433 MHz On-Off Keying One-Way Radio Communication System

By Anthony Faubert

Abstract

For my Wireless Communication class final project, I chose to design a one-way radio communication system. I used a 433 MHz ASK transmitter module for the carrier signal, an Arduino to encode and modulate messages into the ASK transmitter, and a Software Defined Radio to receive the raw radio signal.
Over the course of the project, I wrote code for the Arduino, developed a filtering architecture in Gnuradio Companion, and wrote Python code to analyze and decode the filtered data to recover the messages that were in the Arduino.
Once I was successfully passing messages through my system, I optimized the design and got reliable data recovery at roughly 1 kbps at close ranges.

Introduction

The purpose of the final project in *CSE 490W: Wireless Communications* is to get hands-on experience using a Software Defined Radio (SDR) to apply concepts we learned in the class. For my final project, I chose to create a data transmission system using a cheap 433 MHz transmitter module and the SDR receiver we had been using in the class. I chose that module because I had bought it in a set with a receiver, but I had never figured out how to get the receiver to work properly.
As an aside, this class gave me some insight into why I never got that receiver to work. The receiver almost certainly requires some sort of preamble in order to differentiate the signal from noise, and I never used any kind of preamble when I tried to get it to work.
My initial goal for the project was to create a transmitter that would repeatedly send the message “Hello, world!”, and then develop code to use the SDR to recover that message. My end goal was to develop a transmitter that could send arbitrary messages and then reliably recover the messages with the SDR, at the highest data rate I could manage.

Hardware

I chose my hardware at the beginning and made very few changes over the course of the project. The final hardware setup is shown in Figure 1 (below).

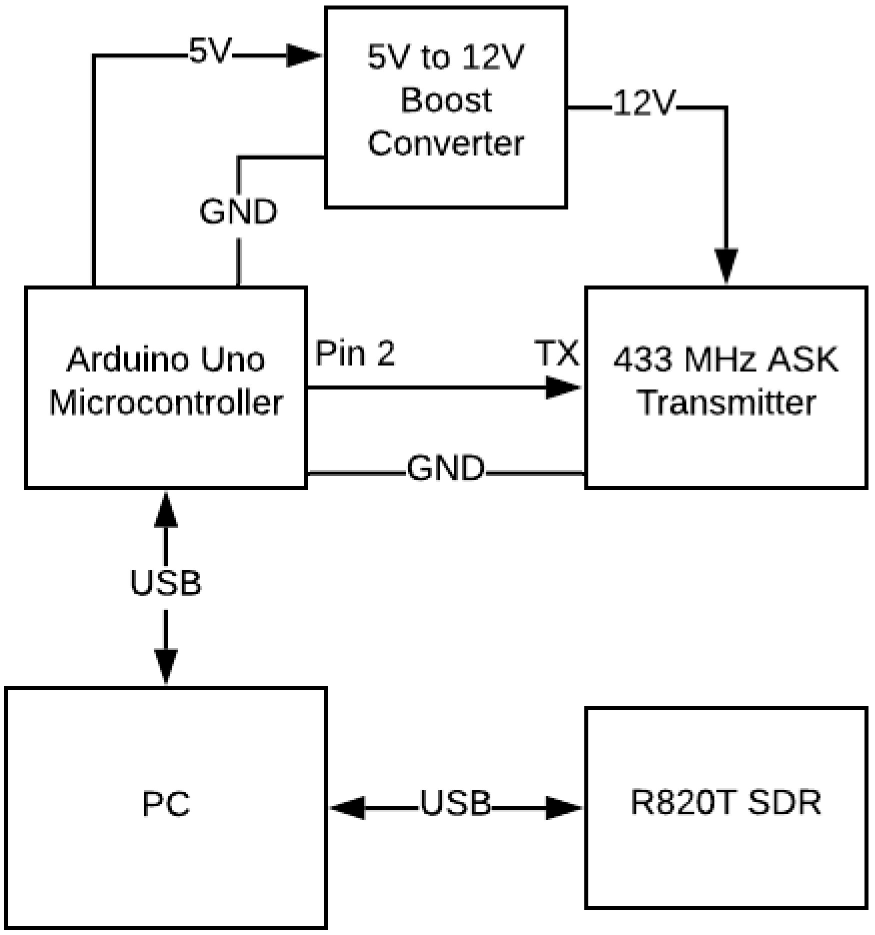


Figure 1: Block diagram of hardware connections.

Initially, I was using an STM microcontroller instead of the Arduino Uno, but I switched to the Arduino because I was having difficulty getting the serial port on the STM working. Originally, I was powering the transmitter off of the microcontroller's 5V, but I later added the boost converter to take advantage of the transmitter's maximum transmit power.
The exact parts used were as follows:

- Arduino uno: Elegoo Uno R3
- Boost converter: XTW-SY-8 Boost Converter
- SDR: Nooelec NESDR RTL2832+R820T with a ~14cm monopole antenna
- Transmitter: XD-FST FS1000A (equivalent: https://www.ebay.com/itm/254431683926) with a hand-made ~34.5cm monopole antenna

Developing the Receiver

In the class, we used the Gnuradio Companion (GRC) software to gather data with the SDR. GRC is a program where you create a flow chart that connects the SDR to various existing filter blocks to process and view the data that the SDR captures.

The first test I did with GRC was to see if I could visually detect the transmitter being on or not. I made a GRC graph with the SDR tuned to 433 MHz and connected the output of the SDR to a waterfall display block (frequency components vs time graph) and was able to see the carrier frequency appear and disappear when I put power or ground on the transmitter signal pin. I also noticed that the carrier frequency changed substantially when I moved my hands anywhere near the transmitter.

The unstable carrier frequency was a problem for my initial plan to simply demodulate the received signal. Since the carrier frequency was dynamic, if I wanted to demodulate by multiplying by a sinusoid I'd have to constantly adjust the frequency of the sinusoid to match the carrier frequency. I thought about measuring the carrier frequency and adjusting the demodulation frequency on the fly, but I gave up on that when I found out I couldn't make my own custom GRC blocks.

My original plan for decoding was to write a GRC embedded Python block that would process the data and recover messages in real-time, but almost anything I wrote would cause GRC to freeze, so I gave up on real-time processing.

I didn't see a straightforward way to measure the carrier frequency using the built-in GRC blocks, so I needed a new demodulation/filtering scheme. Early on, I came up with the idea to compute the FFT of the signal and then threshold any frequency component inside a certain range of frequencies, like I did with my eyes while looking at the waterfall plot, but I never put that plan into action. Figure 2 (below), illustrates that idea.

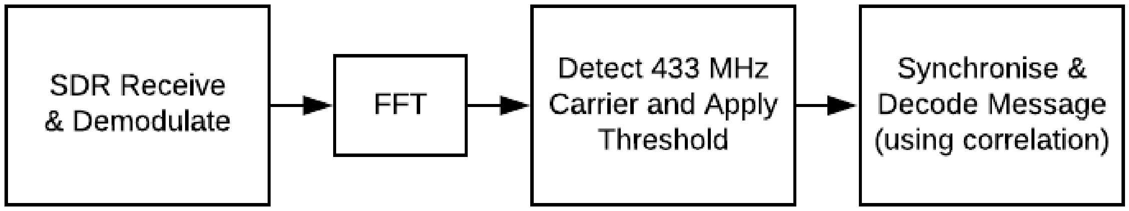

SDR Receive & Demodulate
FFT
Detect 433 MHz Carrier and Apply Threshold
Synchronise & Decode Message (using correlation)

Figure 2: Block diagram of my initial demodulation/filtering goal architecture from my project proposal slides.

I figured there might already be a GRC block that would do a good job of demodulating my signal, so I looked through the block list and found an AM Demodulation block. I figured that would work for my application because On-Off Keying (OOK) is technically a type of Amplitude Modulation (AM). That AM block worked reasonably well, and I developed most of my decoding code with the AM Demod block as my filter, but it had a built-in Low-Pass Filter (LPF), it added an offset to the signal, and it didn't produce clean rectangular pulses even with the transmitter and receiver right next to each other, so I wanted a better solution that I could fully understand. At one point I ran into DC bias issues with the signal decoding and tried to remove the offset by high-pass filtering the AM Demodulation output, but that wasn't really any better than the AM Demodulation output by itself, and I ended up solving the DC bias problem by changing the coding scheme, which I will discuss later.

I had another issue with my system as it stood with the AM Demodulation block. I needed to reduce the data rate from the SDR's 1024 kHz sample rate down to something more reasonable (128-64 kHz) so that my slow decoder wouldn't have to process more samples than it needed to. I started by simply throwing out all but 1 out of every 16 samples to bring the rate down to 64 kHz, but the professor suggested that I would likely get better results by taking the average of those 16 samples instead of throwing out 15/16ths of my data, so I implemented that.

Later on, I was thinking about better filters and realized that my averaging downsampler was roughly 1 operation away from being an envelope follower. One method of envelope following is to take the average power over a certain window of signal, and then consider that average power to be the envelope of the signal. Consider the equation for the average power of a discrete signal:

$$P_{avg,N} = \frac{1}{2N+1} \sum_{n=-N}^{N} |x[n]|^2$$

My averaging filter is effectively computing this: $Y = \frac{1}{N} \sum_{n=0}^{N-1} x[n]$ (with N=16), which is equivalent to the $\frac{1}{2N+1} \sum_{n=-N}^{N}$ part of the average power equation. The only thing left is to compute the average of $|x[n]|^2$ instead of $x[n]$, which is easy because $|x[n]|^2 = x[n]x^*[n]$, where $x^*[n]$ is the complex conjugate of $x[n]$. So I simply added a multiply-by-conjugate block that took in the signal from the SDR as both inputs, computing $x[n]x^*[n] = |x[n]|^2$ and then fed that into my averaging filter, and tossed out the AM Demodulation block. The result was an enormous improvement in the signal quality over the AM Demodulation scheme, as you can see in Figures 3 and 4 (below).

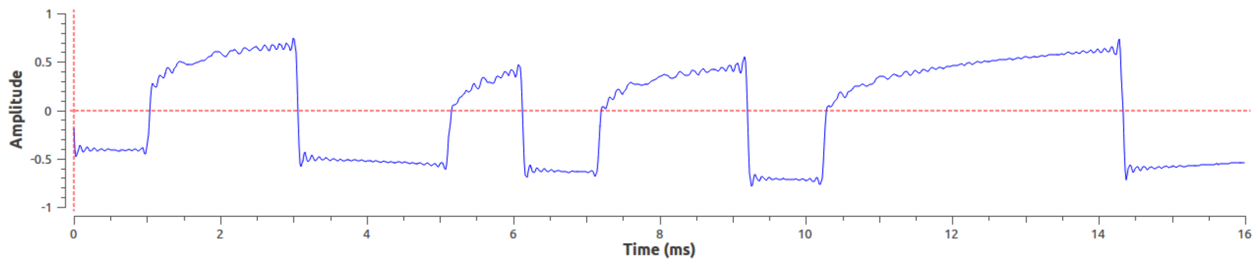


Figure 3: Time-domain plot of the SDR -> averaging filter downsampling -> AM Demodulation block -> high-pass filter block -> signal.

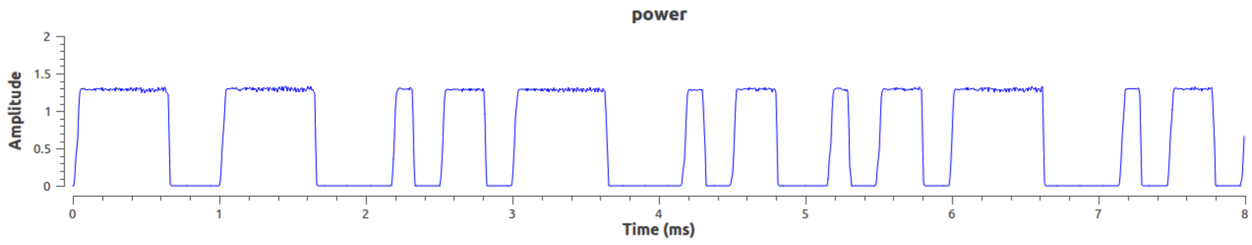
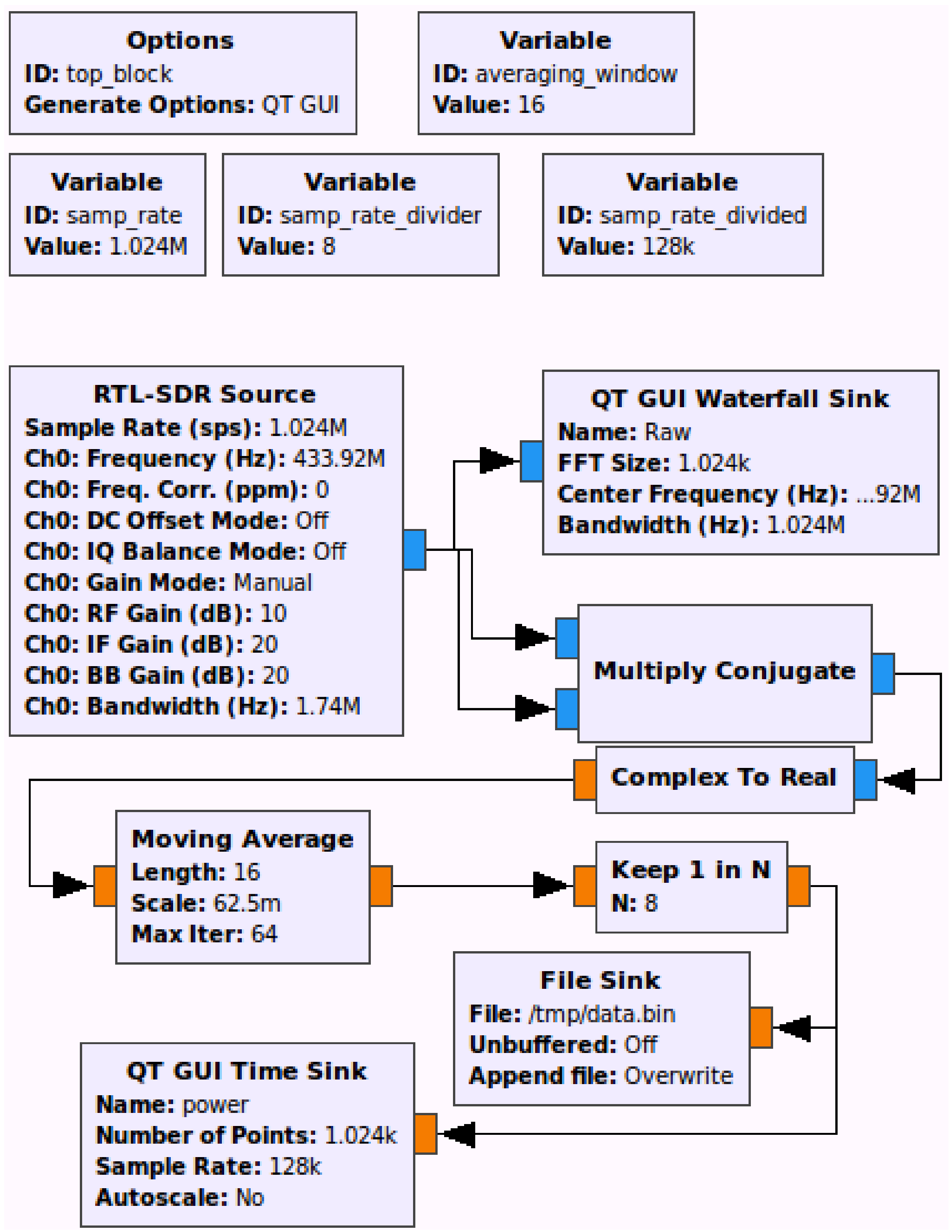


Figure 4: Time-domain plot of the SDR -> square magnitude -> averaging filter downsample -> signal. (envelope follower)

The output from the envelope follower (Figure 4) is clearly much closer to the original rectangular pulse signal from the Arduino than the AM Demodulation output (Figure 3). The results are so good that it almost looks like a signal straight off an oscilloscope.

When I had my decoder working, I played with the sample rate of the SDR and the parameters of my envelope follower to optimize my filtering. Surprisingly, I found that increasing the sample rate of the SDR substantially impeded the decoder's ability to recover messages, even when I decoupled the SDR sample rate and the averaging filter window size. Eventually, I settled on the lowest available SDR sample rate (1024 kHz) and downsampled it to 128 kHz with the envelope follower.

My final GRC graph is shown in Figure 5 (below, on the next page). Note that the Multiply Conjugate block computes $y = ab^*$, where a is the top input, b is the bottom input, and y is the output. The Complex To Real block simply throws away the (now zeroed) imaginary component to increase computational efficiency and simplify the contents of the file it records to. The QT GUI blocks are plots that are updated in real-time and help with debugging. The File Sink block takes all data sent to it and writes it to a file so that I can load it into my decoder program for message recovery.


Options
ID: top_block
Generate Options: QT GUI

Variable
ID: averaging_window
Value: 16

Variable
ID: samp_rate
Value: 1.024M

Variable
ID: samp_rate_divider
Value: 8

Variable
ID: samp_rate_divided
Value: 128k

RTL-SDR Source
Sample Rate (sps): 1.024M
Ch0: Frequency (Hz): 433.92M
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20
Ch0: Bandwidth (Hz): 1.74M

QT GUI Waterfall Sink
Name: Raw
FFT Size: 1.024k
Center Frequency (Hz): ...92M
Bandwidth (Hz): 1.024M

Multiply Conjugate

Complex To Real

Moving Average
Length: 16
Scale: 62.5m
Max Iter: 64

Keep 1 in N
N: 8

File Sink
File: /tmp/data.bin
Unbuffered: Off
Append file: Overwrite

QT GUI Time Sink
Name: power
Number of Points: 1.024k
Sample Rate: 128k
Autoscale: No

Figure 5: My final Gnuradio Companion flow chart for filtering and recording signals to be processed by the decoder.

Protocol and Decoder

The first encoding scheme I came up with was to repeatedly send a sync pulse, “Hello, world!” with each bit being either a bit-1 symbol or a bit-0 symbol, and a bit-end symbol. I chose the symbols as follows:

- Sync = “0000 1111 0000”
- Bit ‘1’ = “1100”
- Bit ‘0’ = “1010”
- End = “0000”

Where “0” is 1 ms of the transmitter being on and “0” is 1 ms of the transmitter being off. I chose those bit and end symbols because they were distinct and would still be distinct if every “1” was a “0” and vice-versa. I call this being “polarity-agnostic”. There really was no need for it to be polarity agnostic, but this early in the project didn’t think it would hurt anything, and it seemed like a potentially useful property for the coding scheme to have, especially while I was still developing the receiver and didn’t know what the filtered signal was going to look like, exactly.

My method of decoding was to find the sync pulse, align to it, and then advance to the first bit symbol in the message. Then I would compute the absolute value of the correlation of the normalized signal with the bit ‘1’ symbol, the bit ‘0’ symbol, and the end symbol, and whichever had the highest correlation was the winner. If a bit symbol was the winner, the corresponding bit would be appended to the message. If the end symbol was the winner, the message would be processed and then it would start looking for a sync pulse. Here is the algorithm in pseudocode:

```
while (enough data left):
    while (abs(correlate(normalize(first 12 ms of data),
syncSymbol)) < threshold):
        delete a sample from the start;
    delete a sample from the start until
(abs(correlate(normalize(first 12 ms of data), syncSymbol))) starts
to decrease; # find the best alignment
    # Now we have synchronized to the sync pulse
    delete the first 12 ms of data; # jump to the first bit symbol
    forever:
        winningSymbol = bestCorrelatingSymbol(normalize(first 4 ms
of data), [bit1Symbol, bit0Symbol, endSymbol]);
        delete the first 4 ms of data;
        if winningSymbol == endSymbol:
            processMessage();
            break from forever loop;
        else if winningSymbol == bit1Symbol:
            appendMessage(1);
        else:
            appendMessage(0);
```

Figure 6: Pseudocode for the first iteration of my decoding algorithm

The problem with that algorithm was that when normalizing the signal, I didn’t remove any DC bias, which caused the correlation to always favor the end symbol because it had a large DC offset. While trying to deal with this, I figured I might try removing the DC offset from the signal by high-pass filtering it, but that didn’t go well and I realized that my coding scheme needed to change to deal with this problem. So I developed a new coding scheme:

- Sync = “111000 10010” (11-bit Barker code)
- Bit ‘1’ = “011110”
- Bit ‘0’ = “001011”

Note that I was still trying to make the encoding polarity-agnostic (for no real reason), and I was thinking more about reliability than throughput, so I increased the bit symbol length from 4 ms to 6 ms. The Sync pulse is now a Barker code, which has very low correlation with shifted versions of itself, which may or may not be useful for this application.
The end symbol is also no longer its own symbol. Instead, I just look for the first 6 ms of the sync pulse and double-count that as the end symbol and part of the sync pulse.
I chose the values of the bit symbols by writing a program that would try every possible combination of two patterns of length 6 and compute the correlation with the first 6 ms of the sync pulse and each other. Then from the set of choices that had the lowest correlation to the other bit symbols or the “end symbol” part of the sync pulse, I chose a random set that had a perfect DC balance. Note that the first 6 ms of the sync pulse also has a perfect DC balance.

With this new coding scheme I had solved the DC balance problem that caused me difficulties with correlation. I then had to deal with a new problem: the transmitter’s timing was bad. Instead of fixing the transmitter’s timing, I figured I should just make the receiver immune to poor timing problems. The solution I came up with was instead of having the decoder just append a bit to the message when it detects a symbol, append that bit and then shift left and right by a few data samples until you find the best correlation value, thereby resynchronizing on every bit of the message.
The last change I made to the protocol and decoder was treating the last 16 bits of the message as a CRC-16 checksum and adding some code in the message processor to verify it and report error or successful message recovery. After implementing the new coding scheme and that change, the decoder was complete.

Transmitter

The transmitter code was fairly straightforward. I would have the bit symbols and sync pulse programmed in, have my code walk through the patterns, and just call the delay function to wait for 1 ms for every subsection of a symbol.

To debug my code, I had the code time how long it would take to send 1 repetition of the message, which would take $(12 + 4 \times 13 \times 8 + 4)\ ms = 432$ ms ("Hello, world!" is 13 bytes), and then print the time over the serial port. After fixing some bugs and very carefully going over my code, I still was seeing about 450 ms on the timer instead of the expected 432, so I got suspicious and wrote my own delay1ms() function that would also record how many times it was called. It turns out it was being called 432 times, yet the internal timer was recording ~450 ms.

I suspected there was some sort of interrupt strangeness going on that would be difficult to debug, so it was at this point that I decided to add the timing immunity to my decoder, and I stopped bothering to try and fix the timing issue because the system was reliably getting the data through.

The last changes I made were to add the ability to change the message being sent by sending the new message to the Arduino via the serial port and make the transmitter compute a CRC-16 checksum of the message and append it to the end before sending it.

Note: my transmitter contained a buffer for the message bytes which was 256 bytes long, and because of that, a counter, and how I handled CRCs, the maximum message length is 253 bytes.

Optimization

At this point, my data rate was pathetic. The transmission was very reliable, so I could send long messages with no errors, but even with an infinitely long message, the maximum throughput would only be $\frac{1\ bit}{6\ ms} \approx 0.167\ kbps$, which is abysmal. So I tweaked the receiver parameters and managed to reduce the bit symbol down to 960 μs long (160 μs sub-symbols) instead of 6 ms long while still having reasonable reliability for maximum-length messages.

Results

Longer packets have more bits and therefore are more likely to contain an error, causing them to fail checksum validation. Because of that, I tested my system with a maximum-length message (253 bytes). I received data for roughly 1 minute and then ran the decoder to determine the number of packets that passed or failed checksum verification. There were 22 successful packets

and 5 unsuccessful ones, one of which was at the end of the recording and was likely cut off midway when I stopped recording, leaving the actual success ratio being 22:4, or ~85%.
Next, I tried 64-byte packets. I theorize that the error rate for these should be the error rate of the 253-byte packets scaled down based on their relative sizes. The 253-byte message is actually 255 bytes with the checksum, and the 64-byte message is 66 bytes with the checksum, so the error rate should be roughly $(1 - 0.85)\frac{66}{255} \approx 4\%$, which would give a success rate of about 96%. For the experiment, I set the transmitter to send a 64-byte message repeatedly and recorded data for roughly one minute. This time there were 111 successful packets and no failures. That's substantially better than my prediction, which would have been ~4.5 failed packets out of 111 total. Thinking it through more carefully, if the probability of there being an error in any one single byte is p_e, then the chance of getting 255 bytes through is $(1 - p_e)^{255}$, which we know from experiment is roughly 0.85. We can then derive that the chance of getting 66 bytes through is $(1 - p_e)^{66} = ((1 - p_e)^{255})^{\frac{66}{255}} \approx (0.85)^{\frac{66}{255}} \approx 96\%$, which matches up with my intuition, but not the actual results. Interesting.

Now that we have some numbers for the error rates, we can evaluate the throughput of this system. Here is a derivation of an equation for the throughput vs packet length (the "packet" being the message in between the sync pulse and the checksum at the end):

$$throughput_{max} = \frac{1}{subsymbolPeriod} \frac{packetLen}{syncSymbolLen + (packetLen + checksumLen)dataSymbolLen}$$

$$throughput_{max} = \frac{1}{160\ \mu s} \frac{packetLen}{11 + (6\ bits^{-1})(packetLen + 16\ bits)} = (6250\ Hz)\frac{packetLen}{(6\ bits^{-1})packetLen + 107}$$

So the theoretically highest possible data rate through this system (assuming no errors) would be $\lim_{packetLen \to \infty} throughput_{max} \approx 1.042\ kbps$, and the highest possible data rate given my packet length limit would be $(6250\ Hz)\frac{253 \times 8\ bits}{(6\ bits^{-1})253 \times 8\ bits + 107} \approx 1.033\ kbps$.

Now we can factor in errors:

$$throughput_{avg} = (1 - errorsPerPacket_{avg})throughput_{max} = packetSuccessRate_{avg} \times throughput_{max}$$

$$throughput_{avg} = (1 - errorsPerPacket_{avg})(6250\ Hz)\frac{packetLen}{(6\ bits^{-1})packetLen + 107}$$

So the throughput I was getting with 253-byte packets was $85\% \times 1.033\ kbps \approx 0.878\ kbps$, and the throughput I was getting with 64-byte packets was $100\% \times \frac{(6250\ Hz)64 \times 8\ bits}{(6\ bits^{-1})64 \times 8\ bits + 107} \approx 1.007\ kbps$.
So from this, we see that the throughput will go up if the packet length is long, but the packet length being long makes the probability of there being an error anywhere in the packet larger, therefore decreasing the throughput. This makes the system somewhat adaptable for optimizing throughput depending on how much noise is introduced for a particular application.

I have recorded a demo video of the system available here: https://youtu.be/cRuMV5pi6OY
The code for the project is available in the Appendix.

Conclusion

I got some great experience putting the course content to work and I learned a lot about filtering signals, synchronizing to a data stream, and encoding data. I fulfilled my end goal of the project by creating a one-way wireless communication system that could reliably receive arbitrary messages, and I managed to get the data rate up to about 1 kbps.
The data rate is low by absolute standards, and I know I could improve my coding scheme by reducing the lengths of the symbols, but considering the transmitter module costs ~$2 and is only advertised for 4 kbps in perfect conditions, I am pleased with my throughput results as well.

Thanks to professor Joshua Smith for creating this hands-on assignment for his Wireless Communication course. This system was lots of fun to design, and I learned a lot.

Appendix

This project is available on GitHub (https://github.com/AnthonyFaubert/ee490w_finalproject), and a demo of the project is available on YouTube (https://youtu.be/cRuMV5pi6OY).
However, I have also included the code in this report so that it is possible to replicate the project with the contents of this report directly.

Appendix sections:

Arduino Transmitter Code (C/C++)

```
#include <util/crc16.h>

#define SYNC_PATTERN_LEN 11
uint8_t PATTERN_SYNC[SYNC_PATTERN_LEN] = {1,1,1,0,  0,0,1, 0,0,1, 0};
#define BIT_PATTERN_LEN 6
uint8_t PATTERN_BIT0[BIT_PATTERN_LEN] = {0,0,1,0,1,1};
uint8_t PATTERN_BIT1[BIT_PATTERN_LEN] = {0,1,1,1,1,0};

#define _PRINTLN_SDS(prefix, num, suffix) \
  Serial.print(prefix); \
  Serial.print(num, DEC); \
  Serial.println(suffix);

#define DEFAULT_MSG "Hello, world!"
uint8_t msg[256];
uint8_t msgLen = 13;
int syncPattern = 0;
int bitPattern = 0;
uint16_t datai = 0;

void debugVal(uint8_t val) {
  digitalWrite(3, (val & 0x01) ? HIGH : LOW);
  digitalWrite(4, (val & 0x02) ? HIGH : LOW);
}

void setMsg(char* str) {
  uint16_t crc = 0;
  msgLen = 0;
  for (uint8_t i = 0; str[i] != 0; i++) {
    crc = _crc16_update(crc, str[i]);
    msg[i] = str[i];
    msgLen++;
  }
  msg[msgLen++] = (uint8_t) ((crc & 0xFF00) >> 8);
  msg[msgLen++] = (uint8_t) (crc & 0x00FF);
}

void setup() {
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);

  setMsg(DEFAULT_MSG);

  Serial.begin(9600);
  Serial.println("CSE 490W Final Project transmitter v64. Msg on next line.");
  // Println(msg):
  for (uint8_t i = 0; i < msgLen; i++) Serial.print((char) msg[i]);
  Serial.println();
}

void loop() {
  static int debug = 0;
  static unsigned long t1 = 0, t2 = 0, t3 = 0, t4 = 0;
  static int loops = 0;

  static char nextMsg[256];
  static uint8_t nextMsgLen = 0;
  unsigned long timeout = micros() + 160UL;
```

```
if (syncPattern < SYNC_PATTERN_LEN) { // Do sync pattern
  if (debug == 0) {
    t1 = millis();
    debug++;
  } else if (debug == 3) {
    t4 = millis();
    _PRINTLN_SDS("Started sync @", t1, "ms");
    _PRINTLN_SDS("Started data @", t2, "ms");
    _PRINTLN_SDS("Finished data @", t3, "ms");
    _PRINTLN_SDS("Started next sync @", t4, "ms");
    _PRINTLN_SDS("Looped ", loops, " times");
    debug++;
  }
  debugVal(1);

  digitalWrite(2, PATTERN_SYNC[syncPattern] ? HIGH : LOW); // sync pattern

  syncPattern++;
} else { // Do data after sync pattern
  if (debug == 1) {
    t2 = millis();
    debug++;
  }
  debugVal(0);

  // datai = {13'byteIndex, 3'bitIndex}
  char dataByte = msg[datai >> 3];
  char dataBitMask = 1 << (datai & 0x0007); // 1 << 3'bitIndex

  if (dataByte & dataBitMask) { // bit '1' pattern
    digitalWrite(2, PATTERN_BIT1[bitPattern] ? HIGH : LOW);
  } else { // bit '0' pattern
    digitalWrite(2, PATTERN_BIT0[bitPattern] ? HIGH : LOW);
  }

  bitPattern++;
  if (bitPattern == BIT_PATTERN_LEN) {
    bitPattern = 0;
    datai++;
  }
  if (datai >= ((uint16_t)msgLen << 3)) { // if datai > bytes2bits(MSG_LEN)
    if (debug == 2) {
     t3 = millis();
     debug++;
    }

    datai = 0;
    syncPattern = 0;
  }
}

while (micros() < timeout) { // delay for 1ms from start of function
  if (Serial.available() > 0) { // handle input
    char c = Serial.read();
    if (c == '\n') {
     nextMsg[nextMsgLen] = 0;
     setMsg((char*) nextMsg);
     nextMsgLen = 0;
     if (Serial.availableForWrite() > 4) Serial.println('ACK');
    } else {
     nextMsg[nextMsgLen++] = (uint8_t) c;
    }
  }
}
```

```
  loops++;
}
```

Message Recovery Code (Python) (some commented out code omitted removed for clarity)

```
#!/usr/bin/python3

import matplotlib.pyplot as plt
import numpy as np
import struct, time, os, code

FILE = '/tmp/data.bin'
FLOATS_PER_CHUNK = 256
STRUCT = '<' + 'f'*FLOATS_PER_CHUNK
SAMPLE_RATE = 128e3
BIT_PERIOD = 165e-6
SYNC_THRESHOLD = 0.9
SLIP = 8
# loading several floats at a time is substantially faster
t1 = time.time()
numChunks = int(os.stat(FILE).st_size / 4 / FLOATS_PER_CHUNK)
numFloats = numChunks * FLOATS_PER_CHUNK
vals = np.zeros(numFloats, dtype=np.float32)
f = open(FILE, 'rb')
for i in range(0, numFloats, FLOATS_PER_CHUNK):
    vals[i:i+FLOATS_PER_CHUNK] = struct.unpack(STRUCT, f.read(4 * FLOATS_PER_CHUNK))
f.close()
t2 = time.time()
print('%.03f to decode %d floats' % (t2-t1, numFloats))

STATE_SYNC = 0
STATE_DATA = 1
PATTERN_BIT_0 = [0, 0, 1, 0, 1, 1]
PATTERN_BIT_1 = [0, 1, 1, 1, 1, 0]
PATTERN_SYNC = [1,1,1,0, 0,0,1, 0,0,1, 0] # 11-bit Barker code
PATTERN_BIT_END = PATTERN_SYNC[:len(PATTERN_BIT_0)]
#0001,0010,0100,0111 are good patterns
lens = [len(PATTERN_BIT_0), len(PATTERN_BIT_1), len(PATTERN_BIT_END)]
assert(min(lens) == max(lens))

def crc16(bs):
    crc = 0
    for b in bs:
        crc ^= b
        for i in range(8):
            if crc & 1:
                crc = (crc >> 1) ^ 0xA001
            else:
                crc = crc >> 1
    return crc

def normalize(sig):
    '''Normalize a signal and return it'''
    norm = np.sqrt(np.sum(sig * sig))
    return sig / norm

class DataProccessor:
    def patternToSignal(self, pattern):
        protoSignal = []
        for bit in pattern:
            if bit == 0:
                bit = -1
            protoSignal += [bit] * self.bitLen
        return normalize(np.array(protoSignal, dtype=np.float32))
    def __init__(self, data):
        self.bitLen = int(BIT_PERIOD * SAMPLE_RATE)
        self.thresh = SYNC_THRESHOLD
        self.state = STATE_SYNC

        self.dataIndex = 0
        self.data = data
        #plt.plot(self.data[:20000:10]); plt.show(); quit()

        self.pulseSync = self.patternToSignal(PATTERN_SYNC)
        self.syncLen = len(self.pulseSync)
        self.pulseBit0 = self.patternToSignal(PATTERN_BIT_0)
        self.bitSymbolLen = len(self.pulseBit0)
        self.pulseBit1 = self.patternToSignal(PATTERN_BIT_1)
        self.pulseDesync = self.patternToSignal(PATTERN_BIT_END)

        self.message = ''
        self.nextPrint = 0
        self.debug = True
        self.sdbg_mc = 0
        self.messageCount = 0
        self.messageCountGood = 0

    def messageReadByte(self):
        if len(self.message) < 8:
            return b''
        b = 0
        for i in range(8):
            b |= int(self.message[i]) << i
        self.message = self.message[8:]
        return bytes([b])
    def doMessage(self):
        now = time.time()
        rawMsg = self.message
        bs = b''
        while True:
            b = self.messageReadByte()
            if len(b) == 0:
                break
            bs += b
        crcGood = False
        if len(bs) > 2:
            crcActual = crc16(bs[:-2])
            crcExpected = (bs[-2] << 8) | bs[-1]
            if crcActual == crcExpected:
                crcGood = True
            else:
                print("CRCs don't match!", crcActual, crcExpected)
        if crcGood:
            print('Good message:', bs[:-2])
            self.messageCountGood += 1
        else:
            print('Bad message:', bs)
            if len(self.message) > 0:
                print('Raw message:', rawMsg)
                print('Orphan bits:', self.message)
        self.message = ''
            self.messageCount += 1


    def dataLeft(self):
        return len(self.data) - self.dataIndex
    def dataWindow(self):
        if self.state == STATE_SYNC:
            sigLen = self.syncLen
        elif self.state == STATE_DATA:
            sigLen = self.bitSymbolLen
        window = self.data[self.dataIndex:self.dataIndex+sigLen]
        return normalize(window - np.average(window))

    def doSyncState(self):
        while self.dataLeft() > self.syncLen:
            correlation = np.abs(np.dot(self.dataWindow(), self.pulseSync))
            if correlation > self.sdbg_mc:
                self.sdbg_mc = correlation
                self.sdbg_i = self.dataIndex
            if correlation > self.thresh:
                if self.debug:
                    plt.plot(self.dataWindow());plt.plot(self.pulseSync)
                    plt.title('Sync threshold');plt.show()
                    print('Sync detected. Finding best sync...')
                while True:
                    self.dataIndex += 1
                    correlation2 = np.abs(np.dot(self.dataWindow(), self.pulseSync))
                    if correlation2 > correlation:
                        correlation = correlation2
                    else:
                        self.dataIndex -= 1
                        if self.debug:
                            plt.plot(self.dataWindow());plt.plot(self.pulseSync)
                            plt.title('Best sync');plt.show()
                        self.dataIndex += self.syncLen
                        self.state = STATE_DATA
                        return False # not out of data
            else:
                self.dataIndex += 1
        return True # out of data

    def doDataState(self):
        while self.dataLeft() > self.bitSymbolLen:
            signal = self.dataWindow()
            corBit0 = np.abs(np.dot(self.pulseBit0, signal))
            corBit1 = np.abs(np.dot(self.pulseBit1, signal))
            corEnd = np.abs(np.dot(self.pulseDesync, signal))
            corMax = max(corBit0, corBit1, corEnd)

            if corEnd == corMax:
                if self.debug:
                    plt.plot(self.dataWindow());plt.plot(self.pulseDesync)
                    plt.title('Desync');plt.show()
                    self.debug = False
                # finished message
                self.state = STATE_SYNC
                self.doMessage()
                return False # not out of data

            if corBit0 == corMax:
                self.message += '0'
                pulse = self.pulseBit0
            else:
                self.message += '1'
                pulse = self.pulseBit1
            # Go SLIP samples to the left and right searching for the best match for this bit
            bestCor = -1
            bestIndex = -1
            # FIXME: never checked to see if we have enough data to slip forward
            for i in range(self.dataIndex - SLIP, self.dataIndex + SLIP +1):
                self.dataIndex = i
                cor = np.abs(np.dot(self.dataWindow(), pulse))
                if cor > bestCor:
                    bestCor = cor
                    bestIndex = i
            self.dataIndex = bestIndex + self.bitSymbolLen
        return True # out of data

    def work(self):
        outOfData = False
        while not outOfData:
            if self.state == STATE_SYNC:
                outOfData = self.doSyncState()
            elif self.state == STATE_DATA:
                outOfData = self.doDataState()
        print('Decoded %d messages, of which %d were good.' \
                        % (self.messageCount, self.messageCountGood))

proc = DataProccessor(vals)
proc.work()
quit()

# sync = 3*4ms/bit = 12ms
# data = 13bytes * 8bits/byte * 4ms/bit
# end = 1 * 4ms/bit
# total 432ms: ~27.6k samples @ 64kHz
maxI = int(28e3)
step = 16
ms = np.linspace(0, maxI / 64.0, maxI * 1.0 / step)
Y = vals[:maxI:step]
yfill = 0.0
plt.plot(ms, Y)
plt.fill_between(ms, Y, y2=yfill, color='blue', alpha=0.4)
plt.xlabel('time (ms)')
plt.title('Initial chunk of raw data')
plt.tight_layout()

plt.figure()
Z = np.zeros(len(Y))
Z[np.where(Y > 0.5)] = 1
plt.plot(ms, Z)
plt.fill_between(ms, Z, y2=0, color='r', alpha=0.3)
plt.xlabel('time (ms)')
plt.ylim([-0.05, 1.05])
plt.title('Thresholded data')
plt.tight_layout()

plt.show()
```