields may become equal to the last received PID. If a packet has the same PID as the previous packet, nRF24L01 compares the CRC sums from both packets. If the CRC sums are also equal, the last received packet is considered a copy of the previously received packet and discarded.

The Cyclic Redundancy Check (CRC) is the error detection mechanism in the packet. It my either be 1 or 2 bytes and is calculated over the address, PCF, and payload:

The polynomial for 1 byte CRC is $X^8 + X^2 + X + 1$, Initial value 0xFF. The polynomial for 2-byte CRC is $X^{16} + X^{12} + X^5 + 1$, Initial value 0xFFFF

The goal is to develop a transceiver with the same packet format described by the NRF24 module. In reception mode, the receiver (SDR at the CS side) shall locate the presence of a known address to start interpreting the data. While in transmitting, the CS shall assemble the same packet format in order for the module to receive successfully.

6 SCOUTING UNITS

Still under implementation

7 BASE STATION

7.1 Base Station Components

The base station components are listed in Table 7-1 below.

Table 7-1 Base Station Components

Component	Description
Hackrf one	RF Device: • 30 MHz to 6 GHz operating frequency • Half-duplex transceiver • Up to 20 million samples per second • 8-bit quadrature samples (8-bit I and 8-bit Q) • Software-configurable RX and TX gain and baseband filter • Software-controlled antenna port power (50 mA at 3.3 V) • Power amplification up to 15 dBm.
PC	Processing Device: • 64-bit operating system • At least 4 GB RAM • SSD is preferred. • Linux-based OS (Ubuntu 20.04)

7.2 Base Station Software Packages

The following software packages must be installed on the ubuntu-based machine for development.

1. GNU Radio Companion: Used to control the HackRf transceiver which will be used between the control station and the scouting units.

```
sudo add-apt-repository ppa:gnuradio/gnuradio-releases


sudo apt-get update


sudo apt-get install gnuradio python3-packaging
```

2. Arduino IDE (Prototyping): Used to program the nrf24-based controllers for Intervention units.

```
https://www.arduino.cc/en/software
```

3. Hackrf board packages.

```
sudo apt-get install hackrf
```

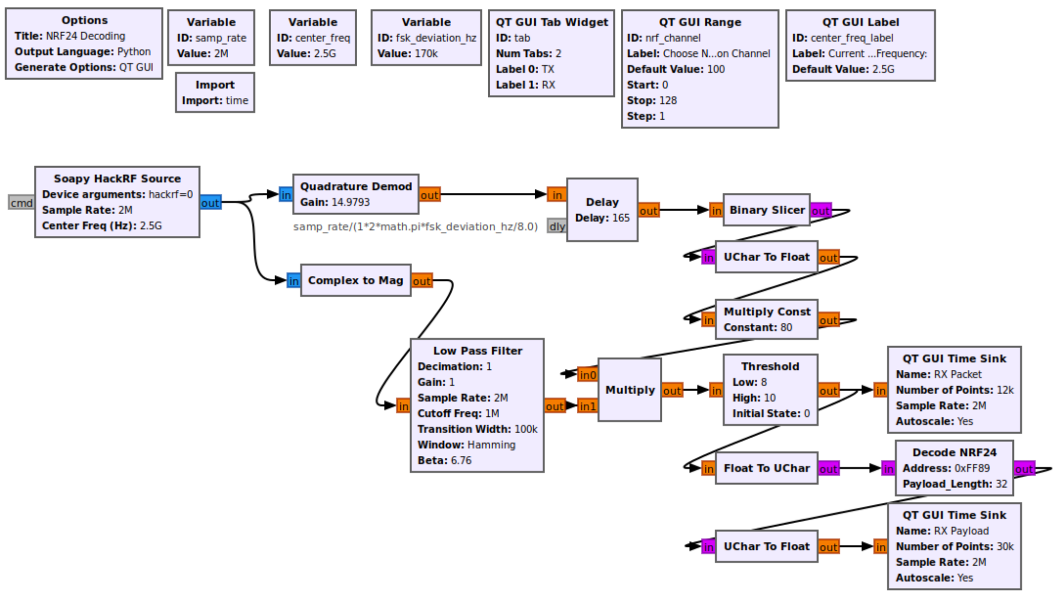
4. to make sure the hackrf is working correctly, open the terminal and type:

```
hackrf_info
```

7.3 Communicating with IUs

IUs use the GFSK modulation technique for message transmission. Hence, a GFSK modulator and demodulator are to be designed and implemented on the base station to handle this communication. Both the receiver and transmitter are put in the same flowgraph but they are separated in this section for explanation reasons.

7.3.1 Building the Receiver and Decoder



Options
Title: NRF24 Decoding
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 2M

Import
Import: time

Variable
ID: center_freq
Value: 2.5G

Variable
ID: fsk_deviation_hz
Value: 170k

QT GUI Tab Widget
ID: tab
Num Tabs: 2
Label 0: TX
Label 1: RX

QT GUI Range
ID: nrf_channel
Label: Choose N...on Channel
Default Value: 100
Start: 0
Stop: 128
Step: 1

QT GUI Label
ID: center_freq_label
Label: Current ...Frequency:
Default Value: 2.5G

Soapy HackRF Source
Device arguments: hackrf=0
Sample Rate: 2M
Center Freq (Hz): 2.5G

Quadrature Demod
Gain: 14.9793
samp_rate/(1*2*math.pi*fsk_deviation_hz/8.0)

Delay
Delay: 165

Binary Slicer

UChar To Float

Complex to Mag

Multiply Const
Constant: 80

Low Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 2M
Cutoff Freq: 1M
Transition Width: 100k
Window: Hamming
Beta: 6.76

Multiply

Threshold
Low: 8
High: 10
Initial State: 0

QT GUI Time Sink
Name: RX Packet
Number of Points: 12k
Sample Rate: 2M
Autoscale: Yes

Float To UChar

Decode NRF24
Address: 0xFF89
Payload_Length: 32

UChar To Float

QT GUI Time Sink
Name: RX Payload
Number of Points: 30k
Sample Rate: 2M
Autoscale: Yes

Figure 7-1 Gnuradio NRF24 demodulator and decoder (V1.02)

The soapy HackRF Source block receives the IQ signal from the air. The user should determine some configuration parameters for this block.

Table 7-2 Soapy HackRF source block parameters

Parameter	Description	Type	Variable	Value
Device arguments	Determines the address of the SDR connected to the PC.	String	-	'hackrf = 0'
Sample rate	Determines the required sample rate, in samples per second, of the hackrf device. It should be chosen according to the received air sample rate.	Float	samp_rate	2e6
Center frequency	Determines the center frequency, in Hz, in which the transmitter is transmitting on.	Float	2.4e9 + nrf_channel * 1e6	2.5e9

Bandwidth	This is the expected bandwidth, in Hz, of the received message.	Float	-	1e6
IF Gain	Intermediate frequency gain.	Float	-	25
VGA Gain	Variable Gain Amplifier.	Float	-	30

The received IQ signal is fed into a magnitude calculator and then into a low pass filter to make it smoother. The magnitude calculator transforms the complex IQ signal into just a magnitude. In other words, it only highlights the necessary part of signal. All the obvious high frequency noises are smoothed via a low-pass filter. The output of the low pass filter is shown below in Figure 7-2 which corresponds to TX identical packets received from the transmitter. The LPF parameters are listed in Table 7-3 below.

Table 7-3 Low pass filter block parameters

Parameter	**Description**	**Type**	**Variable**	**Value**
FIR type	The type of the input and output of this block.	-	-	Float -> Float (Decimating)
Decimation	Decimates the sample rate. i.e., divides the sample rate by this value.	Int	-	1
Sample rate	The sample rate of the message entering this filter.	Float	Samp_rate	2e6
Cutoff frequency	Determines the LPF cutoff freq. Any frequencies above this value will not pass.	Float	-	1e6
Transition width	Is a range of frequencies that allows a transition between a passband and a stopband of a signal processing filter.	Float	-	100e3
Window		-	-	Hamming
Beta	Low pass filter constant	Float	-	6.76

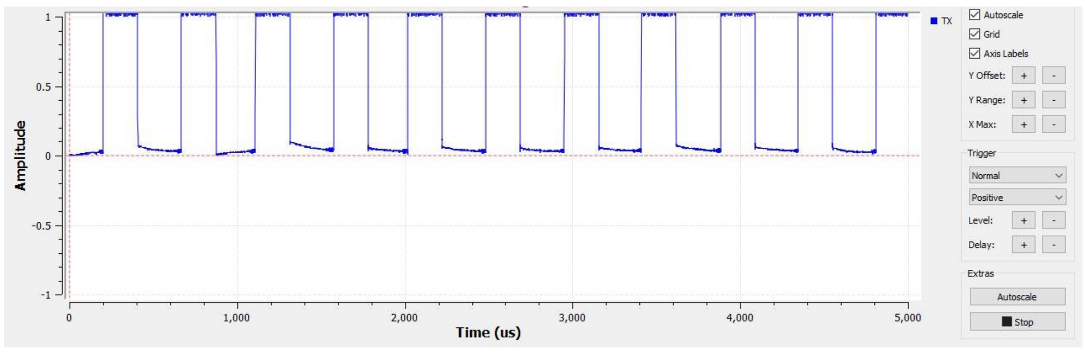


Figure 7-2 Low pass filter response to TX packets being received

On the other side, the IQ signal received, which is GFSK-modulated, undergoes quadrature demodulation which demodulates the FSK signals. This block retrieves the message sent from the transmitter. The output of this block is the signal frequency in relation to the sample rate, multiplied with the gain. Mathematically, this block calculates the product of the one-sampled delayed input and the conjugate un-delayed signal and then calculates the argument of the resulting complex number:

$$Y[n] = \arg\left(A^2 e^{j2\pi\frac{f}{f_s}}\right) \tag{7-1}$$

Where, A is real, and so is A^2, and hence it only scales, therefore $\arg(.)$ is invariant which leads to:

$$\arg\left(A^2 e^{j2\pi\frac{f}{f_s}}\right) = \frac{f}{f_s} \tag{7-2}$$

The gain of this block is determined by the sample rate and the FSK frequency deviation (which is 170 KHz) and pugging the numbers:

$$\text{gain} = \frac{f_s}{2\pi\left(\frac{f_{dev}}{8}\right)} = \frac{2e6}{2\pi\left(\frac{170e3}{8}\right)} = 14.9793 \tag{7-3}$$

The binary slicer block slices a float value producing 1 bit output. Positive input produces a binary 1 and negative input produces a binary zero. In other words, it scales the float input from the quadrature demodulator into 1s and 0s.

The incoming signal may be weak, so a soft amplification has to be done in order to better interpret the signal. Figure 7-3 below shows the output of the binary slicer followed by the gain multiplication of 80. It is clear that the output now is swinging between 0 and 80 instead of 0 and 1.

The following step is to only take the required part of this messy signal. By multiplying the demodulated and amplified signal and the output of the low pass filter (which only focuses on the magnitude of the IQ signal), we can only extract what is necessary and remove what is noise. Hence, the multiply block takes the output of the low pass filter and the output from the amplified quadrature demodulation and outputs the baseband signal.

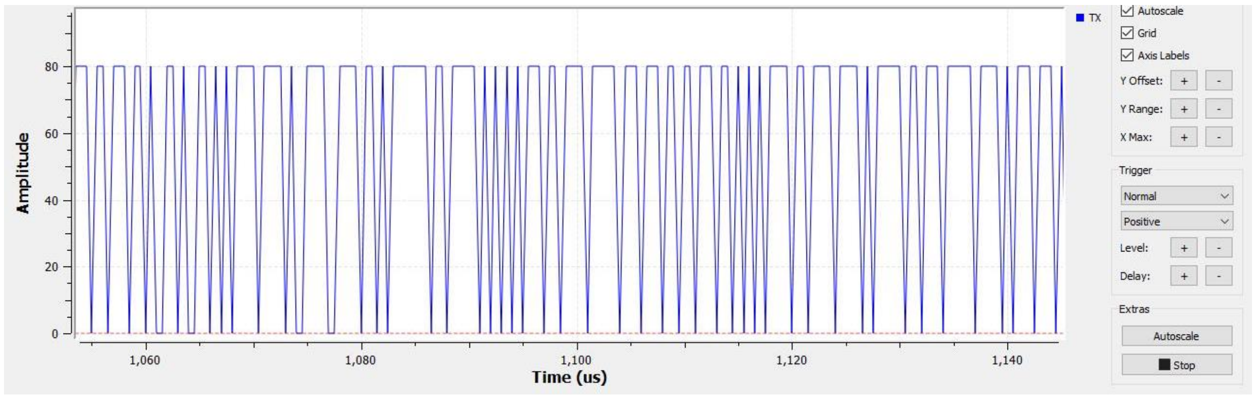


Figure 7-3 Binary slicer and multiply constant output

The output after multiplying the LPF output with the demodulated and amplified signal is shown below in Figure 7-4 and Figure 7-5. It is clearly visible that the noise's amplitude (before receiving the NRF packet) is somewhere between 0 and 4 in amplitude unit.

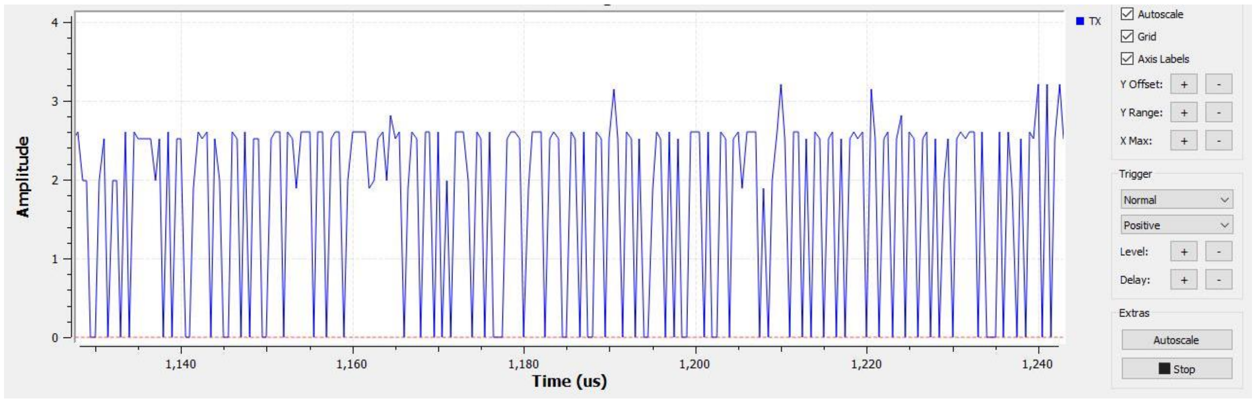


Figure 7-4 Before a NRF packet is received

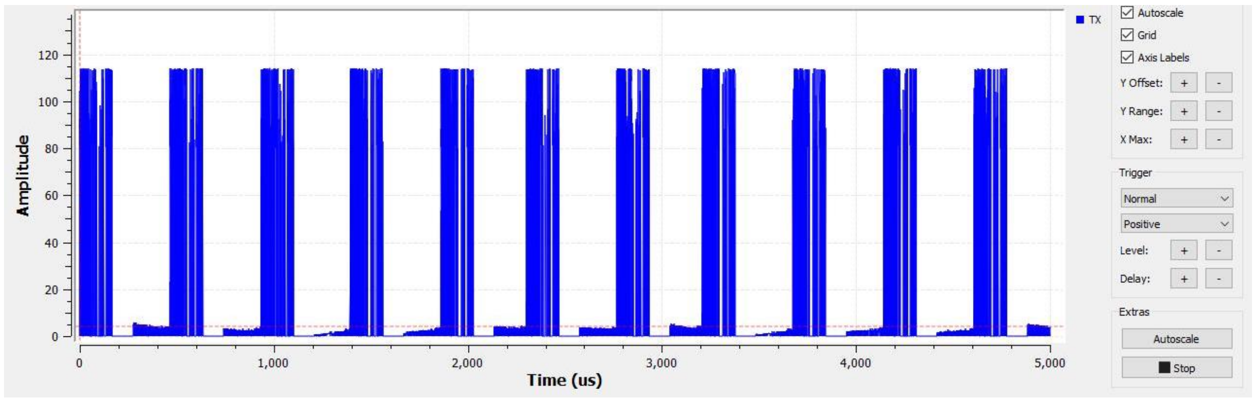


Figure 7-5 After NRF packets are received

Zooming in the above figure:

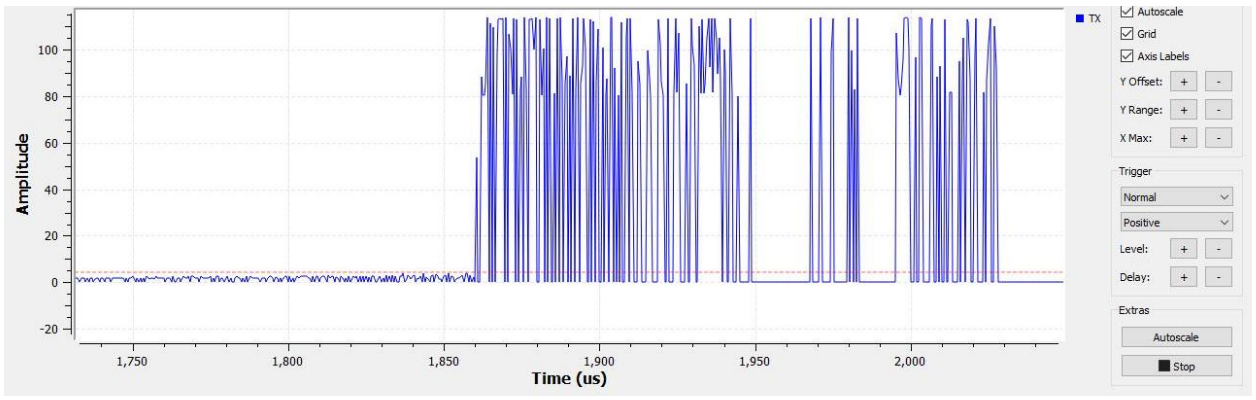


Figure 7-6 After NRF packet is received zoomed in

Figure 7-6 above holds the entire packet within it. But it is still hard to distinguish the packet received from the NRF. Hence, a threshold block is added to filter out the noises from this signal. This block outputs a 1 if the input is greater than its high threshold and a 0 if the input is below its low threshold. The high and low thresholds have been chosen as 10 and 8 respectively. In other words, the necessary information in the signal lay above the amplitude of 10 and anywhere below 8 is considered as noises.

The packet received after adding threshold is shown below in Figure 7-7. The address field is chosen in the Arduino sketch which is completely custom.

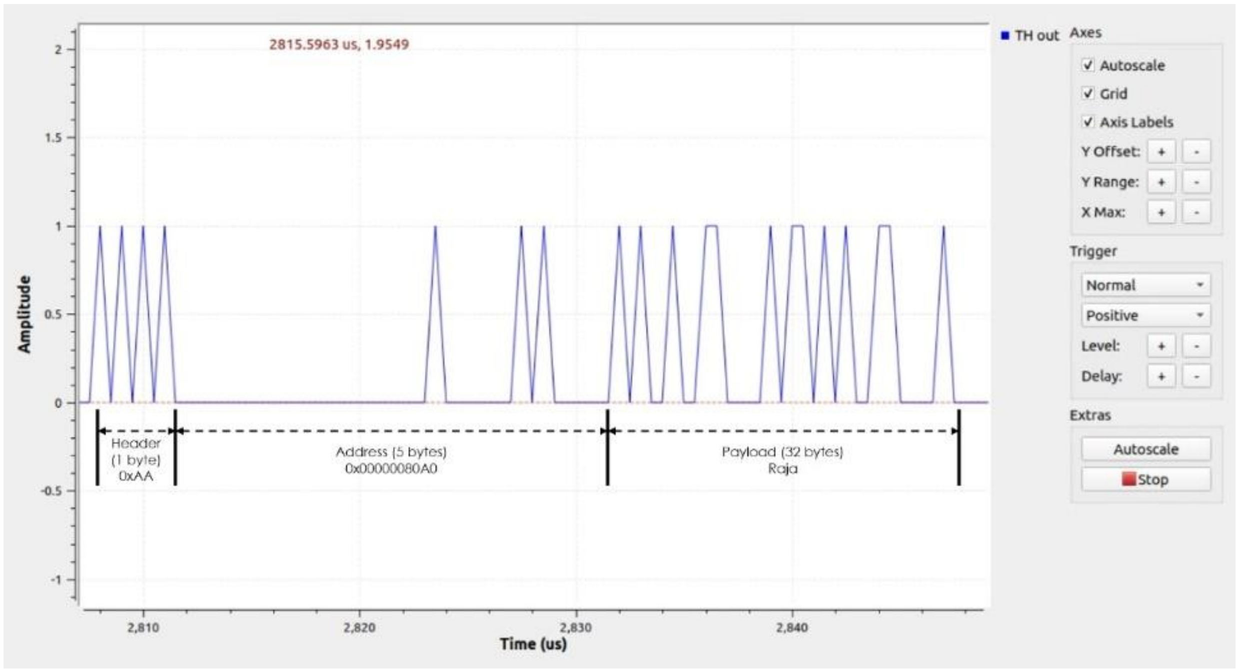


Figure 7-7 NRF24 packet received

After successful reception of the packet, the packet must be decoded to extract the payload. So, a custom python block, decode NRF24, is added. The input to this block is a stream of bytes, it accepts the address and the payload length as parameters, and outputs the payload as stream of bytes. When writing this block, several factors must be considered to ensure proper packet-decoding:

- The input stream is received as a complete list and not element by element.
- The input stream of unpacked bytes (0s and 1s) is in form of a list, input = [0,1,0,0,1,0,….,1,0], with variable length.
- The input stream sample rate is determined by the block before it where it is 2Mbps.
- Processing the input stream and manipulating it with complex CPU functions will result in delayed-sampling period which affects the functionality of the flowgraph.
- The block has 2 main functions; *"init ()"* and *"work ()"*. The *"init"* function is called once when the flowgraph is active and the *"work ()"* function is called whenever there is a new input stream present at the input terminal.

The SW design of this block took into consideration the user custom addresses set by the NRF24 transmitter and the payload size determined also by the TX side.

Table 7-4 Decode NRF24 python block parameters

Parameter	**Type**	**Comments**
Input	Stream of bytes	-
Output	Stream of bytes	-
Address	String of HEX	Max: 5 bytes. Example: FFEEDDCCBB
Payload_length	Int	Min: 1 – Max: 32

As mentioned before, manipulating the input stream with complex functions will absolutely result in delayed-sampling of the input stream and eventually result in bad packet decoding in addition to gnuradio freezing. To solve this problem, the input stream must not be processed when not necessary or when the TX side transmits nothing. The SW design flowchart of this block is shown below in Figure 7-8.

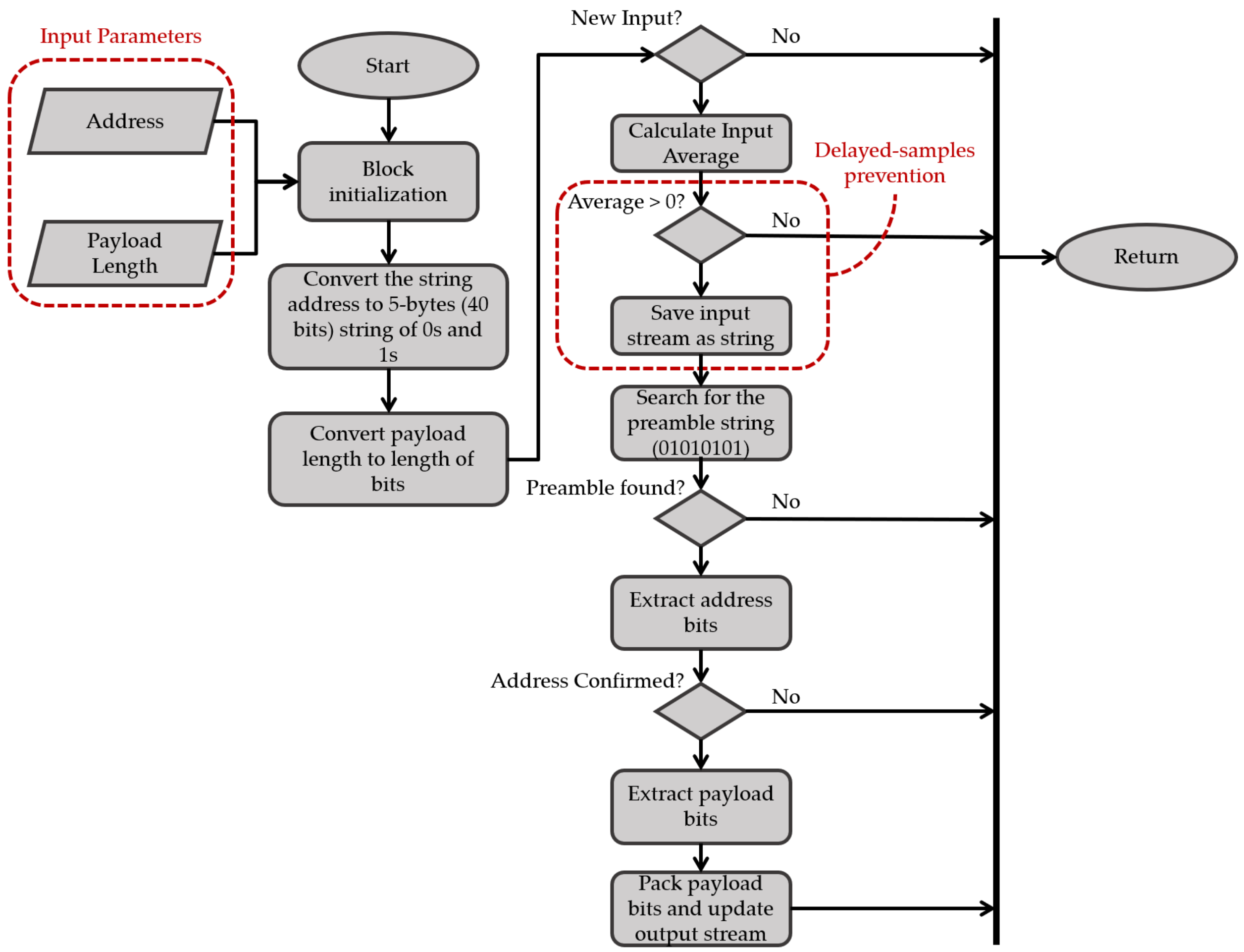

Input Parameters
Address
Payload Length
Start
Block initialization
Convert the string address to 5-bytes (40 bits) string of 0s and 1s
Convert payload length to length of bits
New Input?
No
Calculate Input Average
Delayed-samples prevention
Average > 0?
No
Save input stream as string
Search for the preamble string (01010101)
Preamble found?
No
Extract address bits
Address Confirmed?
No
Extract payload bits
Pack payload bits and update output stream
Return

Figure 7-8 Decode NRF24 python block flowchart (V1.02)

The methodology illustrated above processes the input stream as a whole string of bits (“0100101010…”). Thus, the address parameter, which is input from the user, should be expressed as bits and not bytes (ex: 0x80A0 = 1000000010100000). A mapping function is used to map the address string to its equivalent string of bits which is 5-bytes long.

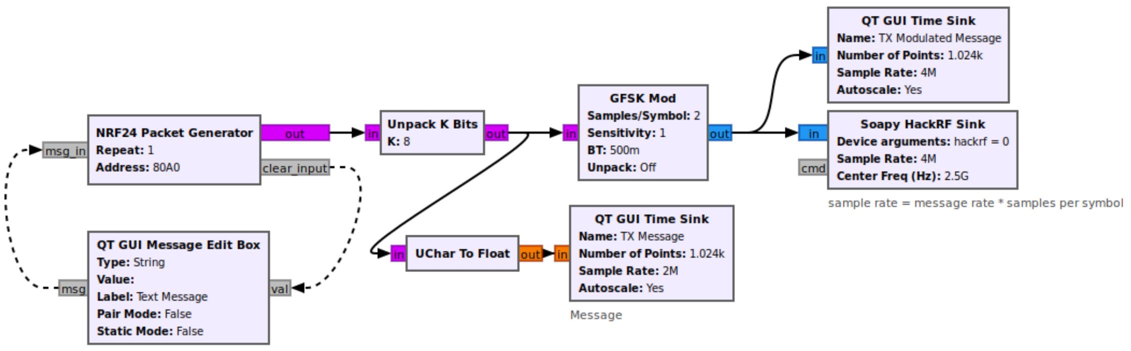
When a new input is present, the average of the list is calculated. If the average is 0, this means that all the input list is 0 and no new packet has arrived and no further processing is required. However, when at least 1 packet is received, the average of the input list becomes different than 0. So, the packet can be processed. This small change calculation will robustly remove the delayed-sampling and process only what is important.

The Decode NRF24 block code is written using VS code and is shown below:

```
"""
Decode NRF24 Packet:
This block receives NRF24 packets at 2 Mbps. If the average of the input list is 0,
this block will ignore the input. Otherwise, useful packet is received, the block will
process the data as a string. This is done to prevent data delayed-sampling.
"""
import numpy as np
from gnuradio import gr
import re
import binascii
import time
import pmt

class blk(gr.sync_block):  # other base classes are basic_block, decim_block,
interp_block

    """NRF24 Decoding - This block decodes the stream coming from NRF24 BTLE
       Parameters:
       self.address: Is the self.address put by the transmitter. The self.address for
both ends must match to decode
             the message.
       Payload Length: The message length in bytes
    """
    def __init__(self, address = '0x00', payload_length = 32):  # only default
arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Decode NRF24',   # will show up in GRC
            in_sig=[np.byte],
            out_sig=[np.byte]
        )

        self.address = address
          # Map the address to its equivalent string of bits
        self.address = str(bin(int(self.address, 16))[2:].zfill(8))
        # Complete the string with MSB 0s to complete the 40 bits
          self.address = '0' * (40 - len(self.address)) + self.address
          # Express the payload length in bits instead of bytes
        self.payload_length = payload_length * 8
        self.text_rec_prev = ""

    def work(self, input_items, output_items):
        bits = "" # Variable string to store input stream
        input = input_items[0] # save the input stream
        avg = sum(input) / len(input) # Calculate Input Average

        # if the input stream is different than 0
        # this avoids delayed-sampling
        if avg > 0:
            for i in range (len(input)):
                bits += str(input[i])

        preamble_header = [m.start() for m in re.finditer('01010101', bits)] # Extract
the preamble sequence
        output_items[0][:] = '0' # Clear all residual data in the output buffer
        if len(preamble_header) > 0:
            for header_index in preamble_header:
                text_rec = "" # Variable to store the characters of the payload

                if bits[header_index + 8: header_index + 8 + len(self.address)] ==
self.address:
```

```
                    PCF = bits[header_index + 8 + len(self.address): header_index + 8
+ len(self.address) + 9]

                    PLD_LENGTH = int(PCF[0:6], 2)
                    PID = int(PCF[6:8], 2)
                    NO_ACK = int(PCF[8], 2)

                    #print(PLD_LENGTH, PID, NO_ACK)

                    payload = bits[header_index + 8 + len(self.address) + 9:
header_index + 8 + len(self.address) + 9 + self.payload_length]

                    for x in range(self.payload_length):

                     # Update the output stream
                        output_items[0][x] = payload[x]

                    for pld in range(0,len(payload) - 1, 8):
                        an_integer = int(payload[pld:pld + 8], 2)
                        ascii_char = chr(an_integer)
                        text_rec += ascii_char

                    if self.text_rec_prev == text_rec:
                        print(text_rec) # Print the payload

                    self.text_rec_prev = text_rec

            return(len(output_items[0]))

        return(len(output_items[0]))
```

7.3.2 Building the Transmitter


NRF24 Packet Generator — Repeat: 1; Address: 80A0

QT GUI Message Edit Box — Type: String; Value: ; Label: Text Message; Pair Mode: False; Static Mode: False

Unpack K Bits — K: 8

UChar To Float

GFSK Mod — Samples/Symbol: 2; Sensitivity: 1; BT: 500m; Unpack: Off

QT GUI Time Sink — Name: TX Modulated Message; Number of Points: 1.024k; Sample Rate: 4M; Autoscale: Yes

Soapy HackRF Sink — Device arguments: hackrf = 0; Sample Rate: 4M; Center Freq (Hz): 2.5G

QT GUI Time Sink — Name: TX Message; Number of Points: 1.024k; Sample Rate: 2M; Autoscale: Yes

sample rate = message rate * samples per symbol

Figure 7-9 Gnuradio NRF24 Transmitter (V1.02)

The transmitter design is the complete opposite of the receiver. The payload must be packetized in the same form of the packet format in Figure 5-2. The whole packet then should undergo GFSK modulation to match the format of the NRF24 packet. Figure 7-9 above shows the gnuradio flowgraph to transmit a text message to the hackRF. The “NRF24 Packet Generator” is a custom block used to combine the payload with the preamble and the address. Note that the CRC checking is disabled in both the gnuradio

TX and the NRF24 RX for the aim of simplicity but will be added in the future. This block has the following user-defined parameters:

Table 7-5 NRF24 Packet Generator block parameters

Parameter	**Type**	**Comments**
Address	string	The write address. This must match with the receiver in order to accept the message. Max: 5 bytes
Repeat	int	The number of times to repeat the TX message.
Msg_in	PMT	Receives the payload from a text field editor in form of pmt message.
Clear_input	PMT	Used to clear the text field editor after processing.
Out	byte	Byte of packed bits.

The TX flowgraph must be executed whenever a new string message is entered in the text field in order to not interfere with the RX flowgraph. Thus, the SW design architecture of the packet generator is as follows:

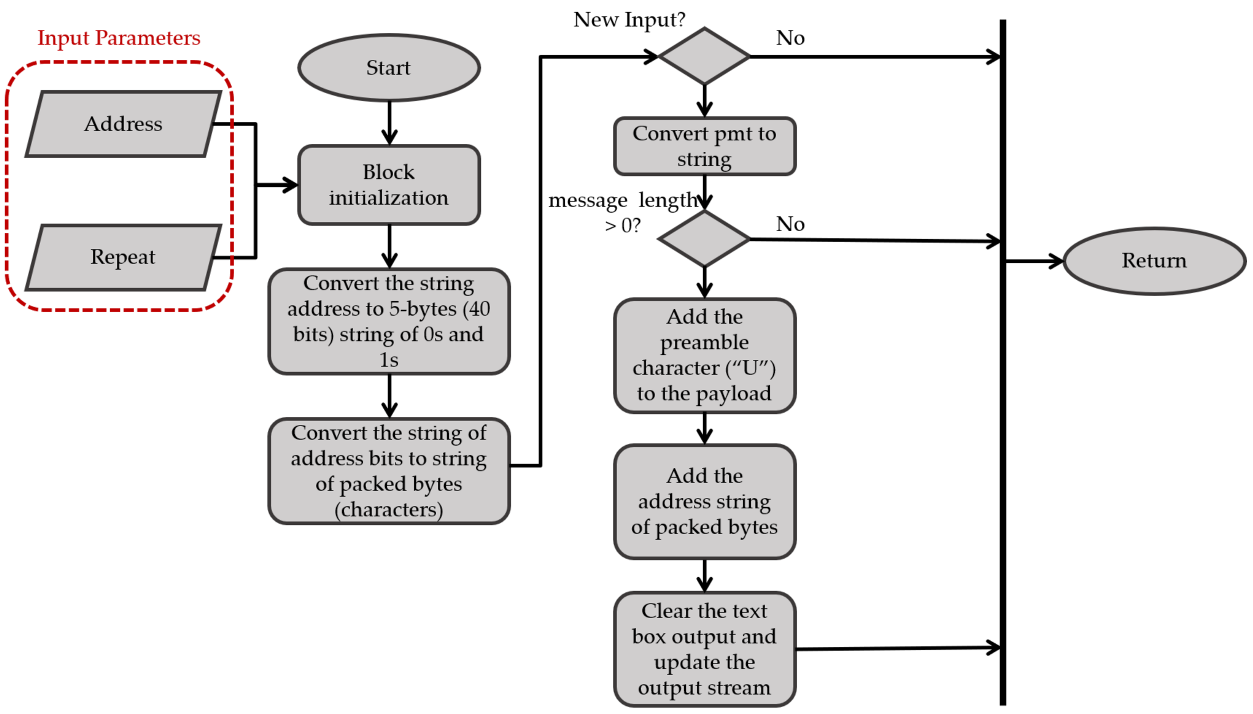

Input Parameters
Address
Repeat
Start
Block initialization
Convert the string address to 5-bytes (40 bits) string of 0s and 1s
Convert the string of address bits to string of packed bytes (characters)
New Input?
No
Convert pmt to string
message length > 0?
No
Add the preamble character ("U") to the payload
Add the address string of packed bytes
Clear the text box output and update the output stream
Return

Figure 7-10 NRF24 Packet Generator block flowchart (V1.02)

When writing a message in the textbox editor and set the repeat parameter to 1, the output byte stream looks like:

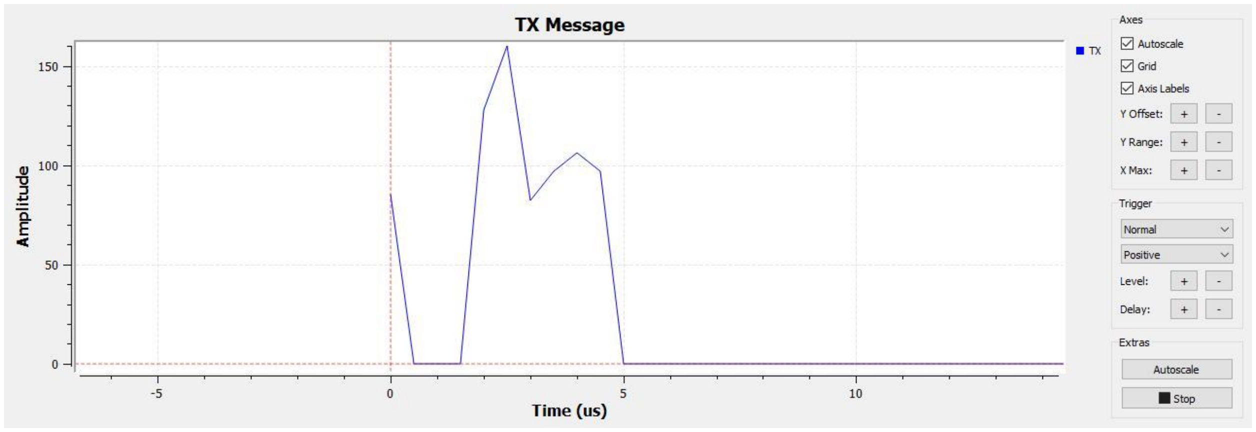


Figure 7-11 Packed TX output byte stream

The code of the NRF24 packet generator block is as follows:

```
import numpy as np
from gnuradio import gr
import pmt
import time

textboxValue = ""

class blk(gr.sync_block):

    """This block generates the NRF24 packet format."""

    def __init__(self, repeat = 5, address = str(00)):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='NRF24 Packet Generator',   # will show up in GRC
            in_sig=None,
            out_sig=[np.byte]
        )
        self.message_port_register_in(pmt.intern('msg_in'))
        self.message_port_register_out(pmt.intern('clear_input'))
        self.set_msg_handler(pmt.intern('msg_in'), self.handle_msg)

        self.repeat = repeat

        self.address = address
        self.address = str(bin(int(self.address, 16))[2:].zfill(8))
        self.address = '0' * (40 - len(self.address)) + self.address

        self.address_str = ""
        for i in range (0, len(self.address), 8):
            an_integer = int(self.address[i:i + 8], 2)
            self.address_str += chr(an_integer)

    def handle_msg(self, msg):
        global textboxValue
        textboxValue = pmt.symbol_to_string(msg)

    def work(self, input_items, output_items):
        global textboxValue
        for i in range (len(output_items[0])):
            output_items[0][i] = 0

        # get length of string
        _len = len(textboxValue)
        if (_len > 0):
            # Add the preamble and the address to the payload
            textboxValue = "U" + self.address_str + textboxValue
            _len += 1 + len(self.address_str)

            for i in range (self.repeat):
                # store elements in output array
                for x in range(_len):
                    output_items[0][x + (i * _len * self.repeat)] =
ord(textboxValue[x])



            textboxValue = ""
            self.message_port_pub(pmt.intern('clear_input'), pmt.intern(''))
            return (len(output_items[0]))
        else:
            return (0)
```

The packet needs to unpacked before undergoing modulation. Thus, an unpack block is added which unpacks the packet to (8 bits per byte). The packet after unpacking looks like:

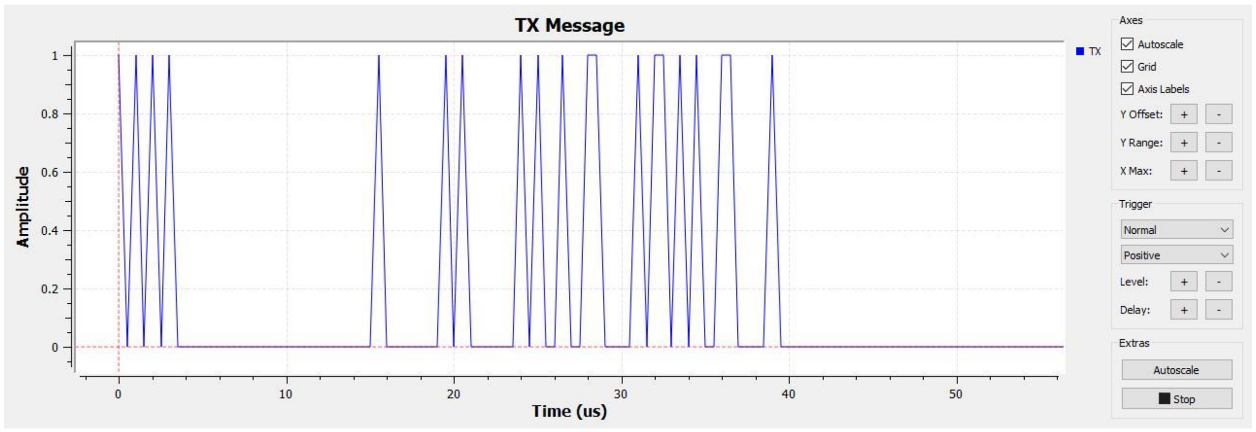


Figure 7-12 Unpacked TX output byte stream

Where the above packet looks very similar to the NRF24 packet detected in Figure 7-7. Thus, the packet is now ready for modulation.

A GFSK modulator block is used to do all the magic. This block converts the stream of unpacked bytes to 2 frequencies:

$$f_{out} = \begin{cases} f_{base} + f_{\Delta}, & if\ message = 1 \\ f_{base} - f_{\Delta}, & if\ message = 0 \end{cases} \quad (7\text{-}4)$$

Where, f_{base} is the center frequency in Hz and f_{Δ} is the FSK deviation frequency of 170 KHz. The input sample rate of the GFSK modulation block is 2Mbps. The output sample rate of the GFSK mod block can be found as:

$$f_{s_{out}} = samples\ per\ symbol \times f_{s_{in}} \qquad (7\text{-}5)$$

Where, the samples per symbol determine how many frequency deviations are there. By default, this is set to 2 and cannot be less than 2. The output of the GFSK mod block, of the same message as above, is as follows:

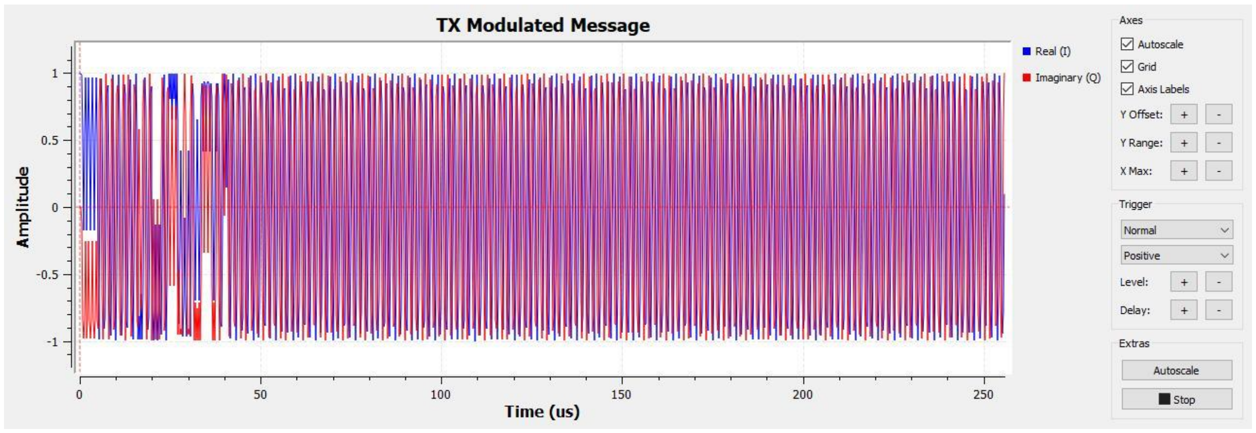


Figure 7-13 GFSK modulation block output

Finally, a soapy hackRF block is used to transmit this message. The parameters of the hackRF sink are:

Table 7-6 Soapy hackRF sink block parameters

Parameter	Description	Type	Variable	Value
Device arguments	Determines the address of the SDR connected to the PC.	String	-	'hackrf = 0'
Sample rate	Determines the required sample rate, in samples per second, of the hackrf device. It should be chosen according to the transmitted air sample rate.	Float	samp_rate * 2	4e6
Center frequency	Determines the center frequency, in Hz, in which to transmit the message at.	Float	2.4e9 + nrf_channel * 1e6	2.5e9
Bandwidth	This is the expected bandwidth, in Hz, of the transmitted message.	Float	-	1e6
VGA Gain	Variable Gain Amplifier.	Float	-	47

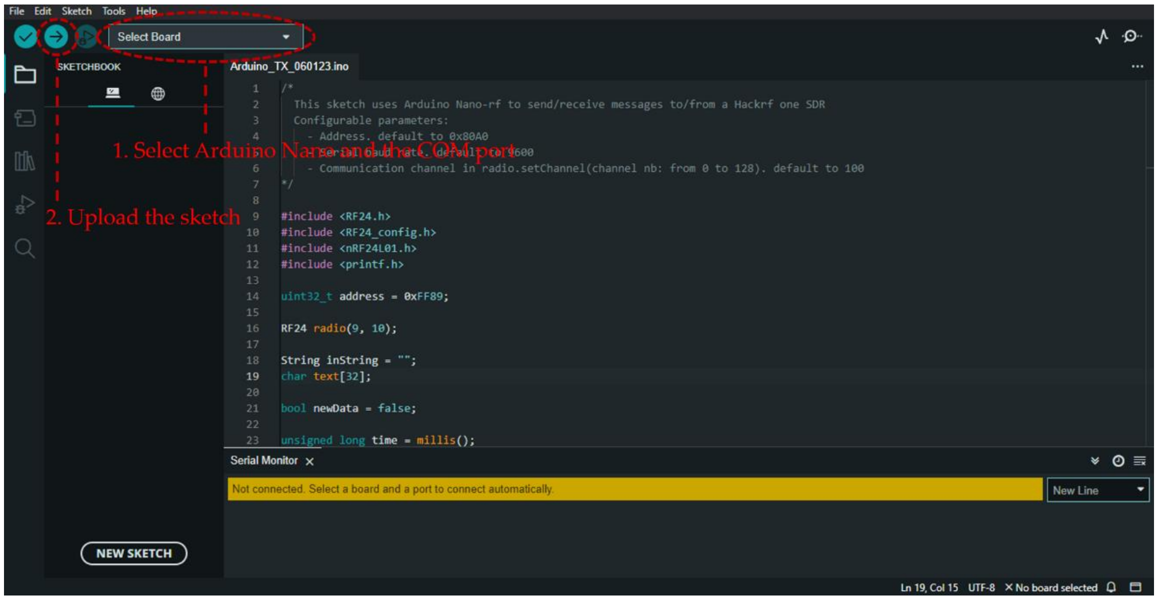
8 How to Use

To run the program, follow the following steps:

1. Open the Arduino transmitter sketch using Arduino IDE.

Arduino_TX_060123
.ino

2. Choose Arduino nano from the select board selection bar along with the COM port and then click the upload button (must be done once). The configurable parameters are the address, the communication channel in the radio.setChannel() function (by default it is 100), and the baud rate.

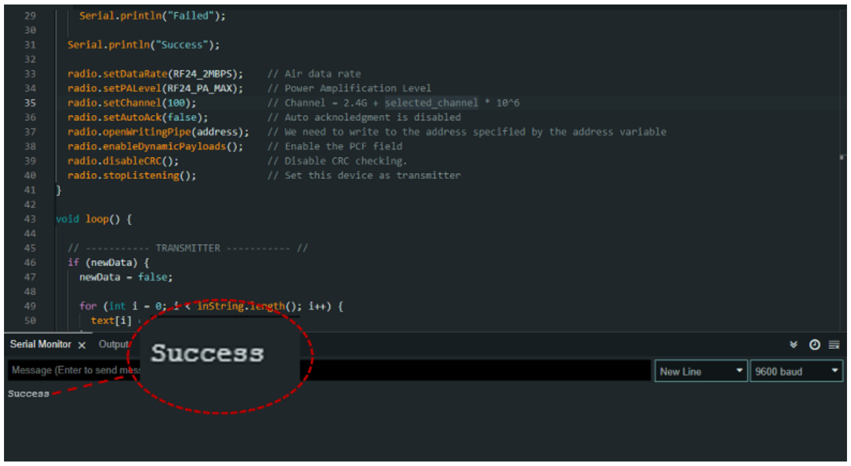


```
/*
  This sketch uses Arduino Nano-rf to send/receive messages to/from a Hackrf one SDR
  Configurable parameters:
    - Address. default to 0x80A0
    - Serial baud rate. default to 9600
    - Communication channel in radio.setChannel(channel nb: from 0 to 128). default to 100
*/

#include <RF24.h>
#include <RF24_config.h>
#include <nRF24L01.h>
#include <printf.h>

uint32_t address = 0xFF89;

RF24 radio(9, 10);

String inString = "";
char text[32];

bool newData = false;

unsigned long time = millis();
```

Not connected. Select a board and a port to connect automatically.

Figure 8-1 Arduino Nano configuration in Arduino IDE

If everything was OK, Arduino IDE must show a “success” message in the serial monitor.



```
    Serial.println("Failed");

  Serial.println("Success");

  radio.setDataRate(RF24_2MBPS);    // Air data rate
  radio.setPALevel(RF24_PA_MAX);    // Power Amplification Level
  radio.setChannel(100);            // Channel = 2.4G + selected_channel * 10^6
  radio.setAutoAck(false);          // Auto acknoledgment is disabled
  radio.openWritingPipe(address);   // We need to write to the address specified by the address variable
  radio.enableDynamicPayloads();    // Enable the PCF field
  radio.disableCRC();               // Disable CRC checking.
  radio.stopListening();            // Set this device as transmitter
}

void loop() {

  // ----------- TRANSMITTER ----------- //
  if (newData) {
    newData = false;

    for (int i = 0; i < inString.length(); i++) {
      text[i]
```

```
Success
```

Figure 8-2 Successful connection to NRF24

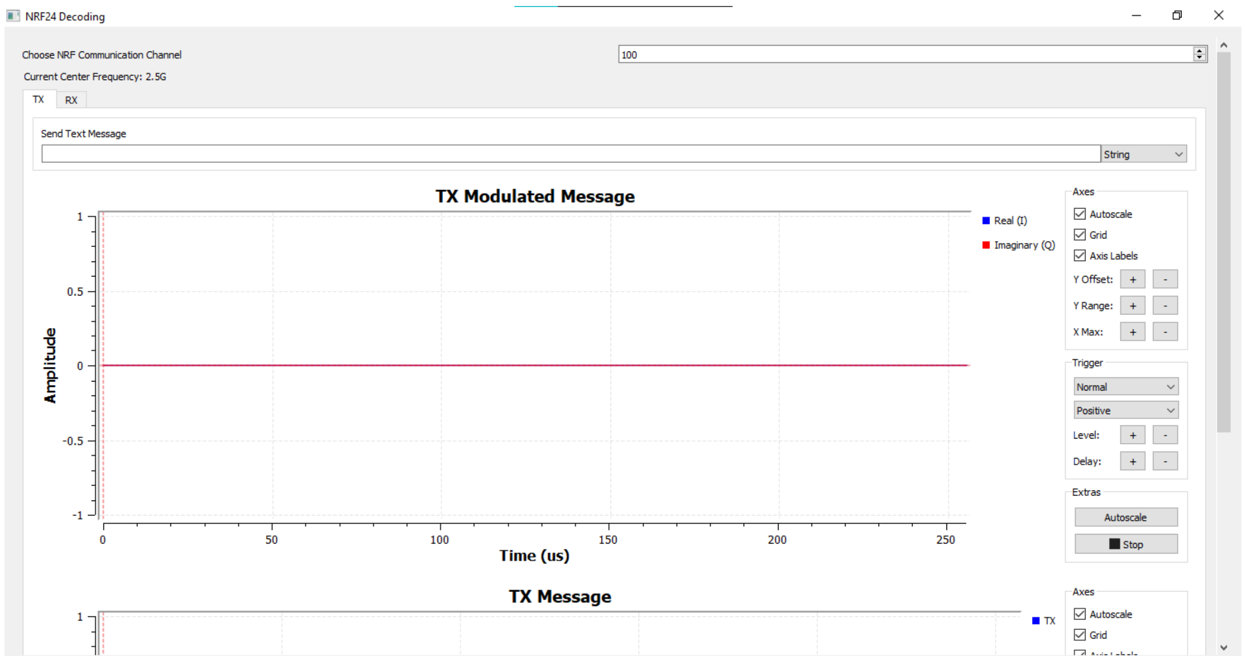
3. Type the message you wish to send in the serial monitor.
4. To receive from the NRF24 module, upload the receiver sketch.

Receiver.ino

5. Open GNUradio flowgraph and click on RUN.

NRF24-demodulator.grc

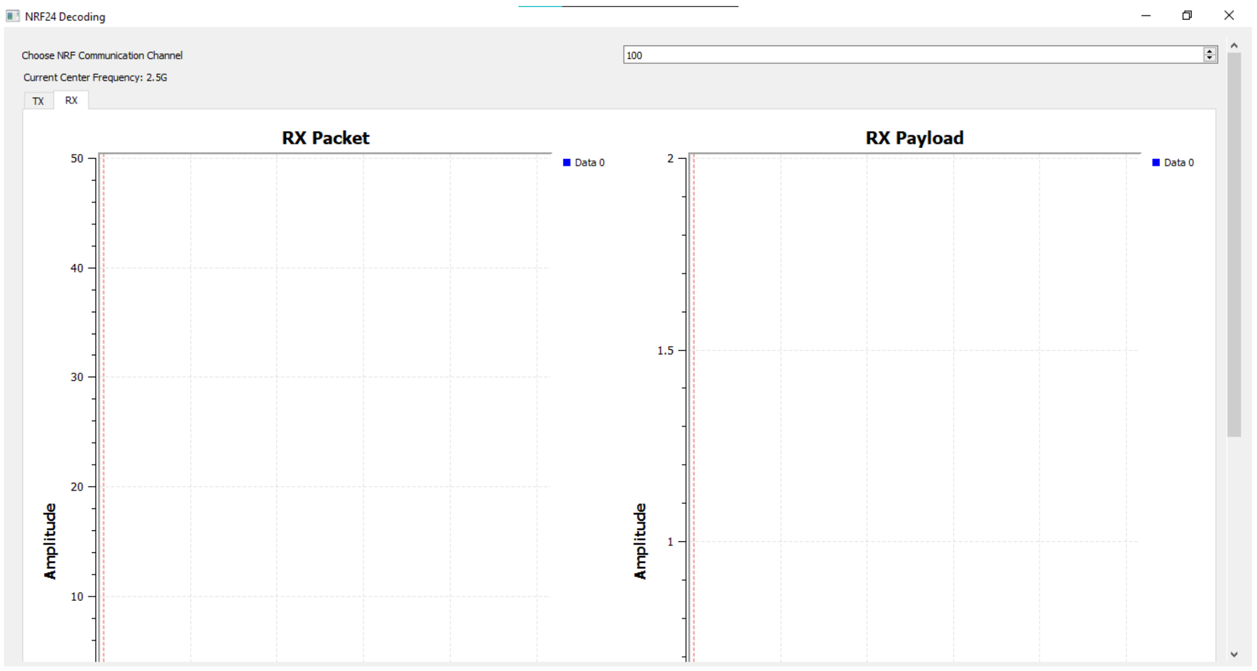
6. Enter desired transmit message in the edit box as shown in Figure 8-3.
7. The RX menu in the GUI shown in Figure 8-3 view the received packets from the NRF24 module.
8. The user can change the NRF communication channel from the counter input.



Choose NRF Communication Channel: 100

Current Center Frequency: 2.5G

Figure 8-3 Gnuradio GUI TX menu



Choose NRF Communication Channel: 100

Current Center Frequency: 2.5G

Figure 8-4 Gnuradio GUI RX menu