Final FSK Project Report

ECE 447: Communications

December 7, 2024

C1C Ben Cometto

Contents

List of Figures

List of Tables

Abstract

Digital modulation is an important aspect of modern communication systems. A simple communication scheme is binary frequency shift keying (BFSK). Here, a system is developed to transmit and receive ASCII characters using BFSK modulation. A complex part of this process is the decoding of the bit signal. The final implementation uses GNU Radio and a Python script. Future development could involve making the system less reliant on perfect data and resilient to noise, and integrating a user interface system.

1 Introduction

In today's world, digital devices are everywhere. Not only are they extremely widespread, modern society relies on constant communication between these digital devices. A simple method of enabling communication with a digital modulation scheme is frequency shift keying (FSK).

FSK works, as the name implies, by encoding symbols with different frequencies. In our case, we are using binary FSK (BFSK) with only two symbols: one for each bit, 0 ("space") or 1 ("mark"). We can control the frequencies assigned to each, the sample rate on either end, and the baud (symbol, and in this case, bit) rate.

A method of taking an ASCII character, encoding it in bits, modulating the bits with BFSK, simulating a perfect transmission, then demodulating the signal back to bits has been previously developed. This development completes the communication system: converting the sampled demodulated signal into ASCII characters.

2 Frequency Shift Keying Modulation and Demodulation

Our BFSK modulation and demodulation scheme is implemented according the the GNU Radio flow graph in Figure 1,

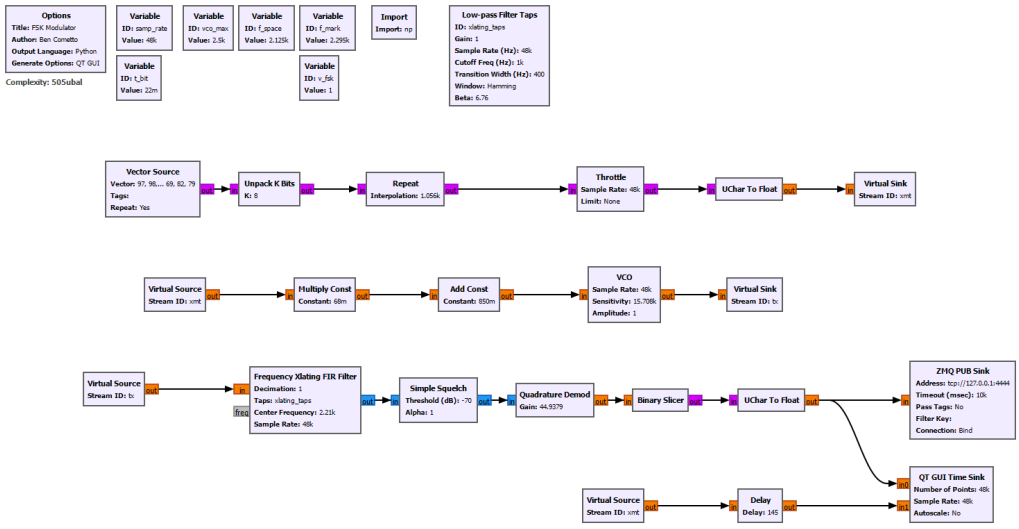

Figure 1: Full BFSK Mod/Demod Flowgraph

First, we will discuss the parameters. In this flow graph, the sample rate on both the transmitting and receiving side is the same. Then, we have a hardware parameter for the voltage controlled oscillator (VCO), which determines the highest frequency that is usable. Additionally, we can set the baud rate (reciprocal of the bit period `t_bit`), and the frequencies `f_mark`, to which a 1 is encoded, and `f_space`, to which a 0 is encoded. Also available is the signal peak voltage `v_fsk`. Finally, we are able to set the message to be transmitted, in terms of the decimal values of the desired ASCII characters. These parameters are summarized in Table 1.

Parameter	Example Value	Description
`samp_rate`	48 kHz	Sample rate of the transmitting and receiving system
`vco_max`	2.5 kHz	Maximum frequency of the voltage controlled oscillator
`f_space`	2.125 kHz	Frequency to which a 0 is encoded
`f_mark`	2.295 kHz	Frequency to which a 1 is encoded
`v_fsk`	1 V	Peak voltage of transmitted signal
`t_bit`	22 ms	Bit period, which is $1/$`baud_rate`

Table 1: Summary of BFSK Parameters

Now, having defined the necessary parameters, we can implement BFSK. First, in GNU Radio, we must convert our ASCII (decimal) values to a signal. First, we unpack the decimal value into a byte (recalling that standard ASCII only uses 7 bits), then repeat each bit for the correct number of samples, found by (`samp_rate`)(`t_bit`). This yields the signal shown in Figure 2.

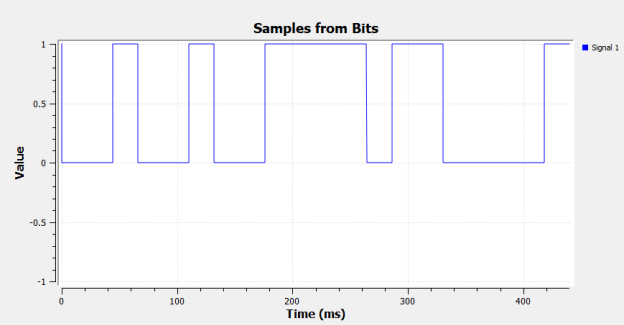

Figure 2: Bits Converted to Signal

This step corresponds to the first line of the flow graph above, and shown in Figure 3.

Figure 3: Flow Graph converting Bits to Signal

The next step is to modulate the signal containing the bits using BFSK. This is done using a voltage controlled oscillator (VCO). The VCO outputs a frequency where an input of 0 V is mapped to 0 Hz, and an input of 1 V is mapped to `vco_max`. Thus, we need to perform a transform on our bit signal, so that our 0 V components of the bit signal are mapped to `f_space` and our 1 V components are mapped to `f_mark`. We can do this using a system of equations, which yields

$$v_{\text{VCO,in}} = \frac{\texttt{f\_mark} - \texttt{f\_space}}{\texttt{vco\_max}}(v_{\text{bit signal}}\frac{\texttt{f\_space}}{\texttt{vco\_max}}).$$

This step is implemented in the second line of the flow graph, which can be seen in Figure 4.

Figure 4: Flow Graph Modulating Signal to BFSK

The result is a successfully modulated BFSK signal. Notice the peaks at `f_space` and `f_mark` in Figure 5

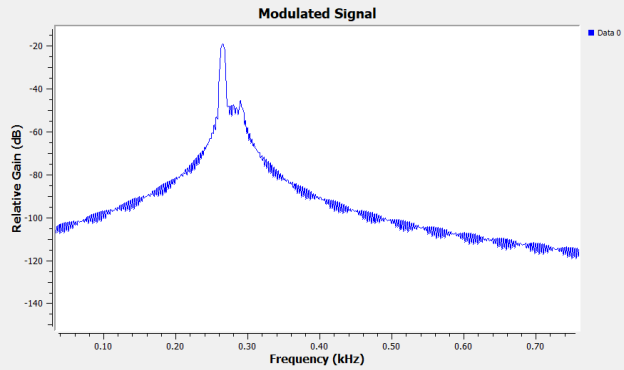

Figure 5: FFT Result of Modulated Signal

Now, we will turn towards demodulation. In order to demodulate, we first translate the two frequencies to be centered around 0 Hz, and low pass filter beyond our two frequencies. Note that this results in a complex signal. The frequency plot of this shifted signal can be seen in Figure 6.

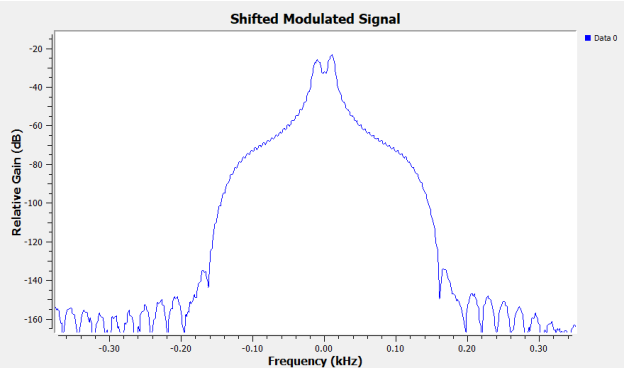

Figure 6: FFT Result of Shifted Modulated Signal

Implementing a squelch reduces the risk of noise, which is not necessary in this noiseless proof of concept. Then, we are able to extract the envelope of the complex signal using quadrature demodulation, which can be seen in Figure 7.

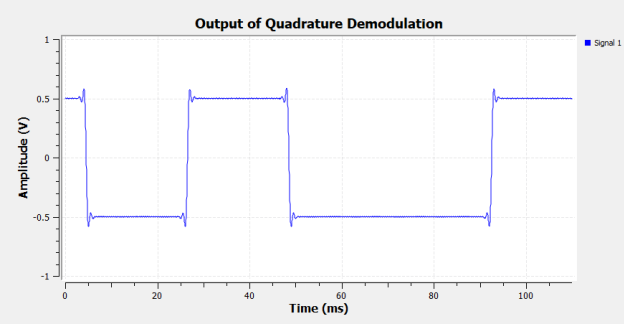

Figure 7: Quadrature Demodulated Signal

Finally, we are able to use a binary slicer to assign samples above zero to a bit of 1 and samples below zero to a bit of 0, as can be seen in Figure 8.

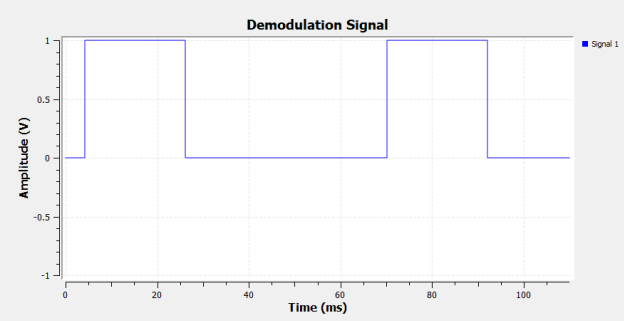

Figure 8: Demodulated Signal

The demodulation process is performed in the final line of the flow graph, and can be seen in Figure 9.

Figure 9: Flow Graph Demodulating Signal

Thus, the final demodulated output is a series of samples, assigned as either 0 or 1.

3 Decoding

The final step in this communication scheme is decoding the received samples. We can recognize that, barring any issues due to noise or channel characteristics, there should be (`t_bit`)(`sample_rate`) samples per bit. In our case, we have no noise, and are decoding a perfect data stream. Thus, we must develop a system that will be able to count samples, then log bits into bytes, and each stored byte can become an ASCII character.

3.1 Test Data

Before discussing the implementation, we will first characterize our provided test data, a recording of a transmitted message. First, we know that the baud rate is 62.5 symbols per second, which means we have that `t_bit` = 16 ms. With a sample rate of 48 kHz, we thus have 768 samples per bit. To "receive" the test data, we will use the simplified receiving flow graph in Figure 10.

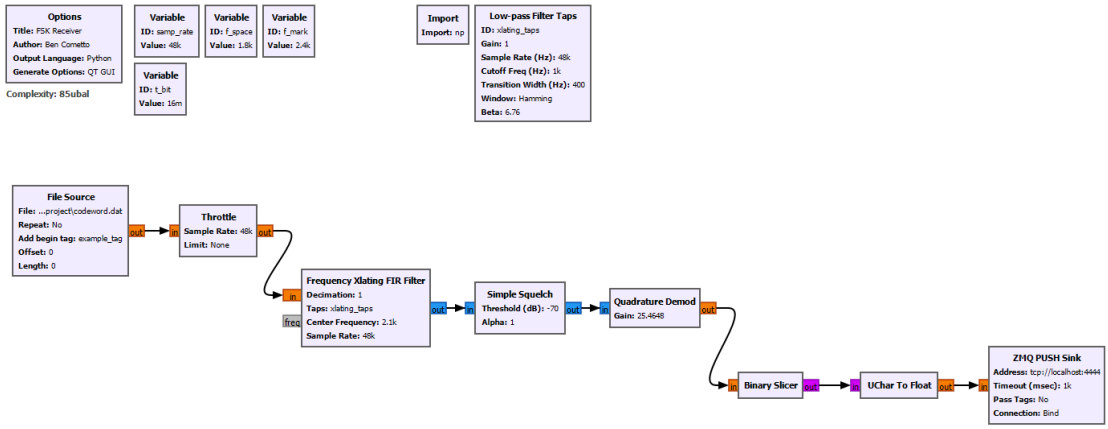

Figure 10: Demodulation-Specific Flow Graph

Then, looking in the frequency domain as in Figure 11, we can see that `f_space` is at 1.8 kHz, and `f_mark` is at 2.4 kHz.

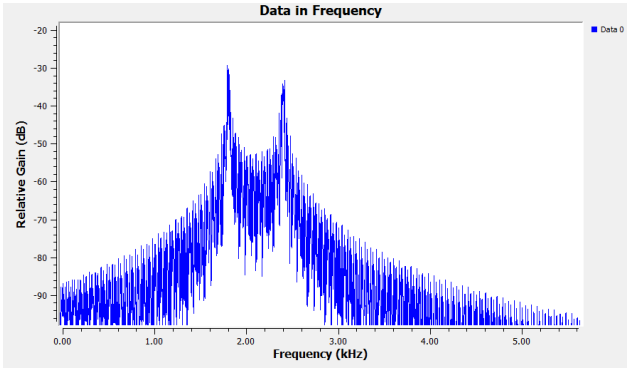

Figure 11: Test Data in Frequency Domain

3.2 Implementation

As counting samples will more easily be done directly in Python, our first task is to transfer the samples from GNU Radio to our Python script in real-time. The tool we will use for this is ZeroMQ, a network/messaging library. In this situation, we are using the push/pull or "pipeline" architecture, as our communication is one-directional. (Note that I wrote a custom ZeroMQ system, rather than directly using the provided example functions.)

Once the data is in our Python script, we are able to begin processing it. In general, we will count samples until we reach 768. Then, we will add the appropriate bit to our bit queue. When there are enough bits in the bit queue, we will convert them into a byte, which is converted into a character using ASCII encoding. Finally, we will output the character.

To begin, we need a method to ensure our counter was aligned with each bit. Here, this process is simplified due to the perfect nature of the data. We also need a method of throwing out outlier samples. In our demodulation system, the GNU radio quadrature demodulation block outputs incorrect values when at the beginning of the file, making it important to ignore outliers even with the prefect data. This can be seen in Figure 12.

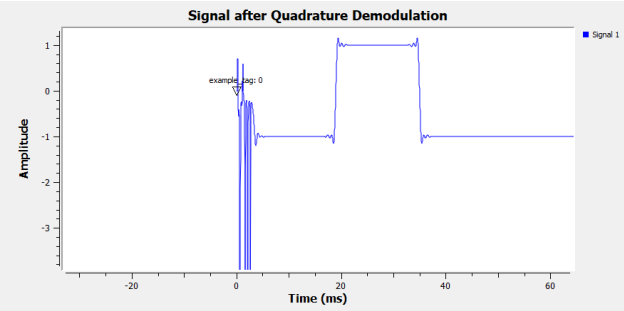

Figure 12: Early Output of Quadrature Demodulation

To do this, we have implemented a tolerance `tol_other` = 30. When we count a sample that is not what the script believes is the current bit, it increases the count `count_other`. When we count `tol_other` number of samples of the other value, the script decides to consider that bit to be the current bit.

With these strategies, the script accurately counts samples and retrieves bits. The next two steps are relatively straightforward. First, when enough bits (at least 8) have been recorded, the script combines them into a single byte, converts the byte to an integer, and then outputs the corresponding ASCII character.

3.3 Results

The script is able to accurately recover the encoded message in both test data files. The encoded messages can be seen in Table 2.

File	Message
`codeword.dat`	`ALLYOURBASEAREBELONGTODFECSOMEONESETUSUPTHEBOMB`
`nocodeword.dat`	`ALLYOURBASEAREBELONGTOUSSOMEONESETUSUPTHEBOMB`

Table 2: Message Results

Notice, then, that that data is a quote from a Japanese game titled *Zero Wing* that was translated into English poorly. One quote is, "All your base are belong to us," and the other is, "Someone set us up the bomb." (Note that, according to Wikipedia, the true second quote would be "Someone set up us the bomb.") The codeword version of the data replaces "us" with "DFEC."

For details on the Python implementation, see the attached code in Appendix A. For access to the project files, see the GitHub repository at `https://github.com/dbcometto/447_finalproject`.

4 Conclusion

Using GNU Radio and ZeroMQ to communicate with a separate Python script, we have developed a system to demodulate and decode ASCII characters modulated via Binary Frequency Shift Keying (BFSK). When combined with the modulation flow graph, we have thus created an entire digital modulation system. With minor modification, this sytem can be applied to a myriad of other problems.

If this system is to be used in an environment without perfect data transmission, the system would need to be able to find the beginning of each bit, and line up the bits inside each byte correctly. To solve the first problem, the counting scheme would need to be upgraded. To solve the second problem, overhead would likely need to be added to the message. Additionally, a user interface should be developed for the script.

Overall, this project provided a strong introduction to digital modulation, specifically BFSK. Beyond this, it enabled me to learn how to work with ZeroMQ, a useful tool for allowing multiple scripts/devices to communicate using sockets. Finally, I was able to greatly expand my familiarity with GNU Radio, and gain experience working with communication systems.

Documentation

Throughout this project, I did my own work. I used the ZeroMQ documentation, StackOverflow, and ChatGPT (contact me for access to the transcript, I unfortunately cannot generate a link) to gather information on syntax, capabilities, and function use, and to assist in troubleshooting. I also used StackExchange for Latex help. Additionally, I used Wikipedia to find the quote, and to better understand FSK. In class, I briefly discussed the project with C1C Stoup, who suggested that I look in the frequency domain to find a change in `f_mark` and `f_space`. I also briefly discussed the project with C1C Coleman, who pointed out that there was a change in the middle of `codeword.dat` as compared to `nocodeword.dat`, whereas I had only compared the beginning and end. I also discussed the length of my files with C1C Coleman. I also briefly discussed the project with C1C Blouin, who confirmed he was also dropping a character at the end, and receiving a handful of ones at the beginning. Throughout, I did my own work.

A Contents of `fskDecode_final.py`

```python
# This is the python script that will decode the FSK ascii data using push/pull
#
#
# Written by Ben Cometto, 7 Dec 2024
#
#
#

import zmq
import struct
import sys

# config
debugging = False
debugging2 = False
verbose = False

# variables
socket_loc = "tcp://localhost:4444"
samp_rate = 48000
t_bit = 0.016
sps = int(samp_rate*t_bit)
tol_other = 30 # number of acceptable sample differences before switching majority sample

count = 0
count_other = 0
current_bit = 0
saved_bits = []
current_data = b""


#  Set up socket to talk to GNU Radio
context = zmq.Context()
pull_socket = context.socket(zmq.PULL)
pull_socket.connect(socket_loc)

print(f" I am setup and waiting for messages on {socket_loc} with {sps} SPS")


# Main Processing Loop
while True:

    # Get new data
    try:
        raw_data = pull_socket.recv(flags=zmq.NOBLOCK)
        current_data += raw_data
    except:
        pass

    # If there is new data to process, count samples
    while len(current_data) >= 4:

        # pull float out of byte data and convert to int
        bit = int(struct.unpack('f', current_data[:4])[0])
        current_data = current_data[4:] # delete data off of the front of the queue
        count += 1

        if (current_bit != bit): # handle non-current sample value
            count_other += 1
            if debugging:
                print(f"Sample not counted!  count_other: {count_other} at count: {count}")

        if (count_other >= tol_other): # decide to switch current sample value
```

```
            if debugging2:
                print(f"Switching to {current_bit} from {bit} at count: {count}", end = "")

            current_bit = bit
            count_other = 0

            if debugging2:
                print(f"and newcount: {count}")


        elif (count >= sps): # log a bit
            if debugging2:
                print(f"Bit counted at count: {count}!",end="")
            count = 0
            count_other = 0
            saved_bits.append(current_bit)
            if debugging2:
                print(f"  Now we're at {saved_bits}")




    # Deal with collected bits
    while len(saved_bits) >= 8:

        binary_list = [str(i) for i in saved_bits[:8]] # combine 8 bits into a byte
        saved_bits = saved_bits[8:] # reindex

        byte_binary = "".join(binary_list) # combine bit strings
        if debugging:
            print(f"Received Byte: {byte_binary}")

        myint = int(byte_binary,2) # convert string to char
        mychr = chr(myint)


        if verbose: # display the character
            print(f"Received Char: {mychr}")
        else:
            print(mychr, end="")
            sys.stdout.flush()



```