William Kenji Kiplinger
https://github.com/Kenjum/CS380-P2/

_____PhysLayerClient.java_____

```
import java.net.Socket;
import java.io.InputStream;
import java.io.OutputStream;
import javax.xml.bind.DatatypeConverter;
import java.util.Hashtable;

public class PhysLayerClient {
	public static void main(String[] args) throws Exception {
		try (Socket socket = new Socket("18.221.102.182", 38002)) {
			System.out.println("Connected to server.");

			InputStream is = socket.getInputStream();
			OutputStream os = socket.getOutputStream();

			int[] getFromServer = new int[384];
			double preamble = 0;

			// The if statement builds up the preamble for the first 64 signals.
			// Everything else is caught
			// and stored in getFromServer[i]. The Mentioned below is the math
			// of bytes needed. 384
			// comes from the message (320 bytes) and preamble (64 bytes).
			for (int i = 0; i < getFromServer.length; i++) {
				getFromServer[i] = is.read();
				if (i < 64)
					preamble += getFromServer[i];

			}

			// Initially you will get a 64 bit preamble this will be sent to you
			// as 64 unsigned bytes in
			// this simulation that you need to add up and divide by 64 to find
			// a baseline.
			preamble = preamble / 64;
			System.out.printf("Baseline established from preamble: %.2f\n", preamble);

			// I receive 32 bytes of data --> 32 bytes = 32*8 bits = 256 bits.
			// These bits are encoded using
			// 4b/5b hence I will receive (256/4)*5 = 320 bits. In this
			// simulation every bit is
			// sent as an unsigned byte therefore you will receive 320 bytes
			// from the server.
			int[] decode = new int[320];
			int[] NRZI = new int[320];
```

```
// The baseline is an average between the high and low of the
// preamble. This will fill in the
// appropriate 1's and 0's to set up the NRZI.
for (int i = 0; i < decode.length; i++) {
	if (getFromServer[i + 64] > preamble)
		decode[i] = 1;
	else
		decode[i] = 0;

}

// This is where NRZI is decoded. We follows the rules of NRZI which
// is basically if it is a one,
// switch to the other side. If it is a zero, you remain constant.
for (int i = 0; i < 320; i++) {
	if (i == 0)
		NRZI[0] = decode[0];
	else {
		if (decode[i - 1] != decode[i])
			NRZI[i] = 1;
		else
			NRZI[i] = 0;

	}
}

// Standard 4 to 5-bit table. However the way it is implemented
// here, we have the
// binary string representation for the 5 bit and the integer
// representation for the 4 bit.
// I have it this way with the 5 bit as the string so I can match
// this up in a later portion
// of the code and spit the value up that is associated with it.
Hashtable<String, Integer> fiveToFour = new Hashtable<String, Integer>();
fiveToFour.put("11110", 0);
fiveToFour.put("01001", 1);
fiveToFour.put("10100", 2);
fiveToFour.put("10101", 3);
fiveToFour.put("01010", 4);
fiveToFour.put("01011", 5);
fiveToFour.put("01110", 6);
fiveToFour.put("01111", 7);
fiveToFour.put("10010", 8);
fiveToFour.put("10011", 9);
fiveToFour.put("10110", 10);
fiveToFour.put("10111", 11);
fiveToFour.put("11010", 12);
```

```
                        fiveToFour.put("11011", 13);
                        fiveToFour.put("11100", 14);
                        fiveToFour.put("11101", 15);

                        // The for loop goes to every patch of 5 bits. Inside the loop, i is
                        // divided by 5 to get to the first value place.
                        // With i taken to the first value place of the 5 bits, I can now
                        // cleanly place the proper bits in the correct place (1's and 0's).
                        // The fiveToFour.get will have the NRZI[i+...] get the appropriate
                        // Integer value (0-15).
                        int[] fourB = new int[64];
                        for (int i = 0; i < NRZI.length; i = i + 5) {
                                fourB[i / 5] = fiveToFour
                                                .get("" + NRZI[i] + "" + NRZI[i + 1] + "" + NRZI[i + 2] + "" +
NRZI[i + 3] + "" + NRZI[i + 4]);
                                System.out.println(NRZI[i] + " "+ NRZI[i+1]);
                        }

                        // fourB has a bunch of hex values that are in integer form, but we
                        // are going to put this all into convertedToHex which is a byte
                        // array. The bitwise operations below is merely combining fourB array items.
                        // It will allow for the 32 (64 cut in half). The first value adjusts
                        // 4 bits to the left (for integers it's a multiplication of 16) and adds the
                        // next int from fourB[].
                        byte[] convertedToHex = new byte[32];
                        int index = 0;
                        for (int i = 0; i < 64; i = i + 2) {
                                convertedToHex[index] = (byte) ((fourB[i] << 4) ^ (fourB[i + 1]));
                                index++;
                        }

                        //converts what we have in convertedToHex to Hex values.
                        System.out.println("Received 32 bytes: " +
DatatypeConverter.printHexBinary(convertedToHex));

                        // This sends to check if we got the correct data
                        os.write(convertedToHex);

                        // This lets the program know whether the process was carried
                        // successfully.
                        index = is.read();
                        if (index == 1)
                                System.out.println("Response good.");
                        else
                                System.out.println("Response bad.");

                }
                System.out.println("Disconnected from server.");
```

```
        }

}
```