

Audio Modem Interface



Watson Capstone Project No. WCP33

Charlie Miller, CoE
Kevin Felice, EE
Nelson Li, EE

Faculty Advisor: Scott Craver

April 28, 2017
Revision: -

Submitted in partial fulfillment of the requirements of ME 494/EECE 488
in the Spring Semester of 2017.

Thomas J. Watson School of Engineering and Applied Science
State University of New York at Binghamton

Executive Summary

Our project will provide a way for users of an online service to securely authenticate themselves over an acoustic channel between a mobile device and a computer in which they are using the service on. The service can be used as a portable way to log into a system such as a bank webpage that uses a login. Users shall be able to authenticate on their own mobile device so that no adversaries can use keyloggers on a computer keyboard and be able to capture their password. The payload of the data transferred shall also be encrypted so that even if other listeners were to record the data, they could not use it to do anything. Each person has their own unique private key in order to decrypt data. In order to communicate, the user must pass a challenge response protocol in which they decrypt a randomly generated encrypted string from the server. Upon successful decryption, the user can hash the message and send it back to the server where they will either be successfully authenticated or not. This system will be useful for situations where the computer being used to access the service is untrusted, such as one at a computer cluster in a public library. All communication will be done over an acoustic channel. This creates much potential for compatibility over hundreds of devices.

Table of Contents

Executive Summary	1
List of Figures	3
List of Tables	3
1. Problem Definition	4
1.1. Problem Scope	4
1.2. Technical Review	4
1.3. Design Requirements	5
2. Design Description	6
2.1. Overview	6
2.2. Detailed Description	8
2.2.1. Hardware	8
2.2.2. Software	9
2.2.2.1. Mobile Device	9
2.2.2.2. Web Page	10
2.2.2.3. Server	10
2.2.3. Communications	10
2.2.3.1. Untrusted Computer/Server	10
2.2.3.2. Untrusted Computer/Mobile Device	11
2.2.4. Encryption	12
2.3. Use	13
3. Implementation	13
3.1. Hardware	13
3.2. Software Features	16
4. Evaluation	18
4.1. Overview	18
4.2. Testing and Results	18
4.3. Assessment	24
5. Budget and Schedule	25
6. Future Plans	27
6.1. Minimizing Error Rates	27
6.2. Cryptographic Algorithm Improvement	28
6.3. Android Application Development	28
References	29
Appendix A: Test Results	30
Appendix B: Diagrams	34
Appendix C.1: Web Page Computer Code	35
Appendix C.2: Android Application Code	38

List of Figures

Figure 1. Operation Context Diagram	7
Figure 2. System Diagram	8
Figure 3. Audio Cable Circuit Diagram	9
Figure 4. Incoherent PSK Signal	11
Figure 5. Audio Signal on Oscilloscope Without Power Supply	14
Figure 6. Audio Signal Recorded Without Power Supply	14
Figure 7. Audio signal on oscilloscope without power supply	15
Figure 8. Audio signal recorded with power supply	15
Figure 9. Audio decoding process	17
Figure 10. Possible audio jack pin configurations	25
Figure 11. Cryptographic Handshake Demo	30
Figure 12. Audio Capture Demo on AMITDP	31
Figure 13. FSK Captured by Webpage	31
Figure 14. Presence of the Two FSK Tones	32
Figure 15. Audio Output on Oscilloscope	32
Figure 16. Output Read from Acoustic Channel	33
Figure 17. Detailed Operational Context Diagram	33

List of Tables

<u>Table 1. Project Budget</u>	25
<u>Table 2. Project Schedule</u>	26

1. Problem Definition

1.1. Problem Scope

The purpose of this project is to design and build a system, or device that will allow a user to remotely authenticate with a computer without using a password through an audio connection. The problem is creating an audio modem connection in which data can be safely and efficiently transferred over. This consists of a connection through the audio jack of a computer with either a smartphone or key fob device. The computer then must be able to establish a secure connection with a server and transfer data back and forth through an audio sampling web browser.

1.2. Technical Review

Computer security is an issue more relevant now than ever. With technology growing each an every day, people are out there trying to find ways to break into this technology and steal information for many different reasons. Adversaries may hack into servers in order to steal information such as credit cards, passwords, social security numbers, and much other personal information. Some want to use it for their own good such as to steal money or impersonate other people. This is a growing issue as society moves towards using computers and smartphones for everything.

Cybersecurity and how to ensure personal information is not being used by people who should not have it is an important issue. According to IC3 [1], the Internet Crime Complaint Center, 288,012 people reported a complaint of an internet crime in 2015. They also say that only about 15% of victims actually report an internet crime. With this many internet crimes happening a year, it is unimaginable what the total damage to people have been.

Another pressing issue is efficient data transfer. It is important that the data that is being transferred is being transferred not only safely, but efficiently as well. If a user is using a device that sends information to a remote server, they are going to want all of their information to be transmitted and not just some of it. More specifically, audio is an analog signal whereas data is digital. So the audio signal must be converted to a digital signal of ones and zeroes. It then must be modulated using a method such as frequency shift keying in order to be transmitted.

Making an audio modem connection requires creating the connection through the audio jack of a computer. It is common to make a device that can connect to the audio jack or use a smartphone to connect to the computer such as this project is doing. A device that uses this concept is the Square Magstripe Reader. This device is used as a mobile payment method in which the device plugs into the audio jack of a phone, and a credit card can be used to pay for whatever service or product that needs to be paid for. The reader then remotely sends the data out and the payment can be made. Applications such as these are exactly why we need this

kind of audio modem technology. They are also why it is important to make sure these devices are secure especially when dealing with valuable information such as credit card data.

1.3. Design Requirements

There are many important requirements that need to be fulfilled to ensure the project is successful. They are broken up into three categories: system capability requirements, system external requirements, and other requirements. System capability requirements is then broken up into three subcategories: overall system requirements, authentication device requirements, and remote server requirements. (See WCP33 Audio Modem Interface Project Specification document for a complete list of requirements.)

WCP-R-001: The system shall establish an audio modem connection between a device and a remote server through only an audio channel

This means the connection between the device and computer in which the browser that is sampling the audio is on shall be connected by the audio jacks of the two devices. This requirement came directly from the client due to the need to establish the connection with audio jacks as the main proponent. Using the audio jacks, an audio modem can be established. It is very important as it was requested by the client that it be through the audio jack as a Square Card Reader is, and not through a USB connection.

WCP-R-002: The system shall have a false alarm rate of at most 10^{-6}

WCP-R-003: The system shall have a miss rate of at most 10^{-2}

A false alarm in this case is when the system authenticates a user when it should not. A miss in this case is when the system does not authenticate when it should. Both cases want to be avoided as much as possible. The lower the false alarm and miss rates, the better the system is working. These requirements come directly from the client who wants to ensure that the system is working effectively. In this project, false alarms could potentially be worse than misses. This is because a false alarm could happen when an adversary is trying to get into the system. The system would accidentally authenticate the user and allow them to use the system for whatever means they intend. When trying to create a secure system, it is crucial that situations like this are prevented in order to keep the system safe to use.

WCP-R-004: The device shall be able to modulate data to be sent over an audio output

WCP-R-005: The device shall be able to demodulate data sent over an audio input

These key requirements are related to each other. These requirements apply to both the authentication device and remote server requirements. **WCP-R-011** and **WCP-R-012** are the same as the ones above, except instead of the device it is the web page that shall be able to do these. These requirements mean that in order to transfer the data efficiently, the output from a device will have to be modulated after converting the data from analog to digital. This output then becomes the input for the other device it is connected to and this must be demodulated to then be able to convert the data back into an analog signal. Both the authentication device and the web page must be able to do this process in order to effectively communicate back and forth. This requirement does not come from the client directly, but is required when dealing with data transfer between devices with audio signals. It is crucial in communication systems or else the data won't be properly transferred.

WCP-R-017: The voltage and impedances shall be properly matched between the headphone jack and microphone input to allow efficient communications

This means that when there is an output from the device's headphone jack, the impedance from the headphone jack needs to match the impedance from the microphone input on the computer that the device is sending the data into. This also is not a requirement that came directly from the client, but a necessary requirement based on equipment protection. If these impedances don't match, the device or the computer it is connected to can potentially be harmed and damaged. Matching these impedances will ensure no equipment is damaged while using the audio modem.

WCP-R-019: The data sent through the system shall be encrypted so that an eavesdropper cannot use it for their own purposes

All data that is being transferred must be encrypted due to the many internet hackers that could be trying to steal our data at any information. When it comes to transferring any data online through a web server it is always in the user's best interest to make sure their data is secure. Encrypting data with a secure algorithm will ensure that even if an adversary gets ahold of the information, they can't do anything with it.

WCP-R-023: A cryptographic handshake shall occur before data is exchanged for security purposes to prevent adversaries from impersonating a user

A cryptographic handshake can be a protocol where a public key is exchanged, and messages are encrypted back and forth in order to prove a user is who they say they are. This cryptographic handshake will establish the connection between the user and the server and provide security in that the user is who they say they are and not an adversary impersonating them. This is extremely important in the security of the overall system because this alone can stop an adversary from impersonating a user even if they have intercepted some encrypted data already. This requirement came directly from the client, but is also one that is a must when communicating data over a server as this project does. One needs to establish a secure channel before transferring any data if one wants to ensure the data is not going to be stolen and used.

2. Design Description.

2.1. Overview

The audio modem system is a system that allows the user to authenticate with a remote computer through their mobile device without the use of a password. The authentication will be verbally spoken through microphone of the mobile device. Before data is sent from the mobile device to the remote computer, the server has to establish a cryptographic handshake with the mobile device to ensure that all the data being transferred back and forth will be secure and encrypted. After the handshake is established, the mobile device will use the public key received from the server to encrypt data before sending it to the server to decrypt.

The process is designed so that even if an eavesdropper were to snatch the data while it is being transferred, it could not be used to impersonate the user. When the server verifies the user is not an imposter, their account information will be displayed on the computer for the user.

Figure 1 shows the operational context of the audio system. The mobile device establishes a connection with the computer via an audio connection that will be a 4 pin 3.5mm audio cable. Data is transferred between the mobile device and the server. While the mobile device is a trusted medium, the computer may not be a trusted device such as a computer in a public library. The user would verify their credentials through the trusted mobile device. A more detailed version of the operational context diagram can be found in Appendix A.

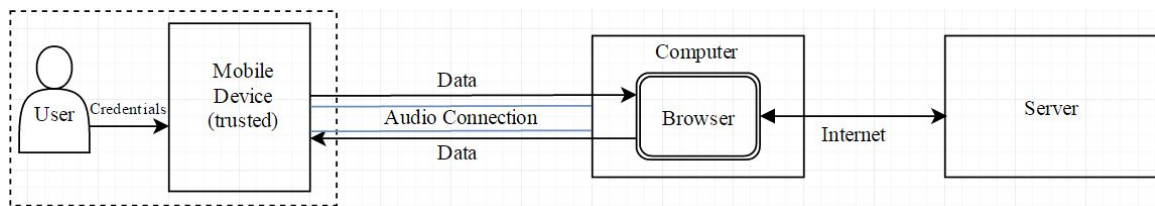


Figure 1. Operational Context Diagram

Figure 2 illustrates what goes on between the server and mobile device. When the computer and mobile device is first connected, the mobile device will send a handshake request to the server. The server will then send the public key to the mobile device and then data can now be sent between them both. Audio data is encrypted and then sent to the server to be decrypted via the private key. When the user is authenticated, the user's data will be displayed on the computer's web browser.

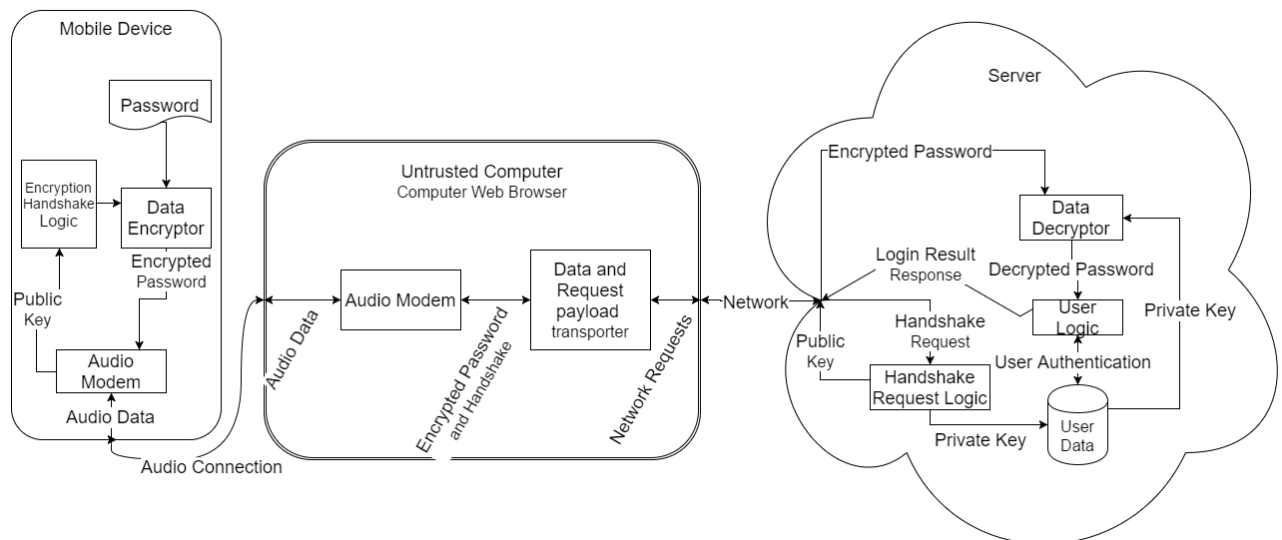


Figure 2. System Diagram

2.2. Detailed Description

2.2.1. Hardware

The only major hardware component of the audio modem system is the audio cable that connects the computer and mobile device. Because information will be passed in both directions, the cable needs to be bidirectional so a 4 pin 3.5mm audio cable was chosen as the medium. It has both a headphone output and microphone input so that data can be both sent and received in both the computer and mobile device.

However, a connection cannot be established by just connecting a male to male cable to both devices. That is because the signal needed to drive a headphone is much larger than the signal produced by the microphone. If no circuit were in place to decrease the signal, the audio jack of the microphone input may very well be damaged. Also, another problem is that the headphone output outputs stereo sound while the microphone input only accepts mono sound. Therefore the headphone output will need to be summed to become mono sound before it can pass through the microphone input.

For our design, we first decided that we would need a DC blocking capacitor in line with both left and right audio outputs. The value of the capacitor will have to be greater than $C = \frac{1}{2\pi f_c X_{in}}$ where C is the value of the capacitor, f_c is the cutoff frequency, and X_{in} is the input impedance. Next, two small resistors (~500 Ω) will be inserted in line to sum the stereo sound. Lastly, the output from the sum will need to be attenuated so that it can be safely inputted into the microphone input without damaging the audio jack. An attenuation of 1/50 should be enough but more testing will need to be done in order to produce the best result after attenuation. Here, a voltage divider should suffice and a potentiometer will go in the divider circuit. This circuit will need to be replicated for both audio jacks as data is being passed in both directions. Figure 3 illustrates this circuit design.

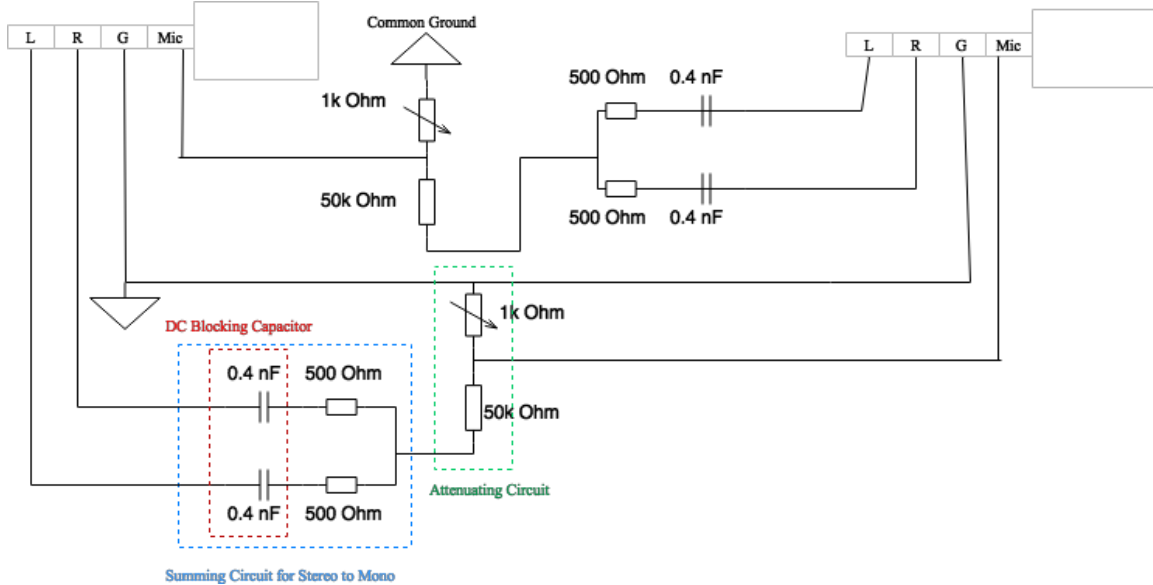


Figure 3. Audio Cable Circuit Diagram

2.2.2. Software

The system in its entirety runs on 3 different machines: The mobile device, the computer displaying the webpage, and the server. The mobile device will be running Android 5.0 Lollipop or later. The device will run an app that will contain the user's credentials in order to log in to the service. The computer in which the web page is displayed will have a JavaScript application embedded in it. The application is responsible only for demodulating the audio signal sent by the mobile device and sending it to the server, and modulating the server data and sending it to the mobile device. The server receives the data sent by the web page JavaScript. It is responsible for setting up a cryptographic handshake between itself and the mobile device. It will also determine if the proper credentials were sent by the user, and send an appropriate response.

2.2.2.1 Mobile Device

The purpose of the mobile device is to establish a connection to the server and send/receive information to/from the server. All this will be done through a custom designed app for the mobile device. The app will be designed to run on android mobile devices and will work for all android phones that have Android 5.0 Lollipop or later versions. The first time the user downloads the app, the app will ask for permission to access the microphone. The app will have an authenticate button that when pressed, will first send a handshake request to the server. After the handshake occurs, the mobile device will receive that public key and then the user will be able to verbally authenticate via the mobile device. The audio data will be modulated, encrypted using the public key, and sent to the server for authentication. If the authentication returns a successful attempt, the mobile device will display an alert indicating that the user has successfully logged in. If the authentication does not match the records in the server, the mobile device will display an alert indicating that the authentication is invalid.

2.2.2.2 Web page

The sole purpose of the webpage is to transport all the data sent between the mobile device and the server. The computer that runs the browser hosting the JavaScript is untrusted. So, all the data that goes through the web page is encrypted. The web page JavaScript application is made of two components. The audio modem, and the data transporter. The audio modem uses the Web Audio API to perform all the tasks concerning data modulation and demodulation [2]. For the application to work, the web page must be connected to the microphone of the computer. Access to the microphone must be explicitly given by the user through the browser. The application will attempt to communicate with the mobile device before doing anything else. An error message will appear if the communication test fails. Full details on how the audio modem works can be found in 2.2.3 Communications - 2.2.3.2 Untrusted Computer/Mobile Device. Once the payload has been received and fully demodulated, the data transporter will send the data with an HTTP POST request to the server. Similarly, when solicited, the server may send data for the mobile device. The data transporter will receive data intended for the mobile device. That data will be modulated with the audio modem. The audio modem will use the Web Audio API AudioBuffer to play the modulated message. If the user successfully gives the correct credentials, they will be logged in.

2.2.2.3 Server

The server will run a PHP script to handle all data sent from the mobile device. The server is responsible for enforcing authentication of the user. When it receives a POST request, it checks if the request is for a handshake, or an authentication. If it is an authentication, the payload is given to the decryptor. The decryptor will find the appropriate key based on the user from its database. Based on the encryption scheme we use (see 2.2.4), the server shall send a “challenge” to the mobile device. The device will then need to respond to that challenge with the correct answer. If the correct answer is given, the server will log the user in, sending them to the dashboard on the untrusted computer. If the incorrect answer is given, the server will not do anything except tell the web page to show the user it was unable to authenticate with the provided credentials. If the server is given a handshake request, it will create a unique key based on the chosen encryption scheme. That key will be used in the future for decrypting future authentication requests.

2.2.3. Communications

2.2.3.1 Untrusted Computer/Server

The communications that take place between the untrusted computer browser and the server shall take place over the internet, using HTTPS protocol. The website must be served over a secure connection because critical features of the Web Audio API will not work on an insecure website [3]. Both handshake requests and authentication requests will be

administered through POST requests generated from the JavaScript running on the browser. These requests will be Asynchronous Javascript And XML calls to create a seamless experience to the user. If the server returns with a failed authentication, it will tell the user on the login page, the user will be notified on the web page. However, if the authentication is successful, the login landing page will be loaded instead.

2.2.3.2 Untrusted Computer/Mobile Device

Innovation takes place in the communication between the untrusted computer and the mobile device. The connection between both devices is a physical wire described in section 2.2.1 Hardware. The audio jack output of one device is connected to the mic input of the other. The encoding technique will be one of the following: Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK), Phase Shift Keying (PSK), and Quadrature Amplitude Modulation (QAM). The signal sent by both devices must be coherent, or have no jumps in slope. The figure below depicts an incoherent signal.

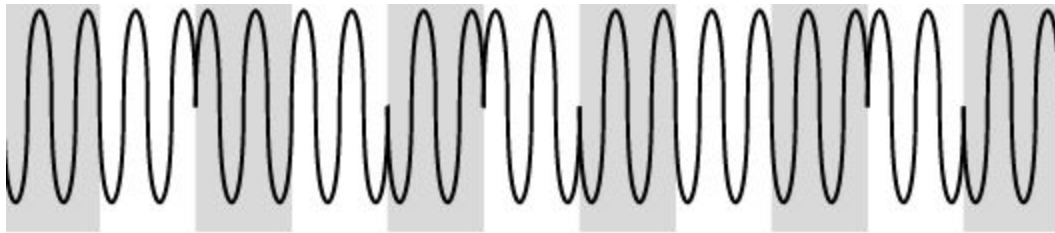


Figure 4. Incoherent PSK signal

ASK will change the amplitude of the signal to represent a 1 or 0 in the data being sent. FSK sends 2ⁿ different frequencies to represent the data. For example, the device might send 4 different frequencies to represent 00, 01, 10, or 11. PSK uses different phases to represent the data. Finally, QAM uses a combination of PSK and ASK to represent the data. The encoding scheme we use will be determined in the spring. Factors in our consideration is the required bandwidth, bit rate, and ease of implementation. ASK is relatively easy to implement, however, the amplitude of the signal not be strong enough to provide a reliable signal. In other words, the Signal to Noise Ratio (SNR) may be too great for ASK to be our choice. FSK is also relatively easy to implement, however, it requires much bandwidth, because it must use at least two different frequencies to represent its data. This problem can be mitigated by reducing the baud rate, the rate in which each signal element is sent. However, that reduces the bit rate. Though the bit rate is not a priority in our project because the size of the payload is not large (~50 bytes). FSK will likely be our choice. PSK does not use much bandwidth, however, it is hard to implement, especially if the signal must be coherent. QAM can achieve a high bit rate along with using minimal bandwidth, but comes with the challenges of both ASK and PSK.

The baud rate will be consistent in each direction. The baud rate will be determined in the Spring. The baud rate will be directly affected by the SNR of the physical connection. It will be optimized so that it will achieve a success rate of at least 99% while achieving the fastest bit rate possible. See WCP-R-003.

Both devices will attempt to communicate with each other before doing any real work. This is to make sure the audio connection is good. It is a likely scenario that the user will have their audio muted, making communication impossible. Once the reliable connection is created, the web page will continuously listen to the microphone input. Only the web page does this because the server will not send information to the mobile device unsolicited. Only after the mobile device makes a request does it start listening for audio input. Data reception is generally the same however. The modem takes the FFT over 4096 samples of the input for every 256 samples taken in. If the frequency 20kHz is present, it will start recording. It will continue recording until that frequency is gone. No data will be sent until the 20kHz tone has played for 1 second, and it will stop 1 second after all data has been sent. A 19kHz frequency will play to indicate where the data payload is. The modem cuts off the recording so that it has only the payload. The PCM data will then be analyzed based on the baud rate, device sample rate, and data encoding scheme we choose. In the likely case of FSK, the FFT will be taken at each data element of the signal. The frequency will determine which bits are meant to be conveyed.

Our choice of mobile device will be both the Samsung Galaxy S5 and the Nexus 6P. Therefore, we must use the Android environment to perform the audio modem duties. Android provides the `AudioTrack` class which can play a PCM audio buffer defined by the programmer [4]. The buffers follow which encoding scheme we choose in the spring. On the JavaScript side, the Web Audio API provides the `AudioBuffer` class which also stores a PCM audio buffer defined by the programmer which can also be played [5].

2.2.4 Encryption

All communications need to be encrypted in order to ensure that if an adversary gets a hold of the information, that they can't do anything with it without the key that would decrypt it. The encryption algorithm and security protocol needs to be selected (see section 4.2). First and foremost the security protocol would establish a secure connection with the server. This can be done by a cryptographic handshake involving the swapping of keys. The computer can then send an encrypted message to the user, and then the user can decrypt it and send it back. Knowing the user is the only one with the private key, the server would know this is really the right user and not an adversary if it was decrypted and sent back. Other methods to consider with this is using either a timestamp or a simple phrase in the message such as "This message is intended for the user." So if an adversary stored a message and tried to send it back later in a replay attack, the computer would either see one of two things; 1) the timestamp is out of date and it is an old message or 2) the message says "This message is intended for the user," but is being sent back to the server. The server would then know it is someone trying to break in and would not establish the connection.

Secondly, after the secure connection is established, all data needs to be encrypted that is sent over the channel. When dealing with valuable information like passwords, it is good practice to send and store hashes of the password rather than the password itself. Therefore if someone were to steal the password somehow, they would be looking at a random hash rather than the actual password and wouldn't know what to do with the hash. Some common encryption methods to consider are RSA, Data Encryption Standard (DES), Advanced Encryption

Standard (AES), and Blowfish. All these common encryption methods have been proven to be efficient and hard to break.

2.3. Use

In order to use this project, the user will have to download the application on their mobile device. The mobile device will then have their credentials on it, or the user will input their credentials such as a username and password. They will then press an authenticate button which will then send the data as an audio signal, and convert it into digital data. The data will be modulated and encrypted and sent through the connection to the computer where google chrome will audio sample it. It will then be sent to the server after a secure connection has been established through a cryptographic handshake. The server should then demodulate and decrypt the information and be able to authenticate the user.

The audio modem interface can be used for many different companies and people. An example is a bank webpage where users can authenticate their passwords in order to log onto the bank's website in order to do whatever they may need on that site. It could also be used in applications like the Square Card Reader where you can send a person's credit card information over the connection to make a transaction.

3. Implementation

3.1 Hardware

The prototype was designed as an interactive webpage where a user can interact with a server in order to log in. The prototype serves the purpose of securely authenticating a user through communication through an acoustic audio channel. Originally the communication was designed to occur through an audio cable with a circuit connected in order to match voltages and impedances of the audio jacks of the computer and Android phone. However, during implementation problems arose that did not allow for successful further implementation of the cable and circuit. One problem that occurred was when a power supply was plugged into a computing using the project, interference was created throughout the circuit resulting in the circuit allowing noise to dominate the signal. This can be seen in the figures below. Figures 5 and 6 show our signal without the power supply connected going to the oscilloscope, and the audio signal recorded by the computer respectively. Figures 7 and 8 show the signal being corrupted with the power supply going to the oscilloscope and the audio signal recorded by the computer respectively.

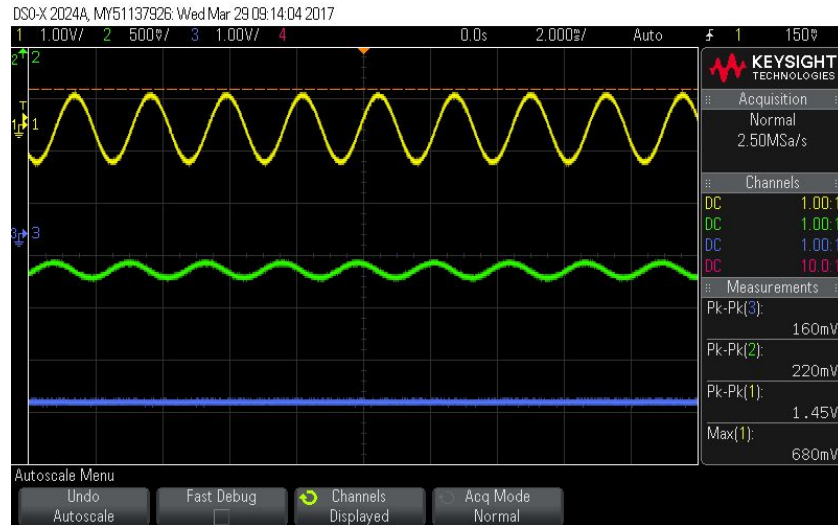


Figure 5: Audio signal on oscilloscope without power supply

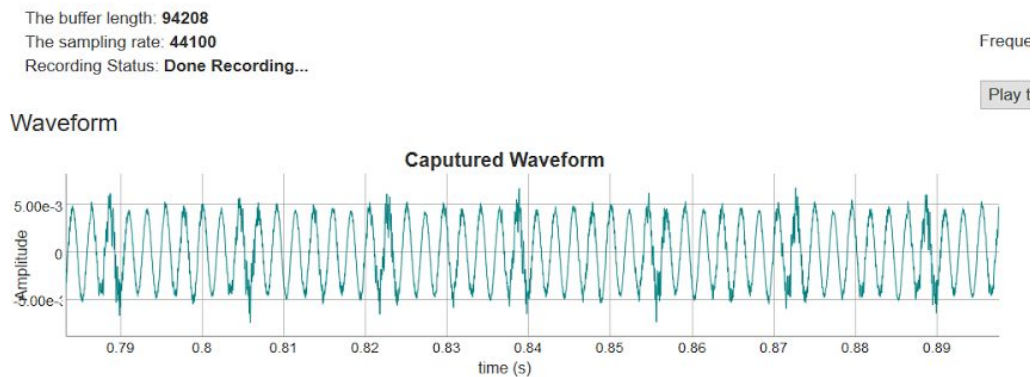


Figure 6: Audio signal recorded without power supply

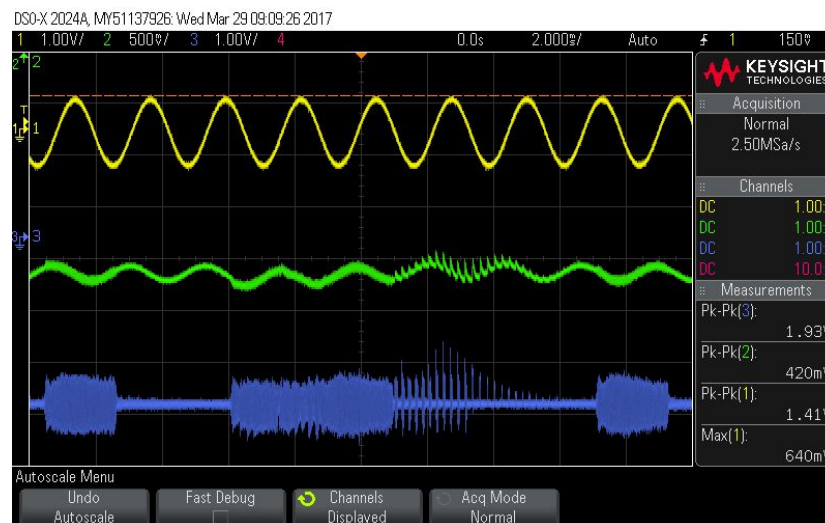


Figure 7: Audio signal on oscilloscope without power supply

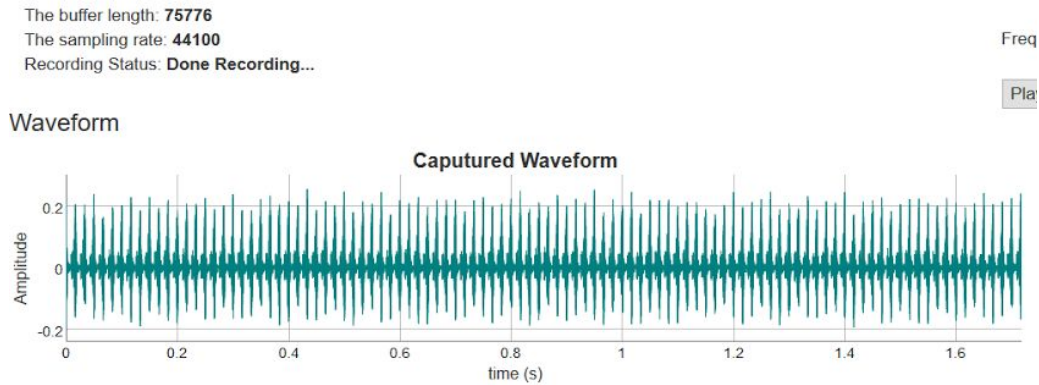


Figure 8: Audio signal recorded with power supply

One solution we came up with for the interference problem was to use a ferrite clamp on core. The ferrite core is a magnetic core used as a clamp on inductor that would help prevent eddy currents. Unfortunately, this solution did not prove successful.

Another problem that occurred with our circuit is that it failed to work properly between two computers that weren't similar and both didn't have a bidirectional headphone/microphone jack. Computers without the bidirectional jack did not recognize the circuit when it was connected, resulting in the computer using the wrong microphone to record. After experiencing these problems, our faculty advisor then suggested to us that we switch to an acoustic channel by playing and recording audio signals out loud to the web page. While at first this may seem like an unsecure method to communicate data since adversaries could record anything played out loud, the cryptographic methods used would prevent adversaries from knowing the messages of anything they record. All messages will be encrypted or hashed and therefore the adversary would not be able to tell what they are unless if they had the private key to decrypt them in which case would be the equivalent of knowing a user's password.

3.2 Software Features

The web page is being hosted by cloud9.

The web page contains many key features in order to accomplish this. To start in order to begin communicating a cryptographic challenge response handshake must occur. The user would request to the server to begin communication by typing in their username. The server will first lookup the username to see whether they already exist or not. If the user doesn't exist, they will be added to a "users" text file and a public key and private key will be generated for them. If they do exist, their public key and private key will have already been generated therefore the server will then find the user's public key. It will then generate a random string that it will hash and store in a text file, and also encrypt the random string with the user's public key.. It will POST this encrypted random string as a challenge to the user. The user must be able to decrypt this random string and send back a hash of it to the server to be compared to the original hash. Since the user is the only one with the private key, they are also the only one able to accomplish the decryption. Since the server is generating a random

nonce and therefore be different every time a user requests to communicate, an adversary won't be able to simply record the decrypted hash and use it in the future. The hash will be different each time a user wants to communicate and only the person with the private key will be able to generate the correct one. If the user is able to decrypt the message, and hash it to the same value that the server stored, then they will be authenticated. In order to accomplish the decryption, the user can copy and paste the encrypted challenge posted on the website to the next textbox that says "Paste Authentication challenge here." Upon clicking submit, a php script will run and decrypt the message using the user's private key, hash it, and post it back to the website. The user can then copy and paste this into the next textbox reading "Paste supposed hash of original message here." Upon clicking submit, this will then run the last php script that will check whether the user submitted hash is equal to the original hash or not. If it is equal the user will be authenticated, and if it is not the user will not be authenticated.

In order for each communication to happen between user and server, another key feature must be used. Since the communication is happening over an acoustic channel, there are two features that can make this happen. There is a record button that will record an audio input, decode the audio signal into bits, and back into ASCII characters. There is also a button that will do the opposite in order to transmit a message into an audio signal. The transmit string button will take a string, encode into bits and transmit the bits as an acoustic signal. These buttons must be used during the challenge response protocol in order to communicate the information from the user to the server.

3.3 Audio Modem

The encoding technique we chose was Frequency shift keying or FSK. We chose this for encoding for 3 reasons

1. Implementation was the easiest and saved us time. Additionally, there were already implementations of encoding and decoding with this scheme which helped get us started. Our code however, is completely original.
2. It was easiest to debug. We could easily see the differences in frequencies as opposed to different phases or amplitudes. We could also hear the differences
3. It is resistant to some noise

Since our data came in the form of ASCII characters, we were able to convert those characters into a bitstream which was derived from each character's ASCII code assignment. A 1700Hz sine wave was played for all 1's, and 1300Hz sine wave was played for all 0's. Each bit was played at a baud rate of 100 bits per second. The greatest common factor of 1700 and 1300 is 100. Therefore each signal will be starting with an angle of 0 at the start of every signal element, or chip. This allowed us to locate where both the 1700 sine and 1300 sine were at angle 0, which must be the start of a chip. We used WebRTC's web API to play our data. Specifically, `AudioBufferSourceNode` was used

We used the WebRTC's web API's to capture the acoustic signals into our Javascript program. Samples came in 2048 samples at a time, but were processed on the spot. We were able to synchronise the window for bit detection was by playing both 1700Hz and 1300Hz tones

simultaneously. A window half the size of a chip is used to determine which bit is being played. The signal of that window was multiplied by both a 1700 sine and 1300 sine separately to find out the power of either frequencies present in the window. This was compared with the total power of the the window. A percentage for each frequency was calculated as so:

$$\text{percent_1700} = \text{power_1700} / \text{total_power}$$

$$\text{percent_1300} = \text{power_1300} / \text{total_power}$$

In this implementation, we got excellent results where percentages for either 1300Hz or 1700Hz were at 99% half of the time. However, the other half had poor results where percentages were 25% and 35%. Upon looking at the windows for bit detection, it was found that the synchronising signal did not work as well as expected.

Fortunately, inspection of the power percentages for each bit in multiple run throughs of bit detection showed that the synchronizing signal percentages were very similar to each other. From this we are able to distinguish a synchronizing signal from a data signal that has been poorly synced. Thus we added a string of 1's and 0's to the end of the initial syncing signal. With this we were able to detect if the first synchronizing signal failed. We would shift the bit detection window if that happens to be the case. The string would then be followed by two syncing signals to tell the interpreter that the real payload is to follow.

Finally we appended a syncing signal after the data payload to tell the interpreter that is the end of the signal.

Once the bits were recorded, the interpreter would then pack the bits back together 8 bits at a time. They were parsed into ASCII characters as before. This data would then be sent to the correct place. The figure below show the beginning of a transmission, from the syncing 0's and 1's, two additional syncing signals, followed by the databits. Highlighted are the windows for bit interpretation. The red highlighted windows are for the data bit windows. 2's indicate sync signals. 4's indicate less confident sync signals.

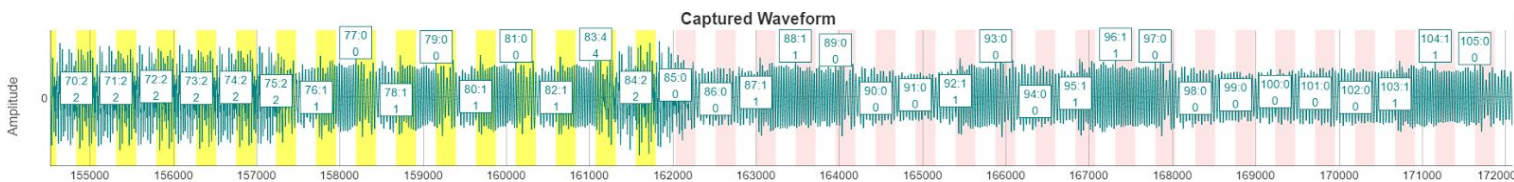


Figure 9: Audio decoding process

4. Evaluation

4.1. Overview

In order to evaluate the design of our Audio Modem, all tests can be done in the senior design lab by any student. Many of the tests can be accomplished using the “Audio Modem Interface Technology Demo Page” (AMITDP) described in the WCP33 - Test Plan and Procedures document. We used a combination of experimental testing and user testing in order to test each project requirement. All the major project requirements can be found in the summary table in Section 1: Problem Definition as well as a full set of requirements in the WCP33 - Project Specifications document. The AMITDP has many useful features on it that will allow testing of many requirements such as its ability to record audio signals, to transmit bits as an audio signal, and to perform a cryptographic handshake. All referenced tests and details of each individual test can be found in the WCP33 - Test Plan and Procedures document. The results of each test can be found in Appendix A: Test Results.

4.2. Testing and Results

4.2.1. WCP-R-001: The system shall establish an audio modem connection between a device and a remote server through only an audio channel

This means the connection between the device and computer in which the browser that is sampling the audio is only an audio connection. In our case we made an acoustic channel that the device and server will be communicating through. In order to communicate, one device must transmit an audio signal while the other records. The results of this test can be seen in figure 12 in Appendix A: Test Results. We tested this with user testing with a computer and mobile phone next to each other. The mobile device would transmit a message, while the other computer would receive and decode the message. We then transmitted a message back to the other computer while it recorded and then decoded the message. Both were successful in transmitting, recording, and decoding messages.

4.2.2. WCP-R-002: The system shall have a false alarm rate of at most 10^{-6}

From this, an attacker must only be successful 1 in 1 million times. Because our test T001 shows that our encryption algorithm works, that must mean all data payloads are successfully encrypted. The only way for an attacker to be successful is to crack the encryption which would take years or decades if they are lucky.

4.2.2.1. WCP-R-003: The system shall have a miss rate of at most 10^{-2}

Use of an acoustic channel proves to be a difficult task. It was also difficult to correctly find the start and end of each transmission. If this was incorrect, it could throw the entire interpretation off. In the end. We were able to get the bit error rate to 15%. However, even this could greatly impact the system as a whole, as the ASCII characters would be interpreted incorrectly. In a future implementation, we would incorporate hamming codes into our payloads so that the system as a whole will be more tolerant to bit errors.

4.2.3. WCP-R-004: The device shall be able to modulate data to be sent over an audio output

Encrypted data shall be modulated then sent over an acoustic channel for another device to be demodulated as data. We tested this using T006 - Modem and an Android phone. Our code used frequency shift keying (FSK) for the modulation scheme. We outputted data and used an oscilloscope to measure the signal being sent. Also, we used the web page to verify that the correct frequencies were being sent. The requirement is met after testing the signal multiple times and verifying that the sent frequencies were correct.

4.2.4. WCP-R-005: The device shall be able to demodulate data sent over an audio input

Encrypted data shall be demodulated when received over an acoustic channel. We tested this using T006 - Modem and an Android phone. Our code used frequency shift keying (FSK) for the modulation scheme. We outputted data and used an oscilloscope to measure the signal being sent. Also, we used the web page to verify that the correct frequencies were being sent. The requirement is met after testing the signal multiple times and verifying that the sent frequencies were correct.

4.2.5. WCP-R-006: The device shall not send nor receive data over an internet connection

Since our project is communicating through an acoustic audio channel, no data should be sent over the internet. We tested this using T007 - Internet Connectivity. We shut off the internet and played an audio signal from one computer to another. The computer receiving the signal recorded it and decoded the message without having to be connected to the internet. All communication between devices will be done this way and therefore the requirement is satisfied.

4.2.6. WCP-R-007: The device shall send and receive data only over an audio output and input respectively

The audio outputted will be received by the microphone and speakers of an Android phone. We tested this using T008 - Data Exchange. Although we did not end up using the circuit to send and receive data, the acoustic channel worked fine. Outputted data will be displayed on the webpage when it is captured by the microphone. We also capture the data on an oscilloscope to ensure that the data matched. The phones were both successful in sending and capturing data with minimal data loss.

4.2.7. WCP-R-008: The device shall perform a test to make sure the audio connection is valid.

Using an acoustic channel, there is no auxiliary cable that needs to be connected anymore. This is beneficial in the aspect that the device no longer has to make sure any audio connection is valid. Since there is no connection at all, as long as the computer and device are in close proximity the system will work as desired.

4.2.8. WCP-R-009: The server shall give a web page that is available to anyone with all of the requirements listed below

This was verified with tell T012. The page has loaded on 5 different computers with internet access. All functions of the system were available on all 5 computers.

4.2.9. WCP-R-010: The web page shall contain scripts that can run on Google Chrome v. 53.0.2785.143 m or better.

4.2.10. WCP-R-011: The web page scripts shall be able to modulate data to be sent over an audio output.

Encrypted data shall be modulated then sent over an acoustic channel for another device to be demodulated as data. We tested this using T006 - Modem and a personal laptop. Our code used frequency shift keying (FSK) for the modulation scheme. We outputted data and used an oscilloscope to measure the signal being sent. Also, we used the web page to verify that the correct frequencies were being sent. The requirement is met after testing the signal multiple times and verifying that the sent frequencies were correct.

4.2.11. WCP-R-012: The web page scripts shall be able to demodulate data sent over an audio input.

Encrypted data shall be demodulated when received over an acoustic channel. We tested this using T006 - Modem and a laptop. Our code used frequency shift keying (FSK) for the modulation scheme. We outputted data and used an oscilloscope to measure the signal being sent. Also, we used the web page to verify that the correct frequencies were being sent. The requirement is met after testing the signal multiple times and verifying that the sent frequencies were correct.

4.2.12. WCP-R-013: The web page script shall send and receive data over an audio output and input respectively

The audio will be sent and received by two computers set up side by side. We tested this using T008 - Data Exchange. Although we did not end up using the circuit to send and receive data, the acoustic channel worked fine. Outputted data will be displayed on the webpage when it is captured by the microphone. We also capture the data on an oscilloscope to ensure that the data matched. The computers were both successful in sending and capturing data with minimal data loss.

4.2.13. WCP-R-014: The web page script shall perform a test to make sure the audio connection is valid

This is another requirement that is satisfied when switching to an acoustic channel. Using an acoustic channel in which all data is transmitted via an audio signal played out loud, no audio cable connection is needed. When abandoning the circuit as advised by our faculty advisor, we no longer need to perform a test to ensure that the audio connection is valid.

4.2.14. WCP-R-015: The device shall be modular and portable

The authenticating device should be able to be carried with the user wherever they go. This was another test satisfied by T009 - Physical Attributes with inspection. Our web page works great on Android device web browsers. Since mobile devices are portable and can be taken wherever you go, the requirement is satisfied.

4.2.15. WCP-R-016: The system shall interface between an Android device and personal computer

While the Android application itself hasn't been developed, the Audio Modem can still actually run and interface between an Android device and a personal computer. This was verified in all tests that included using the AMITDP with both an Android phone and a personal computer such as T001 - General Encryption. The web page can run on both personal computers and mobile devices. When we captured audio on the mobile phone it was just as effective as capturing audio on the personal computer. The Android was also able to transmit data and authenticate users just as well as the personal computer. Therefore this requirement has been satisfied. In the future it could be better satisfied with the development of an Android application.

4.2.16. WCP-R-017: The voltage and impedances shall be properly matched between the headphone jack and microphone input to allow efficient communications

When using an audio cable to connect two bidirectional headphone jacks, it is important to make sure that the voltage levels and impedance levels between the two devices are properly matched or damage could occur to the devices. This requirement was taken care of when switching to an acoustic channel as advised by our faculty advisor. An audio cable was no longer needed and therefore this requirement was no longer necessary. Using an acoustic channel, the audio signal would be played out loud and therefore not use the headphone jacks.

4.2.17. WCP-R-018: The auxiliary cable shall be a 4-pin 3.5mm male-to-male so that information can be sent and received

This requirement was originally going to be tested using T009 - Physical Attributes, however, as our faculty advisor suggested us to switch to an acoustic channel this requirement no longer became necessary. When testing it on the old circuit, the requirement would have been satisfied. Since we switched to an acoustic channel however, this requirement is no longer needed since all information will be sent and received over the acoustic channel instead of the auxiliary cable.

4.2.18. WCP-R-019: The data sent through the system shall be encrypted so that an eavesdropper cannot use it for their own purposes.

As described in section 2: Design Requirements, this requirement is to prevent eavesdroppers from stealing data that is being transmitted. Using a secure encryption algorithm is important to accomplish this. The RSA encryption algorithm was implemented in order to accomplish this task. When the server sends the challenge for the cryptographic handshake, the message is encrypted using RSA where only the user who has the private key can decrypt the message. This was tested by user testing on the AMITDP. We acted as a user requesting to communicate with the server by inputting a username. The server responded with the public key and the encrypted message. By using the AMITDP to request communication with the server the system was successful in recognizing us as the user and encrypting the message as part of the challenge in the challenge response protocol. The encryption algorithm works great and keeps our information secure.

Even if the adversary were to record the audio data, they could not do anything with it without the private key. Also the server generates a new random string every time a user wants to communicate, therefore an adversary can't record a hash of the message, store it and try and use it to log in at another time. Since the string is different every time, so will the hash of the message and therefore using old information won't help an adversary authenticate as somebody else in the future.

4.2.19. WCP-R-020: The device shall be capable of encrypting its payload.

The mobile device being used must be able to encrypt its payload in order to prevent adversaries from stealing information. This requirement was tested in T001 - General Encryption. The device and server used the AMITDP in order to perform the cryptographic handshake. When requesting to communicate, the server sent an encrypted random string back to us. The server used RSA in order to do so. The AMITDP can be used on both a mobile device and a computer therefore both are capable of encrypting a payload. This requirement was satisfied. RSA is an efficient encryption algorithm that is hard to break. Therefore our results are promising that adversaries won't be able to break it in the future.

4.2.20. WCP-R-021: The server shall be able to decrypt encrypted data sent by the web page separate from the machine which the browser is running on

The server's ability to decrypt encrypted messages is similar to requirement WCP-R-020 in which the device can encrypt a payload. Both requirements are tested in T001 - General Encryption. After the AMITDP encrypted a random string as described in section 4.2.20, we input the encrypted message into the "Paste Authentication challenge here" textbox on the mobile device. The mobile device then decrypted the message using our user's private key. The message was successfully decrypted separate from the computer in which the browser was running on. This requirement was successfully verified through user testing.

4.2.21. WCP-R-022: The authenticating technology shall allow the user to authenticate with multiple machines, with a degree of security akin to having separate passwords for those multiple machines.

Any user that is to use the Audio Modem should be able to log into multiple machines using a different "password" each time. In order to pass the cryptographic challenge response protocol, a user must decrypt a random string with their private key. Each user has a unique public key/private key pair. Every time the user tries to authenticate, they get a unique message as well since it is a randomly generated string. Therefore logging into one machine will not be the same as logging into another machine just as logging into a machine once will not be the same as logging into the same machine at another time. The hash of the random string will always be different and therefore will have the degree of security like having separate passwords for multiple machines. When testing this with T005 - Multiple Device Authentication, we used two different mobile devices to authenticate with the server. Both were successfully authenticated with different messages. Therefore this requirement was

verified.

4.2.22. WCP-R-023: A cryptographic handshake shall occur before data is exchanged for security purposes to prevent adversaries from impersonating a user.

In order to make sure that an adversary is not trying to impersonate a user, a cryptographic handshake can be used to prevent this. The handshake follows a challenge response protocol in which the user requests to communicate, the server sends a challenge such as an encrypted nonce, and the user decrypts the random nonce and sends it back. To make it even more secure the protocol also hashes the secret message so an adversary cannot simply trick a user into decrypting messages for them. This requirement was tested with user testing on the AMITDP. The entire challenge response protocol can be done on the web page. To test this we acted as a user. We inputted our username, and received an encrypted challenge back from the server. We then decrypted the message in the next text box and sent back the hash of the message for the server to compare to the original hash. The system then authenticated us correctly since we sent the correct hash of the original message like the server expected. When we purposely input the wrong hash or encrypted message to be decrypted, we were not successfully authenticated as we shouldn't be. The challenge response protocol worked just how we planned it would. It is impossible for an adversary to impersonate a user unless if they somehow manage to get the user's private key.

4.2.23. WCP-R-024: An adversary who compromises the remote server shall not be able to impersonate the user to other machines.

If an adversary were to compromise the server and get a hold of all the files and code, they should not be able to find user's private keys. This would keep user's safe from impersonation. When performing T004 - Server Security, this requirement failed. When inspecting the code, we realize that the private key is generated on the server and stored on the server in our implementation. However, this is not our original design and therefore would not be a problem in the future if more work is done on the Audio Modem. In the future, an Android application should be developed that would translate the code on the server and be able to do everything the server can. The private keys would be generated and stored on the mobile application of each users' smartphone rather than the server. This would solve the problem of an adversary stealing a user's private key if they compromise the remote server. While the requirement failed now, with more development it could successfully pass this test in the future.

4.3. Assessment

This design is still viable practically, but not economically so. The bit error rate is still too high in order to motivate anyone to use this alternative method of authentication. Even in a quiet room it is still possible to incorrectly interpret the audio signal coming through. Because of this decoding was one of the most challenging aspects of this project.

Much of our time was spent attempting to create a wired interface between any two computers using their audio jacks. Most non-audio-related devices which use the audio jack

of a device is specifically designed with a device in mind. The reason this is the case is that creating such an interface is a very challenging problem. It is very challenging for two reasons:

- 1) The audio jack pin locations are not standardized. The figure below shows 4 different pin configurations in the world:

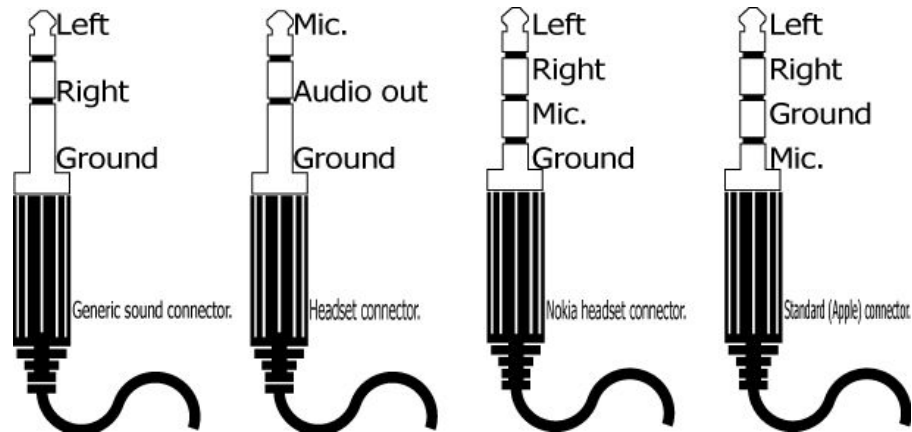


Figure 10: Possible audio jack pin configurations

- 2) There was radio frequency interference coming from the power sources of the computers. The counteraction of this noise was beyond the scope of this project

A development for this product would take more time and money than what money the solution may yield.

There still lies potential in this design, however. The design is platform agnostic; any device that runs javascript and has web access will be able to use it. We did not put in our requirements that the modem shall run on iOS, yet it does with a web browser running on it. Hamming codes ought to be used for the transmission of the data payloads to reduce the error rate of the entire system. With audio modem alone devices may be able to communicate with other over short distances if there are no other signals for communication available.

5. Budget and Schedule

The following table is the project budget broken down into estimated costs, to date money spent, and remaining money left over.

Table 1. Project Budget

Items	Original Estimate \$	Actual to Date \$	Estimate to Completion \$	Estimate at Completion
-------	----------------------	-------------------	---------------------------	------------------------

Web Hosting	50	0	50	50
SSL Certificate	60	0	60	60
3.5 4 pin AUX cable	10	3	7	10
1000 pF + 470 pF ceramic capacitors	5	1	4	5
10k logarithmic + linear potentiometers	10	6	4	10
Resistors	5	0	5	5
Ferrite Cores	4	4	4	4
Total \$	144	14	130	144
Reserve \$	156			156
Funding Limit \$	300			300

The following table is a detailed schedule for the design phase of the project.

Table 2. Project Schedule

		Name	Duration	% Complete	Start	Finish	Predecessors
1		Learning How to Implement Features	14d	100%	10/18/2016	11/04/2016	
2		Understand how to code and design an Android app	7d	100%	10/18/2016	10/26/2016	
3		Setup a basic website in bingsuns	7d	100%	10/18/2016	10/26/2016	
4		Output waveform from Android phone	7d	100%	10/27/2016	11/04/2016	2,3
5		Produce or capture sound on website	7d	100%	10/27/2016	11/04/2016	2,3
6		Create Audio Interface	84d	95%	11/04/2016	03/01/2017	
7		Research	7d	100%	11/04/2016	11/14/2016	
8		Order cable	7d	100%	11/14/2016	11/22/2016	
9		Testing Headphone Jack and Mic Input	1d	100%	11/22/2016	11/22/2016	
10		Making the circuit	2d	100%	11/23/2016	11/24/2016	9
11		Testing the circuit and connection	69d	100%	11/25/2016	03/01/2017	10
12		Build PCB (optional)	1d	0%	11/30/2016	11/30/2016	
13		Design Website	70d	100%	01/20/2017	04/27/2017	1
14		Setup a server	2d	100%	01/20/2017	01/23/2017	1
15		Design a UI	2d	100%	01/24/2017	01/25/2017	14
16		Design a modulation method	20d	100%	03/01/2017	03/28/2017	14
17		Implement chip find function	19d	100%	03/27/2017	04/20/2017	
18		Set up cryptographic handshake	12d	100%	04/10/2017	04/25/2017	16
19		Testing transfer of data between devices	69d	100%	01/23/2017	04/27/2017	
20		Project Evaluation	1d	100%	04/28/2017	04/28/2017	6,13
21		Testing, Analysis, and Inspection of Project Requirements	1d	100%	04/28/2017	04/28/2017	
22		Testing Procedures	4d	100%	04/24/2017	04/27/2017	
23		Communication Testing	4d	100%	04/24/2017	04/27/2017	
24		Security Testing	4d	100%	04/24/2017	04/27/2017	
25		Bandwidth Testing	4d	100%	04/24/2017	04/27/2017	

6. Future Plans

Many steps can be taken in the future in order to make this project more efficient, and secure. This project can be handed down to future students with the AMITDP where they can work to improve the system. There are a few areas where the project needs additional work that can accomplish this. Once these are accomplished, the audio modem may be ready to use in a real world application such as logging into a company webpage. The areas that could use additional work can be broken down into three categories:

6.1. Minimizing Error Rates

In order to minimize error rates, we could incorporate hamming codes into our payloads so that the system as a whole will be more tolerant to bit errors. Hamming codes are binary linear codes used as error correction. Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors and therefore help lower the overall error rates the Audio Modem experiences. Minimizing the error rates is crucial for our project since we want users to be authenticated successfully when they should, and unsuccessfully when they shouldn't.

6.2. Cryptographic Algorithm Improvement

RSA is an efficient and secure algorithm that we chose to use. However there are other algorithms that are also secure, that could be more efficient while using less data. An example of this is Elliptic Curve Cryptography. Elliptic Curve Cryptography uses much smaller key spaces than RSA does. A RSA key size of 1024 bits is equivalent to an Elliptic Curve Cryptography key size of 160 bits which is significantly less. This would allow for less data to be transmitted in communication and therefore our modem working faster with the same amount of security.

6.3. Android Application Developing

The webpage code should be transcribed to Java and used to develop an Android Application. Since the server side code works the same way as the application should, the code can be transcribed to work with an Android application. This can improve security as it will allow the private keys to be stored directly on the phone application where only the user can access.

References

- [1] "Annual Reports." Internet Crime Complaint Center (IC3). Federal Bureau of Investigation, 2015. Web. 21 Nov. 2016. <<https://www.ic3.gov/media/annualreports.aspx>>.
- [2] "Web Audio API." Web Audio API. W3C, 2015. Web. 21 Nov. 2016. <<https://www.w3.org/TR/webaudio/>>.
- [3] <<https://sites.google.com/a/chromium.org/dev/Home/chromium-security/deprecating-powerful-features-on-insecure-origins>>
- [4] <<https://developer.android.com/reference/android/media/AudioTrack.html>>
- [5] <<https://developer.mozilla.org/en-US/docs/Web/API/AudioBuffer>>
- [6] "Audio Modem: Data over Sound." Audio Modem: Data over Sound — Applidium. Applidium, 09 Oct. 2013. Web. 21 Nov. 2016. <https://applidium.com/en/news/data_transfer_through_sound/>.

Appendix A: Test Results

Figure 11 below shows the results of the cryptographic challenge response protocol. Tests T001 - General Encryption, T003 - Cryptographic Handshake, and T005 - Multiple Device Authentication all make use of the AMITDP in this manner. As can be seen for these three tests, all the results are in the screenshot. To begin the user inputs their username in the textbox as can be seen with “Bob” as the input. After clicking submit, the message in the first textbox on the right appears showing the user’s public key and their encrypted challenge. The encrypted challenge is the random string that must be decrypted. This shows that the system is capable of encrypting a payload. The user then copy and pastes the challenge into the “Enter name and paste authentication challenge here” textbox on the left as can be seen. Upon clicking submit, the Decryption response textbox will display the hash of the decrypted message. The private key is used to decrypt the message. The message is then hashed and displayed. This shows that the server is capable of decrypting an encrypted message. It also show that adversaries cannot use the data for their own purposes. Since each message is randomly generated, it will always be different therefore even if an adversary got a hold of the hashed message, they cannot use it later in a replay attack in order to log in. After the decryption is posted in the second textbox on the right, the user can copy and paste the decryption into the “paste supposed hash of original message here” textbox. Upon doing this, the system will either successfully or unsuccessfully authenticate a user. As can be seen in the third textbox to the right, the user was successfully authenticated showing that the whole cryptographic handshake occurred successfully.

Cryptographic Handshake Demo

Type in your username

Bob

Submit

Enter name and paste Authentication challenge here

m84HDE9DNH5vL67Ep
NXikSt3XptnFPvNVuyQ
PEEY2bZDwK1AIY8YIujt
Hpqv7qJOfmRVJ9HqiTV
MzcURVgqj0uigNc//bjyJF
oj81kbwoUulQpVFJP4k1
OJM7JG4bMLzi5hXzsxb
ujQaAII7b0vLImqJX6+X
MdHWjG3/fLUeLal=

Submit

Bob

Paste supposed hash of original message here

a236ea7d187a64d46f10
77fc8890a658

Submit

Log in response will show here (An encrypted message)

1
Hello Bob, your public key is -----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDJw1h
am/CzD4E5uUF0mC6v2Rrg
74m1czQrmO+FPJApYD3O1GU0eZHT/gjJZOE2AI7teKWAHj
b8h/IMBIkjhDmDZeaK
Oe8y5qWFGx3+VtlyB6mMnK5ut4Ne5ryKcAd/dJuUKPrvjyIDD
X6k26TJVT+gXEw7 8hgV6RKlJZb064sM+QIDAQAB -----END
PUBLIC KEY-----
Your encrypted challenge is
m84HDE9DNH5vL67EpNXikSt3XptnFPvNVuyQPEEY2bZDwK
1AIY8YIujtHpqv7qJOfmRVJ9HqiTVMzcURVgqj0uigNc//bjyJFoj
81kbwoUulQpVFJP4k1OJM7JG4bMLzi5hXzsxbujQaAII7b0vL
ImqJX6+XMdHWjG3/fLUeLal=

Decryption response will show here (Hash of secret
message)

1This is the decryption
a236ea7d187a64d46f1077fc8890a658

Login result will show here (successful or unsuccessful)
This page will respond with a hash of the decrypted
message. You just sent
a236ea7d187a64d46f1077fc8890a658 Authentication
Successful

Figure 11: Cryptographic Handshake Demo

As can be seen in Figure 12 below, this is the results of an audio signal being transmitted from the mobile device to the computer that the browser is running on. The browser captures the audio signal successfully where the signal can then be decoded into the message. The computer can also successfully transmit a signal back to the mobile device where the mobile device can capture the audio the same way as the personal computer can. It can also be seen that the mobile device indeed is portable and can run the web page on it. It can therefore be taken anywhere the user needs to go.

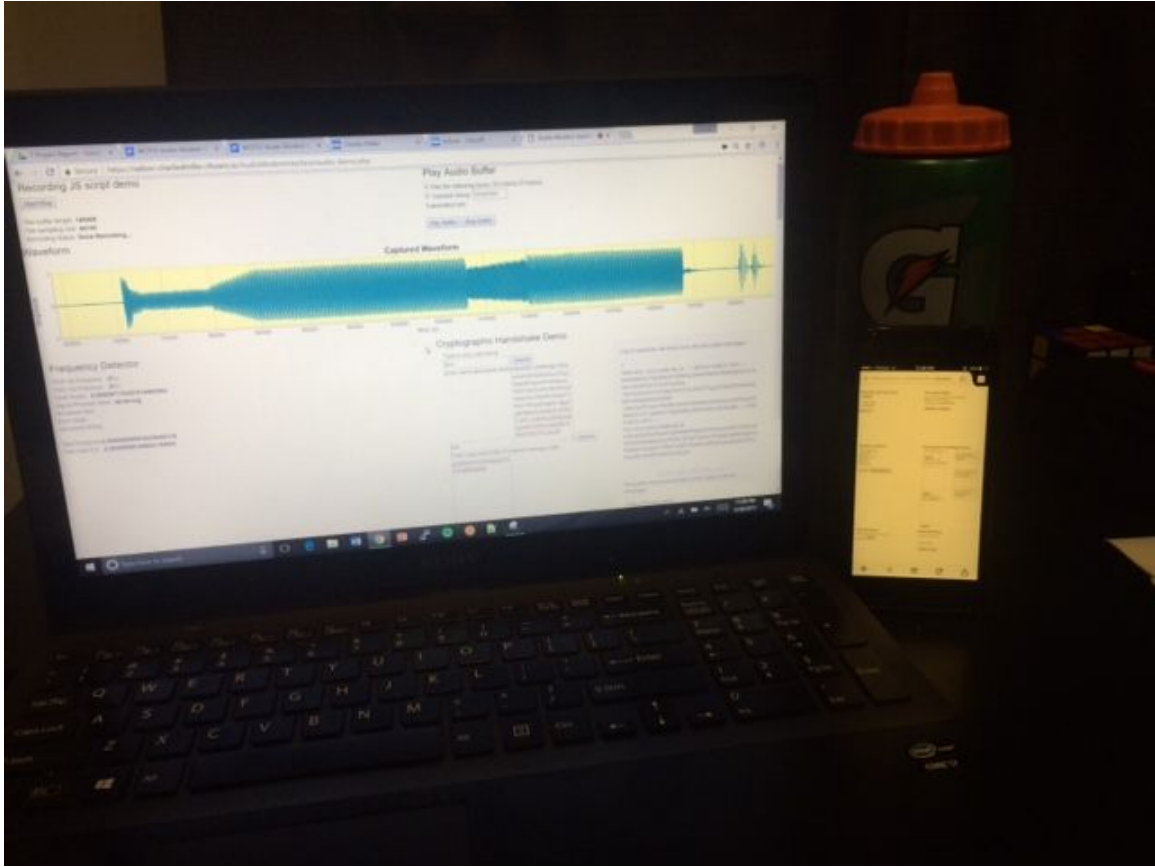


Figure 12: Audio Capture Demo on AMITDP

As shown below, this is test T006 - Modem. From one device to another, a 400 Hz signal is modulated by FSK with a 1700 Hz and 1300 Hz tone. Figure 13 shows that the tones are being recognized by the web page. The data is received through another computer and the results that were captured is displayed as below. This demonstrates that the signal is being modulated.

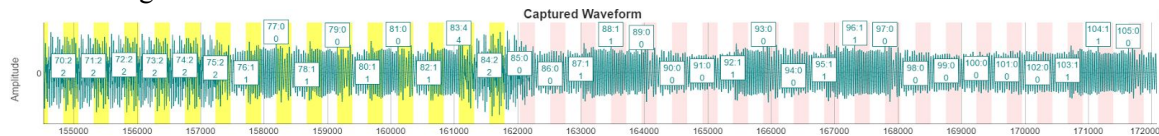


Figure 13: FSK Captured by Webpage. (same as Figure 9)

Frequency Detector

1700 Hz Presence: (11%)
1300 Hz Presence: (40%)
Total Power : **0.00007943393622735834**
Sync Avg 1700 : **4.399434886909361**
Sync Avg 1300 : **0.6000959331411521**
Signal Process State: **recieving**

Figure 14: Presence of the Two FSK Tones

Figure 14 show the signal measured as it is outputted from the computer's headphone jack. Figure 15 shows what the computer captures through the microphone and that is displayed on the web page. Both these figures are part of the test T008 - Data Exchange and shows that data is being transmitted and received by the both devices.

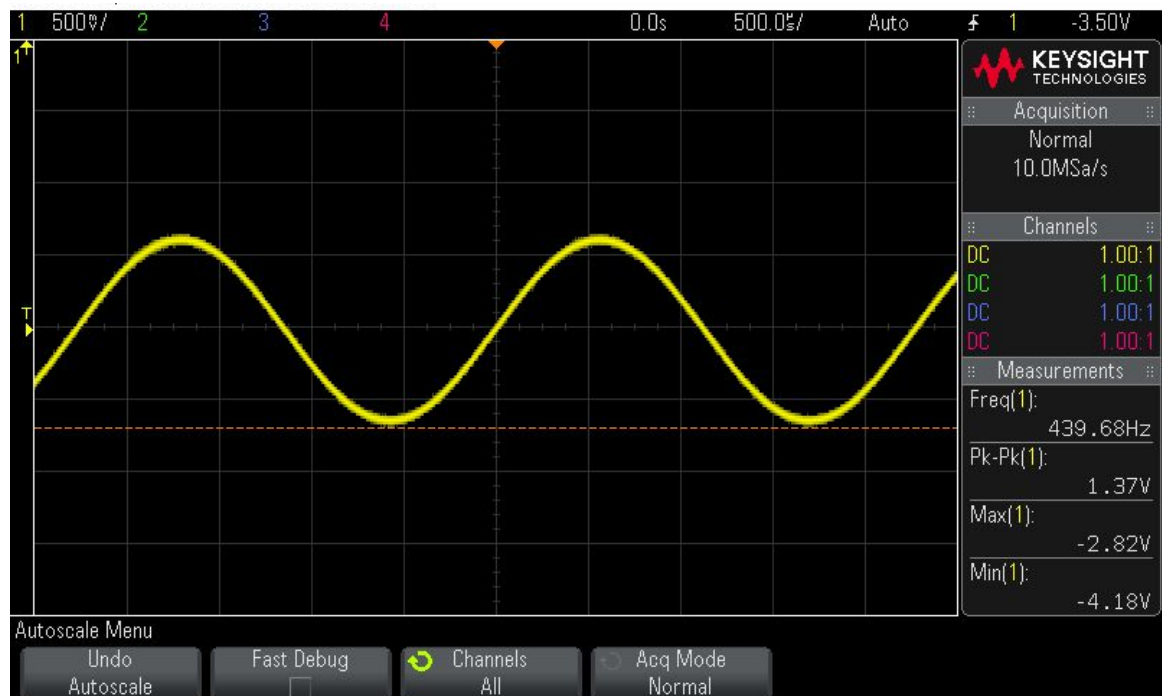


Figure 15: Audio Output on Oscilloscope

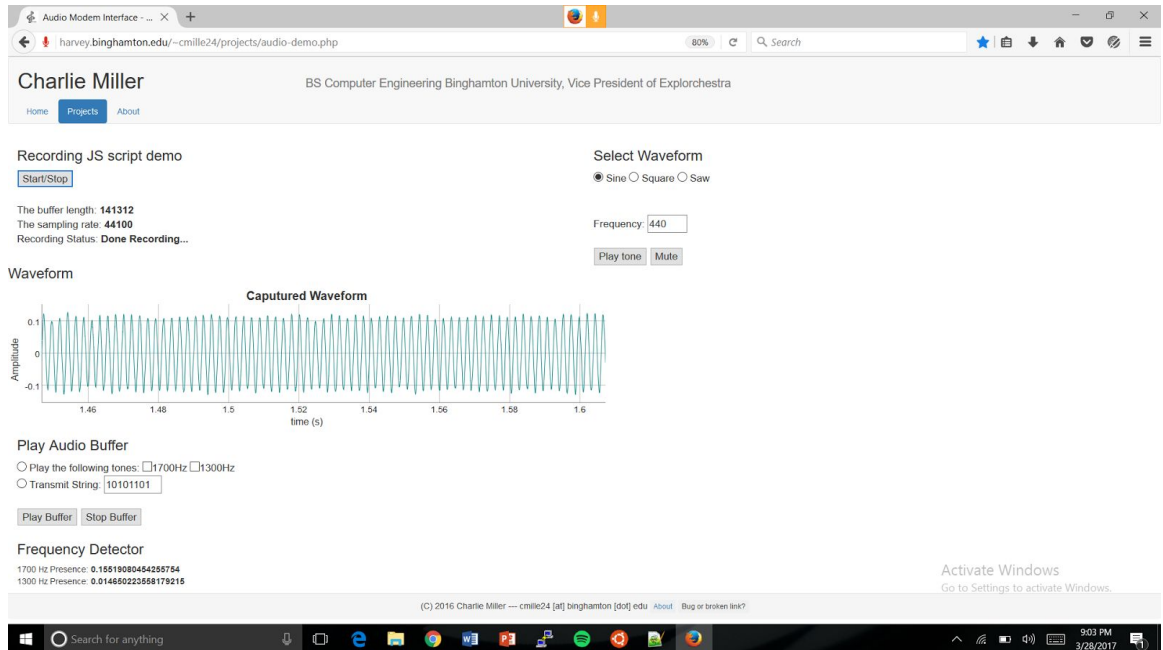


Figure 16: Output Read from Acoustic Channel

Appendix B: Diagrams

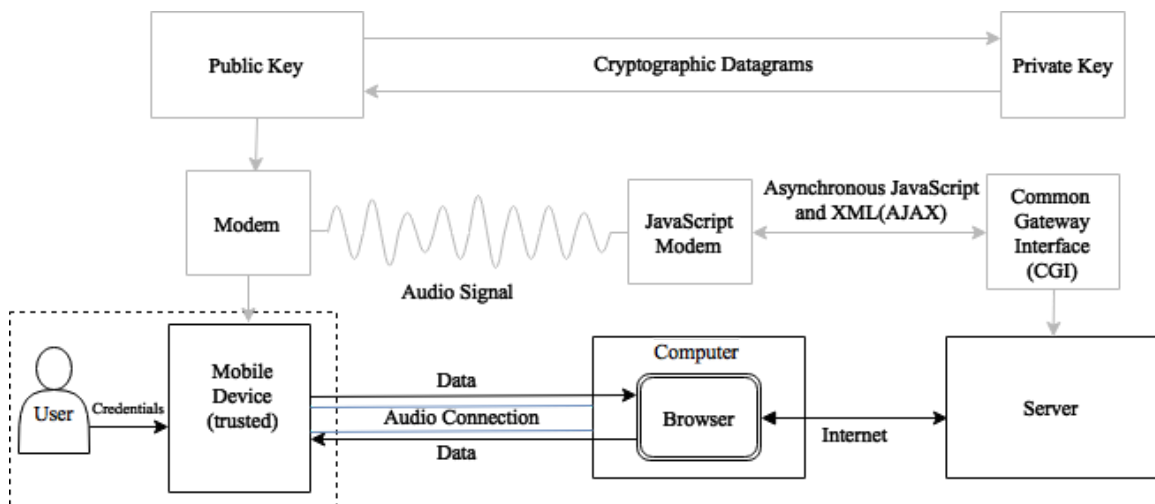


Figure 17: Detailed Operational Context Diagram

Appendix C: Code

C.1 login1.php

```
<?php
if (isset($_POST['decryptedHash']) ){
    $search = $_POST['username'];
    $lines = file('users.txt');
    while (list($key, $line) = each($lines) and !$line_number) {
        $line_number = (strpos($line, $search) !== FALSE) ? $key + 1 : $line_number;
    }

    echo $line_number;

    $dec = base64_decode($_POST['decryptedHash']);
    file_put_contents('./dec.txt', $dec, FILE_USE_INCLUDE_PATH);
    //$pre_encrypt = file_get_contents('./encrypted_data_before_encoding.txt');
    //$priv_key = openssl_get_privatekey('./private.pem');
```

```

//system('openssl rsautl -decrypt -inkey private.pem -in dec.txt -out decrypt.txt');
//$decrypted = file_get_contents('./decrypt.txt');

$fp=fopen("/private{$line_number}.pem","r");
$priv_key=fread($fp,8192);
fclose($fp);

$res = openssl_pkey_get_private($priv_key);

openssl_private_decrypt($dec, $decrypted, $res);

$decrypted_hash = md5($decrypted);

echo "This is the decryption" . "<br>" . $decrypted_hash;

}
else{
    echo "You didn't send anything.";
}
?>

```

C.3 auth3.php

```

<?php
if (isset($_POST['emessage'])) ){
    $message = $_POST['emessage'];
    echo "This page will respond with a hash of the
    decrypted message. You just sent ".$message;
    $original_hash = file_get_contents('./hash.txt',
    FILE_USE_INCLUDE_PATH);
    if($message == $original_hash)
    {
        echo " Authentication Successful";
    }
    else{
        echo " Authentication Unsuccessful";
    }
}
else{
    echo " You didn't send anything.";
}
?>

```

C.4 ajaxDemo.php

```
<?php

// $numberToRespond = (int)$_POST['numberToAdd'] + 1;
// echo $numberToRespond;

    if (isset($_POST['numberToAdd']) )
    {
        if (is_numeric($_POST['numberToAdd'])) {
            $numberToRespond = $_POST['numberToAdd'] + 1;
            echo $numberToRespond;
        }
        else{
            echo "You didn't send a number";
        }
    }
    else
    {
        echo "You didn't send anything.";
    }
?>
```

C.5 audio_demo.php

This is our main file which serves the AMIDP

```
<!DOCTYPE html>
<?php
ini_set('display_errors', 1);
ini_set('display_startup_errors', 1);
error_reporting(E_ALL);

?>
<html>ync = false;
var seenMan = 0;
var breakpoint = 0
//window.watch("manSync",dummy)

function decodeBuffer(buffer, shift, inTransmission){

    var sigWindow = []
    var fullBridgeBitSignal = []
    var sigWindowSize = (samplesPerBaud)/2
```

```

    var sigWindowShift = sigWindowSize/2
    var i = 0
    var j = 0
    var bufferIndex = shift
    var changedGD = false
    //if this isn't the very first buffer to detect a
transmission
    if(inTransmission){
        //Record for debug where samples are being analyzed
        observePoints[OPindex] = [globalIndex + shift -
samplesPerBaud + sigWindowShift, globalIndex + shift -
samplesPerBaud + sigWindowShift + sigWindowSize];
        OPindex++
        //copy over the samples from the last buffer
        for(i = 0; i < bridgeBitSignal.length; i++){
            fullBridgeBitSignal[i] = bridgeBitSignal[i]
        }
        //now copy the other half that takes up the beginning
of this buffer
        j = i;
        for(i = 0; i < shift; i++){
            fullBridgeBitSignal[j] = buffer[i]
            j++
        }
        //Now make a small buffer that is in the middle of
this bit signal
        //which is half the size of a signal and is placed in
the middle of the bit signal
        j = sigWindowShift
        for(i = 0; i < sigWindowSize; i++){
            sigWindow[i] = fullBridgeBitSignal[j]
            j++
        }
        //detect which bit is in the signal and record it to
rx_bits array
        rx_bits[rx_bitsIndex] = detectBit(sigWindow)
        rx_bitsIndex++
    }

    var numChips = 0;
    //Now decode all the bits after shift
    //Do not decode if a full bit signal (baud) cannot fit
    while ((bufferIndex + samplesPerBaud) < buffer.length){
        numChips++;
        //make a small buffer sigWindow that's in the middle
of a bit signal (same as before)
        j = bufferIndex + sigWindowShift

```

```

        observePoints[OPindex] =
[globalIndex+j,globalIndex+j+sigWindowSize];
        OPindex++;
        for(i = 0; i < sigWindowSize; i++){
            sigWindow[i] = buffer[j]
            j++
        }

        //decode the bit -- same as before
        var bit = detectBit(sigWindow);
        rx_bits[rx_bitsIndex] = bit;
        rx_bitsIndex++

        // if(inTransmission && seenMan && bit == 2 &&
changedGD == false){
            //    gotData++;
            //    changedGD = true;
            // }

            //accumulate to find average of power levels for each
freq during the sync signal
            //Will be used to distinguish between a sync signal
and a bad window
            if(rx_bits.length <= 50){
                // syncAvg1300 += pow1300[powArrIndex-1];
                // syncAvg1700 += pow1700[powArrIndex-1];
            }

            // Do fine syncing if we are still there. Otherwise
just go onto the the next chip
            if(manSync){
                //In order for us to shift, there must be a
noticeable difference in the power levels of either freq
                //Additionally, it's gotta be a bad sync! There will
exist some of both signals in a badly synced window
                // The difference of the power levels can be 50% and
50%, diff of 0
                //Shift the window a quarter size of the chip so that
it'll be fully in one chip
                var curPow1700 = pow1700[powArrIndex-1]
                var curPow1300 = pow1300[powArrIndex-1]
                var diff1700 = (curPow1700 > syncAvg1700+5) ||
(curPow1700 < syncAvg1700-5)
                var diff1300 = (curPow1300 > syncAvg1300+5) ||
(curPow1300 < syncAvg1300-5)
                var closeToZero = ((curPow1300 < 5) || (curPow1300 <
5))

```

```

        var badSync = (Math.abs(curPow1300 - curPow1700) < 40
&& !closeToZero)
        if(seenMan == 1){
            if(badSync)
                bufferIndex -= samplesPerBaud/2
            manSync = false
        }

        if(diff1700 || diff1300)
            seenMan++
        }
        bufferIndex += samplesPerBaud
    }

    //We're done syncing with manchester. It'll be false for
the rest of the transmission
    if(manSync && seenMan)
        manSync = false;

    //The first decoded buffer shall be all a sync signal.
We take the average of the values found in it
    //This is to distinguish between a sync signal and a bad
window
    if(rx_bits.length >= 50 && !manSync && seenMan == 0){
        for(var k = 0; k < 50;k++){
            syncAvg1300 += pow1300[k];
            syncAvg1700 += pow1700[k];
        }
        syncAvg1700 = syncAvg1700/(50);
        syncAvg1300 = syncAvg1300/(50);
        document.getElementById("sync1700").innerHTML =
syncAvg1700
        document.getElementById("sync1300").innerHTML =
syncAvg1300
        //We shall sync with a manchester signal in the next
buffer
        manSync = true;
    }

    //Prepare the remaining samples for the bridgeBitSignal
- copy it there
    bridgeBitSignal = []
    i = 0
    for(j = bufferIndex; j < buffer.length; j++){
        bridgeBitSignal[i] = buffer[j]
        i++
    }

```

```

    //return where the next full signal will start. This
    will be the next singal's shift
    return (bufferIndex + samplesPerBaud) % bufferSize
}

function dummy(){
    breakpoint++;
}

var pow1700 = [];
var pow1300 = [];
var powArrIndex = 0;

confidence = []
cIndex = 0

function detectBit(buffer){
    phasor1700 = detectFreq(buffer, baudRate, 1700)
    phasor1300 = detectFreq(buffer, baudRate, 1300)

    pow1700[powArrIndex] = phasor1700.percent;
    pow1300[powArrIndex] = phasor1300.percent;
    powArrIndex++;

    var higher = (phasor1700.percent > phasor1300.percent) ?
1:0;
    var close = (Math.abs(phasor1700.percent -
phasor1300.percent) <= 30)
    var syncFar = (Math.abs(syncAvg1700 - syncAvg1300) >=
40)
    var sInterpretWin = syncFar ? 10 : 10;
    var close1700 = ((phasor1700.percent <
syncAvg1700+sInterpretWin) && (phasor1700.percent >
syncAvg1700-sInterpretWin))
    var close1300 = ((phasor1300.percent <
syncAvg1300+sInterpretWin) && (phasor1300.percent >
syncAvg1300-sInterpretWin))
    var closeToSync = (close1700 && close1300)
    var halfCloseToSync = (close1700 != close1300)

    if(closeToSync || !seenMan){
        return 2 // this is most likely a sync bit
    }

    if(close){
        confidence[cIndex] = 0
        cIndex++
        if(halfCloseToSync){

```

```

        return 4 //we're not sure but it probably is a sync
bit. Used for determining the end of transmission
    }
    return 3 // we're not sure
}
else{
    confidence[cIndex] = 1
}
cIndex++

return higher //It looks like a databit

// if((phasor1300.percent > 3.0) &&
(Math.floor(phasor1700.percent) == 0)){
    //    gotData = true;
    //    return 0;
    // }
    // else if((phasor1700.percent > 3.0) &&
(Math.floor(phasor1300.percent) == 0)){
    //    gotData = true;
    //    return 1;
    // }
    // else{
    //    //this is a sync bit so just return 2 which will be
discarded
    //    return 2;
    // }
}

function testDetectFreq2(){
    //implementation of the goertzel alg
    var buffer = [];
    var i =0;
    var sampleFrequency = audioContext.sampleRate/1700;

    while(i < 2048 - 1){
        buffer[i] = 0.0001 * Math.sin(i / (sampleFrequency /
(Math.PI*2)))
        i++;
    }

    var k = 0.5 + (2048*1700/audioContext.sampleRate)
    var w = (2*Math.PI/2048) * k;
    var cosine = Math.cos(w);
    var sine = Math.sin(w);
    var coeff = 2 * cosine;

```

```

var Q0 = 0, Q1 = 0, Q2 = 0;

var totPower = 0;

i = 0;
while(i < 2048 - 1){
    Q0 = coeff * Q1 - Q2 + buffer[i];
    Q2 = Q1;
    Q1 = Q0;
    totPower += buffer[i] * buffer[i]
    i++
}

var real = (Q1 - Q2 * cosine)
var imag = (Q2 * sine)
var mag = Math.sqrt(real * real + imag * imag);

document.getElementById("test-Presence").innerHTML =
mag;
document.getElementById("test-Power").innerHTML =
totPower;
}

function testDetectFreq(){
    var buffer = [];
    var i = 0;
    var sampleFrequency = audioContext.sampleRate/1700;
    var sampleFrequency2 = audioContext.sampleRate/1300;
    var maxAmp = 0;
    while(i < 2048 - 1){
        buffer[i] = 0.01 * Math.sin(i / (sampleFrequency /
(Math.PI*2))) + 0.01 * Math.sin(i / (sampleFrequency2 /
(Math.PI*2)))
        i++;
    }

    i = 0;
    var Sine = 0;
    var Cosine = 0;
    var totPower = 0;

    // while (i < 2048 - 1){
    //     totPower += buffer[i] * buffer[i]
    //     if(Math.abs(buffer[i]) > maxAmp){
    //         maxAmp = Math.abs(buffer[i]);
    //     }
    //     i++
    // }

```

```

    i = 0;
    //var sinAmp = Math.sqrt(totPower);
    while (i < 2048 - 1){
        totPower += buffer[i] * buffer[i]
        Sine += buffer[i] * Math.sqrt(2) * Math.cos(i /
(sampleFrequency / (Math.PI*2)))
        Cosine += buffer[i] * Math.sqrt(2) * Math.sin(i /
(sampleFrequency / (Math.PI*2)))
        i++
    }

    totPower = totPower/2048
    Sine = Sine/2048
    Cosine = Cosine/2048
    var power = Sine * Sine + Cosine * Cosine;
    document.getElementById("test-Power").innerHTML =
totPower;
    var mag = Math.sqrt(power)
    document.getElementById("test-Presence").innerHTML =
power;

}

// var sine_dy = new
Dygraph(document.getElementById("sine-viz"),
// sinew,
// {
//   title: 'Generated Sine',
//   ylabel: 'Amplitude',
//   xlabel: 'time (s)',
//   labels: [ "Time", "Amplitude" ]
// });

// var buffer_dy = new
Dygraph(document.getElementById("buffer-viz"),
// buffer_graphdata,
// {
//   title: 'Captured Buffer',
//   ylabel: 'Amplitude',
//   xlabel: 'time (s)',
//   labels: [ "Time", "Amplitude" ]
// });

function detectFreq(buffer, baud, freq){
    var sample = []

```

```

var bufIndex = 0
//var normalizedfreq = freq / audioContext.sampleRate;
var sampleFrequency = audioContext.sampleRate/ freq
var Sine = 0
var Cosine = 0
var power = 0
var totPower = 0;
var i = 0;

// var sinew = [[0,0]]
// var buffer_graphdata = [[0,0]]
// var time = []

while (i < buffer.length - 1){

    // time[i] = i/audioContext.sampleRate;
    // sinew[i] = [time[i],Math.sin(i / (sampleFrequency
/ (Math.PI*2)))]
    // buffer_graphdata[i] = [time[i],buffer[i]]

    totPower += buffer[i] * buffer[i]
    Sine += buffer[i] * Math.sqrt(2) * Math.cos(i /
(sampleFrequency / (Math.PI*2)))
    Cosine += buffer[i] * Math.sqrt(2) *Math.sin(i /
(sampleFrequency / (Math.PI*2)))
    i++
}

totPower = totPower/buffer.length
Sine = Sine/buffer.length
Cosine = Cosine/buffer.length

// buffer_dy.updateOptions( { 'file': buffer_graphdata }
);
// sine_dy.updateOptions( { 'file': sinew } );

//get angle in Radians -PI to PI
var angle = Math.atan2(Cosine,Sine)

// if (angle < 0) {
//     angle += 2 * Math.PI
// }

//Map Radians to 0 to samples/period
var phi = (angle * sampleFrequency)/(2 * Math.PI)

var mag = Math.sqrt(Sine * Sine + Cosine * Cosine)

```

```

    power = Sine * Sine + Cosine * Cosine
    //totPower = totPower/(buffer.length - 1)
    return new Phasor(freq, mag,phi,power, totPower)
}

function findChip2(phasor1, phasor2){
    //posL is intended to be the position for the 1300 chip
    var posL = phasor1.phi
    var posS = phasor2.phi
    //var chipStart = posS + 52 * (posL-posS)
    //var chipStart = posS + 51 * (posL-posS)
    var chipStart = parseInt(posL + 51 * (posL-posS))
    chipStart = chipStart % samplesPerBaud

    //Make the first shift be positive (Needed so buffer
    indices are positive)
    while(chipStart < 0){
        chipStart = chipStart + samplesPerBaud
    }
    return chipStart
}

function findChip(phasor1, phasor2)
{
    var shift_p1 = phasor1.phi
    var shift_p2 = phasor2.phi
    var shift_p1_int = parseInt(shift_p1)
    var shift_p2_int = parseInt(shift_p2)

    while (shift_p2_int != shift_p1_int)
    {
        if(shift_p2_int == shift_p1_int + 1)
            break;
        if(shift_p2_int == shift_p1_int - 1)
            break;

        if(shift_p1 < shift_p2){
            shift_p1 += phasor1.sampleFrequency
            shift_p1_int = parseInt(shift_p1)
        }
        else{
            shift_p2 += phasor2.sampleFrequency
            shift_p2_int = parseInt(shift_p2)
        }
    }

    while(shift_p2_int - samplesPerBaud > 0){
        shift_p2_int -= samplesPerBaud
    }
}

```

```

    }

    return shift_p2_int
}

function detectSync(buffer, phasor1700, phasor1300)
{
    //Detect the presence of our carrier frequencies
    var present1700 = (phasor1700.percent > 1.0);
    var present1300 = (phasor1700.percent > 1.0);

    return (present1700 && present1300)
}

// init once the page has finished loading.
window.onload = initAudio;

</script>

<?php

?>
</body>
</html>
<head>
    <title>Audio Modem Interface - Technology Demo </title>
    <link rel="icon" href="image/favicon.jpg">
    <!--<script
src="js/web-audio-recorder-js-master/lib-minified/WebAudio
Recorder.min.js"></script> -->
    <script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/dygraph/1.1.1/
dygraph-combined.js"></script>

    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/
bootstrap.min.css";>

    <!-- jQuery library -->
    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/j
query.min.js"></script>

    <!-- Latest compiled JavaScript -->

```

```

    <script src=
"https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootst
rap.min.js"></script>

</head>
<body>
    <div id='page-wrapper'>
        <!--NavBar-->
        <?php

        ?>
        <!--End NavBar-->
        <div id="welcome">
        <div id="prelim-demos">
        <div id = "recording-demo" class = "col-xs-6">
            <h3>Recording JS script demo</h3>

            <button type="button" id = "recButton"
onclick="audRecord(this)">Start/Stop</button>
            <button type="button" id = "forceButton"
onclick="forceStopRec()">Force Stop</button>
            <br>
            <br>

            The buffer length: <b
id="bufferLengthDisplay"></b><br>
            The sampling rate: <b id="sampleRate"></b><br>
            Recording Status: <b id =
"waveform-message"></b>
        </div>
        <div id = "Audio Buffer Demo" class = "col-xs-6">
        <h3>Play Audio Buffer</h3>
            <input type="radio" value="tones"
id="bufChoice" name = "bufChoice"> Play the following
tones:
                <input type="checkbox" name="tonevals"
value="1700" onclick="updateFreqsToPlay()">1700Hz
                <input type="checkbox" name="tonevals"
value="1300" onclick="updateFreqsToPlay()">1300Hz<br>
                <input type="radio" value="bits" id="bufChoice"
name = "bufChoice"> Transmit String:
                <input type="text" size="8" value="10101101"
id="sequence"><br>
                Transmitted bits: <b
id="transmittedBits"></b><br><br>
                <button type="button"
onclick="audioBufferDemo()">Play Buffer</button>

```

```

        <button type="button"
onclick="playTestBits()">Play Test Bits</button>
    </div>
</div>
<div id = "waveform-demos">
    <div id = "zoomed-waveform-wrapper">
        <h3>Waveform</h3>
        <div id="chart_div" style="height: 230px;
width:auto"></div>
    </div>
</div>

    <div id = "Frequency-detect" class = "col-xs-6">
        <h3>Frequency Detector</h3>
        1700 Hz Presence: <b id="1700presence"></b>
        &nbsp;  <b id="1700percent"></b>%) <br>
        1300 Hz Presence: <b
id="1300presence"></b>&nbsp;  <b
id="1300percent"></b>%)<br>
        Total Power      : <b id="totalPower"></b><br>
        Sync Avg 1700    : <b id="sync1700"></b><br>
        Sync Avg 1300    : <b id="sync1300"></b><br>
        Signal Process State: <b
id="signal-Process-State"></b><br>
        Received Bits: <b id="receivedBits"></b><br>
        Received String: <b
id="received-string"></b><br>
        # Bits Recieved: <b id="numRecieved"></b><br>
        Error Rate: <b id="error-rate"></b><br>
        Total E-Rate: <b id="Terror-rate"></b><br>
        Test Power: <b id="test-Power"></b><br>
        Test Total power: <b
id="test-Presence"></b><br>

    </div>

    <div id = "crypt-handshakes" class = "col-xs-6">
        <h3>Cryptographic Handshake Demo</h3>
        <div id = "crypt-inputs" class = "col-xs-6">
            Type in your username <br>
            <input type="text" id="username">
            <button type="button"
onclick="sendLoginReq()">Submit</button><br>
            Paste Authentication challenge here<br>
            <textarea style="height:200px"
id="encrypted-message"></textarea>

```

```

        <button type="button"
onclick="sendDecryptReq()">Submit</button><br>
        Paste supposed hash of original message
here<br>
        <textarea style="height:200px"
id="decrypted-hash"></textarea>
        <button type="button"
onclick="sendAuthReq()">Submit</button><br>
    </div>
    <div id = "crypt-results" class = "col-xs-6"
style="word-wrap: break-word;">
        <div id = "login-response-wrapper" class =
"well">
            <div>Log in response will show here (An
encrypted message)</div><br>
            <div id = "login-response"></div>
            </div>
            <div id = "decrypt-response-wrapper" class =
"well">
                <div>Decryption response will show here (Hash
of secret message)</div><br>
                <div id = "decrypt-response"></div>
                </div>
                <div id = "auth-result-wrapper" class =
"well"><br>
                    <div>Login result will show here (successful or
unsuccessful)</div>
                    <div id = "auth-result"></div>
                    </div>
                </div>
            </div>

    <div id = "AJAX call Demo" class = "col-xs-3">
    <h3>AJAX Call Demo</h3>
    Input a number and the server will add one to it <br>
    <input type="text" size="8" id="ajaxNumber">
    <button type="button"
onclick="AJAXdemo()">Submit</button>
    <div id = "AJAX-result">
        The result will display here
    </div>
    </div>

    <div id = "waveform-gen" class = "col-xs-3">
    <div id="wave-controls">
        <h3>Select Waveform</h3>
        <div id="type-select" onclick = "changeType()">

```

```

        <input type="radio" value="sine" id="wave-type"
name = "wave-type"> Sine
        <input type="radio" value="square"
id="wave-type" name = "wave-type"> Square
        <input type="radio" value="sawtooth"
id="wave-type" name = "wave-type"> Saw
    </div>
    <br>
    <br>
    Frequency: <input type="text" size="4"
value="440" id="freq">
    <br>
    <br>
    <button id="playButton" onclick="playOsc()">
Play tone</button>
    <button id="muteButton"
onclick="muteOsc()">Mute</button>
    </div>
    <div id = "wave-status">
    </div>
    </div>

    <div id = "waveform-gen" class = "col-xs-6">
    <h3>Test system accuracy</h3>
    Select rx or tx <br>
    <input type="radio" value="tx" id="test-role" name =
"test-role"> tx
    <input type="radio" value="rx" id="test-role" name =
"test-role"> rx
    <button type="button"
onclick="testTxSystem()">Generate Bits</button>
    <button type="button"
onclick="testSystem()">Run</button><br>
    <button type="button" style="height:52;width:120"
onclick="continueTest()">Continue</button>
    <div class="col-xs-6">
        <p>TX Status</p>
        <div id="tx-test-status"></div>
    </div>
    <div class="col-xs-6">
        <p>RX Status</p>
        <div id="rx-test-status"></div>
    </div>
    </div>
    </div>

</div>

<script>

```

```

var context = null, sampleRate = null, rec_volume = null,
audioInput = null;
var areRecording = false;
var recordingLength = 0;
var bufferSize = 2048;
var leftchannel = [];
var leftBuffer = [];
var AudioContext = window.AudioContext ||
window.webkitAudioContext;
var audioContext = null;
var oscillator;
var amp;

var wav_message =
document.getElementById('waveform-message');

var channels = null;
var bufferLength = null;
var myAudioBuffer = null;

var globalSource = null

function AJAXdemo(){
    var numberToAdd =
document.getElementById("ajaxNumber").value
    var requestStr = "numberToAdd=" + numberToAdd
    // var xmlhttp = new XMLHttpRequest();
    // xmlhttp.onreadystatechange = function() {
    // if (this.readyState == 4 && this.status == 200) {
    //
    document.getElementById("AJAX-result").innerHTML =
this.responseText;
    // }
    // }
    // xmlhttp.open("POST", "projects/ajaxDemo.php", true);
    // xmlhttp.send(requestStr);
    $.ajax({
        type: "POST",
        url: "ajaxDemo.php",
        data: requestStr,
        cache: false,
        success: function(result){
            document.getElementById("AJAX-result").innerHTML =
result
        }
    });
}

```

```

function sendLoginReq(){
    var uname = document.getElementById("username").value
    //var udata = document.getElementById("data").value
    var newdata = { 'username': uname };// , 'data': udata};
    $.ajax({
        type: "POST",
        url: "login1.php",
        data: newdata,
        cache: false,
        success: function(result){
            document.getElementById("login-response").innerHTML =
result
        }
    });
}

function sendDecryptReq(){
    var emsg =
document.getElementById("encrypted-message").value
    var uname = document.getElementById("username2").value
    var requestStr = 'decryptedHash=' +
encodeURIComponent(emsg) + "&username=" + uname
    $.ajax({
        type: "POST",
        url: "decrypt2.php",
        data: requestStr,
        cache: false,
        success: function(result){
            document.getElementById("decrypt-response").innerHTML
= result
        }
    });
}

function sendAuthReq(){
    var dhash =
document.getElementById("decrypted-hash").value
    var requestStr = "emessage=" + dhash
    $.ajax({
        type: "POST",
        url: "auth3.php",
        data: requestStr,
        cache: false,
        success: function(result){
            document.getElementById("auth-result").innerHTML =
result
        }
    });
}

```

```

}

// Create an oscillator and an amplifier.
function initAudio()
{
    // Use audioContext from webaudio_tools.js

    console.log("Trying to make oscillator");
    audioContext = new AudioContext();
    testDetectFreq();
    samplesPerBaud = audioContext.sampleRate/baudRate
    oscillator = audioContext.createOscillator();
    //fixOscillator(oscillator);
    oscillator.frequency.value = 440;
    amp = audioContext.createGain();
    amp.gain.value = 0;

    // Connect oscillator to amp and amp to the mixer of
    the audioContext.
    // This is like connecting cables between jacks on a
    modular synth.
    oscillator.connect(amp);
    amp.connect(audioContext.destination);
    oscillator.start(0);

    // Older browsers might not implement mediaDevices at all,
    so we set an empty object first
    if (navigator.mediaDevices === undefined) {
        navigator.mediaDevices = {};
    }

    // Some browsers partially implement mediaDevices. We
    can't just assign an object
    // with getUserMedia as it would overwrite existing
    properties.
    // Here, we will just add the getUserMedia property if
    it's missing.
    if (navigator.mediaDevices.getUserMedia === undefined) {
        navigator.mediaDevices.getUserMedia =
        function(constraints) {

            // First get ahold of the legacy getUserMedia, if
            present
            var getUserMedia = (navigator.getUserMedia =
            navigator.getUserMedia || navigator.webkitGetUserMedia ||
            navigator.mozGetUserMedia ||
            navigator.msGetUserMedia);

```

```

        // Some browsers just don't implement it - return a
        rejected promise with an error
        // to keep a consistent interface
        if (!getUserMedia) {
            return Promise.reject(new Error('getUserMedia is not
            implemented in this browser'));
        }

        // Otherwise, wrap the call to the old
        navigator.getUserMedia with a Promise
        return new Promise(function(resolve, reject) {
            getUserMedia.call(navigator, constraints, resolve,
            reject);
        });
    }

    navigator.mediaDevices.getUserMedia({audio:
    true}).then(gotAudioStream).catch(noAudioerr);

    //INIT for buffer player
    // Stereo
    channels = 2;
}

//-----Functionality for Audio Buffer
Player-----
var freqsToPlay = [];

freqsToPlay.contains = function(obj) {
    for (var i = 0; i < freqsToPlay.length; i++) {
        if (freqsToPlay[i] === obj) {
            return true;
        }
    }
    return false;
}

function updateFreqsToPlay() {
    $(':checkbox').each(function() {
        var tag = $(this).attr('value');
        //tag.replace(/^quicklinksch_/, '');
        if( $(this).is(':checked')){
            if(!freqsToPlay.includes(tag)){
                freqsToPlay.push(tag)
            }
        }
        else{
            var index = freqsToPlay.indexOf(tag);

```

```

        if (index > -1) {
            freqsToPlay.splice(index, 1);
        }
    });
}

function audioBufferDemo() {
    var choice = $('input[name=bufChoice]:checked').val();
    if(choice == "tones"){
        playTones()
    }
    else
    {
        playString()
    }
}

function playTones() {

    //Number of samples per frame
    bufferLength = audioContext.sampleRate * 3;
    myAudioBuffer = audioContext.createBuffer(2,
bufferLength, audioContext.sampleRate)

    for (var channel = 0; channel < channels; channel++) {
        // This gives us the actual ArrayBuffer that contains
the data
        var nowBuffering =
myAudioBuffer.getChannelData(channel);
        for (var i = 0; i < bufferLength; i++) {
            var sF1700 = audioContext.sampleRate / 1700
            var sF1300 = audioContext.sampleRate / 1300
            var currentSine = null;
            if(freqsToPlay.includes("1700") &&
freqsToPlay.includes("1300")){

                currentSine = Math.sin(i / (sF1700 / (Math.PI*2))) +
Math.sin(i / (sF1300 / (Math.PI*2)))
            }
            else if (freqsToPlay.includes("1700")) {
                currentSine = Math.sin(i / (sF1700 / (Math.PI*2)))
            }
            else if (freqsToPlay.includes("1300")) {
                currentSine = Math.sin(i / (sF1300 / (Math.PI*2)))
            }
        }
    }
}

```

```

        nowBuffering[i] = currentSine
    }
}
playBuffer()
}

var compbitarray = [];
var compbitarrayIndex = 0;

function playString(){
    var strToPlay =
document.getElementById("sequence").value;
    var bits = []
    bits = encodeString(strToPlay)
    playBits(bits)
    //[0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1]
}

function encodeString(str){
    var bytes = [];
    var i = 0;
    for (i = 0; i < str.length; i++) {
        bytes[i] = str.charCodeAt(i);
        //bytes = bytes.concat([code & 0xff, code / 256 >>>
0]);
    }

    var byteIndex = 0;
    var encBitArr = [];
    var bitIndex = 0;
    for(i = 0; i < bytes.length; i++){
        for(var j = 0; j < 8; j++){

            encBitArr[bitIndex] = (bytes[i] >>> (7 - j) & 1);
            bitIndex++
        }
    }

    return encBitArr
}

function playBits(bits) {
    document.getElementById("transmittedBits").innerHTML =
formatBits(bits);

    compbitarray = bits
    var baud = baudRate;

```

```

var sF1700 = audioContext.sampleRate / 1700
var sF1300 = audioContext.sampleRate / 1300

//Determine how many samples the whole buffer will be
//We want to include the sync bauds as well, filling 2
buffers on each side with syncing bauds (signal)
var nSyncBauds = Math.ceil(bufferSize/baudRate) * 6;
var nManchesterBauds = 8
var nStartBauds = 2
//var halfSyncBauds = nSyncBauds/2;
var nbauds = bits.length + nSyncBauds + nManchesterBauds
+ nStartBauds

bufferLength = nbauds * samplesPerBaud;
myAudioBuffer = audioContext.createBuffer(2,
bufferLength, audioContext.sampleRate)

//Append sync and manchester signals to the bit array
var header = []
var nheader_bits = (4/6) * nSyncBauds + nManchesterBauds
for(var i = 0; i < nheader_bits; i++){
    //Fill a lot with sync bauds, then manchester it up
    if(i < (2/3) * nSyncBauds){
        header[i] = 2
    }
    else{
        if(header[i - 1] == 0){
            header[i] = 1;
        }
        else{
            header[i] = 0;
        }
    }
}
var startbits = []
for(var i = 0; i < nStartBauds; i++){
    startbits[i] = 2
}

header = header.concat(startbits);

//Append the sync signals at the end
var footer = []
var nfooter_bits = (1/3) * nSyncBauds
for(var i = 0; i < nfooter_bits; i++){
    footer[i] = 2
}

```

```

bits = (header.concat(bits)).concat(footer)

for (var channel = 0; channel < channels; channel++) {
    // This gives us the actual ArrayBuffer that contains
the data
    var nowBuffering =
myAudioBuffer.getChannelData(channel);
    var currentSine = null;
    var bitarrIndex = 0
    var syncIndex = 0;
    var currentBit = null
    var i = 0
    var starting = true
    var ending = false
    for (var k = 0; k < nbauuds; k++) {

        //Determine if we are syncing yet (either beginning
or ending the signal)
        // if( starting || ending ){
        //     currentBit = 2
        // }
        // else{
        //     currentBit = bits[bitarrIndex]
        //     bitarrIndex++
        //     compbitarray[compbitarrayIndex] = currentBit;
        //     compbitarrayIndex++
        // }
        //bitarr[bitarrIndex] = currentBit
        currentBit = bits[bitarrIndex]
        bitarrIndex++

        //Fill baud with the kind of signal we want
        for (var j = 0; j < samplesPerBaud; j++) {
            if (currentBit == 1) {
                currentSine = Math.sin(i / (sF1700 / (Math.PI*2)))
            }
            else if(currentBit == 0) {
                currentSine = Math.sin(i / (sF1300 / (Math.PI*2)))
            }
            else{
                currentSine = Math.sin(i / (sF1700 / (Math.PI*2))) +
Math.sin(i / (sF1300 / (Math.PI*2)))
            }
            nowBuffering[i] = currentSine
            i++
        }
    }
}

```

```

        //Overhead: Watching for beginning and end and
        updating the indicies of the bits and bytes
        // if (starting){
        //     syncIndex++
        //     if(syncIndex == (2/3) * nSyncBauds){
        //         syncIndex = 0
        //         starting = false
        //     }
        //     continue
        // }

        // if(!ending){
        //     //bitIndex++
        //     ending = (bitarrIndex == bits.length);
        //     continue;
        // }

        // if (ending){
        //     syncIndex++
        //     if(syncIndex == halfSyncBauds){
        //         syncIndex = 0
        //     }
        // }

        // if(bitIndex == 8)
        // {
        //     bitIndex = 0
        //     byteIndex++
        // }
    }
    playBuffer()
}

var playing = false
var source

function playBuffer(){
    // Get an AudioBufferSourceNode.
    source = audioContext.createBufferSource();
    source.addEventListener("ended",function(){
        playing = false
    });
    // This is the AudioNode to use when we want to play an
    AudioBuffer
    // set the buffer in the AudioBufferSourceNode
    source.buffer = myAudioBuffer;
}

```

```

    // connect the AudioBufferSourceNode to the
    // destination so we can hear the sound
    source.connect(audioContext.destination);

    playing = true

    globalSource = source
    // start the source playing
    source.start();
    //source.stop();

}

function stopBuffer(){
    source.stop()
}

//-----Oscillator Demo -----

// Set the frequency of the oscillator and start it
running.
function startTone(frequency,type)
{
    if(type == undefined)
        type = "sine";

    console.log("Trying to play");
    oscillator.frequency.value = frequency;
    oscillator.type = type;
    amp.gain.value = 0.5;

}

function changeType()
{
    var type = $('input[name=wave-type]:checked').val();
    oscillator.type = type;
}

function muteOsc()
{
    amp.gain.value = 0.0;
}

function playOsc()
{

```

```

var freq = document.getElementById("freq").value;
freq = Number(freq);
if (isNaN(freq))
{
    document.getElementById("wave-status").innerHTML =
"Frequency entered isn't a Number";
    return;
}

//var type = document.getElementById("wave-type").value;
var type = $('input[name=wave-type]:checked').val();

startTone(freq,type);
}

//-----Functionality for
recorder-----
class Phasor {
    constructor(freq, mag, phi, power, totPower) {
        this.freq = freq;
        this.mag = mag;
        this.phi = phi;
        this.power = power
        this.totPower = totPower
        this.percent = (power/totPower) * 100
        this.sampleFrequency = audioContext.sampleRate/freq
    }
}

noAudioerr = function(e) {
    console.log("Audio source is not connected or is
blocked");
};

gotAudioStream = function(e) {
    console.log("Audio source is connected!");

    // retrieve the current sample rate to be used for
WAV packaging
    sampleRate = audioContext.sampleRate;
    document.getElementById('sampleRate').innerHTML =
sampleRate;

    // creates a gain node

```

```

        rec_volume = audioContext.createGain();

        // creates an audio node from the microphone incoming
stream
        audioInput = audioContext.createMediaStreamSource(e);

        // connect the stream to the gain node
        audioInput.connect(rec_volume);
        recorder =
audioContext.createScriptProcessor(bufferSize, 1, 1);
        //recorder = new Recorder(rec_volume);

        recorder.onaudioprocess = function(e){
        if(!areRecording) return;
            //console.log ('recording');
            var left = e.inputBuffer.getChannelData (0);
        //process# buffers
            buff = new Float32Array (left)
            processBuffer(buff)
            leftchannel.push (buff);
            recordingLength += bufferSize;

document.getElementById("bufferLengthDisplay").innerHTML =
recordingLength;
            console.log('recording')
        };

        // we connect the recorder
        rec_volume.connect (recorder);
        recorder.connect (audioContext.destination);
    }

function mergeBuffers(channelBuffer, recordingLength){
    var result = new Float32Array(recordingLength);
    var offset = 0;
    var lng = channelBuffer.length;
    for (var i = 0; i < lng; i++){
        var buffer = channelBuffer[i];
        result.set(buffer, offset);
        offset += buffer.length;
    }
    return result;
}

var global_dygraph = null;

```

```

function drawChart() {

    // Create the data table.
    var time = [];
    var chart_data = [[]];
    var i = 0;
    for(i = 0; i < leftBuffer.length;i++)
    {
        time[i] = i///sampleRate;
        chart_data[i] = [time[i],leftBuffer[i]];
    }

    var my_dygraph = new
    Dygraph(document.getElementById("chart_div"),
        chart_data,
        {
            title: 'Captured Waveform',
            ylabel: 'Amplitude',
            xlabel: 'time (s)',
            labels: [ "Time", "Amplitude" ],

            underlayCallback: function(canvas, area, g) {
                for(var i = 0; i < observePoints.length; i++){
                    if(bitPoints[i] == 1){
                        highlightArea(canvas, area, g,
                            observePoints[i][0], observePoints[i][1],"red");
                    }
                    else{
                        highlightArea(canvas, area, g,
                            observePoints[i][0], observePoints[i][1],"yellow");
                    }
                }
            }
        });

    global_dygraph = my_dygraph
}

function addAnnotations(g){
    var annotations = []

    for(var i = 0; i < observePoints.length; i++){
        annotations[i] =
        {
            series: "Amplitude",
            x: observePoints[i][0],
            shortText: i.toString() + ":" +
            rx_bits[i].toString()+ "\n" + rx_bits_interp[i],

```

```

        width: 40,
        height:40
    }
}
g.setAnnotations(annotations);
}

function highlightArea(canvas, area, g, l, r,color){
    var bottom_left = g.toDomCoords(l, -20);
    var top_right = g.toDomCoords(r, +20);

    var left = bottom_left[0];
    var right = top_right[0];

    switch (color){
        case "red":
            canvas.fillStyle = "rgba(255, 230, 230, 1.0)";
            break;
        default:
            canvas.fillStyle = "rgba(255, 255, 102, 1.0)";
    }
    canvas.fillRect(left, area.y, right - left, area.h);
}

function recordDemo() {
    areRecording = true;
    // we flat the left and right channels down
    leftBuffer = mergeBuffers ( leftchannel, recordingLength
);
    wav_message.innerHTML = 'Recording...';
}

function stopRecording() {
    areRecording = false;
    wav_message.innerHTML = 'Done Recording...';
    //window.alert("Your recording has a buffer size of:" +
recordingLength);
}

function recordClear() {
    recordingLength = 0;
    leftBuffer = [];
    leftchannel = [];
}

function createTable(tableData) {
    var oldTable = document.getElementById('percents-table')

```

```

if(oldTable)
    oldTable.parentNode.removeChild(oldTable)
var table = document.createElement('table');
table.setAttribute("id", "percents-table")
var tableBody = document.createElement('tbody');

tableData.forEach(function(rowData) {
    var row = document.createElement('tr');

    rowData.forEach(function(cellData) {
        var cell = document.createElement('td');
        cell.appendChild(document.createTextNode(cellData));
        cell.setAttribute("style", "padding-left:22px")
        row.appendChild(cell);
    });

    tableBody.appendChild(row);
});

table.appendChild(tableBody);
document.body.appendChild(table);
}

function generatePercentTable(){
    var tableData = [[]]
    for(var i = 0; i < pow1700.length; i++){
        tableData[i] = [i, pow1700[i], pow1300[i],
rx_bits[i]]
    }
    createTable(tableData)
}

function audRecord( e, testing ) {
    var isTest = false
    getTestBits()
    if (e.classList.contains("areRecording")){
        stopRecording();
        processBits(rx_bits)
        e.classList.remove("areRecording")
        if(!isTest){
            generatePercentTable();
        }
        leftBuffer = mergeBuffers ( leftchannel,
recordingLength );
        drawChart();
        addAnnotations(global_dygraph);
    }else{
        e.classList.add("areRecording")
    }
}

```

```

        recordClear();
        recordDemo();
        resetGlobals();
    }
}

function resetGlobals(){
    shift = 0
    bridgeBitSignal = []
    rx_bits = []
    rx_bitsIndex = 0
    observePoints = [[]]
    OPindex = 0;
    globalIndex = 0;
    gotData = 0 //stays false until we decode a real data
bit
    sigProcessState = "idle"
    pow1700 = [];
    pow1300 = [];
    powArrIndex = 0;
    syncAvg1700 = 0;
    syncAvg1300 = 0;
    manSync = false;
    seenMan = 0;
    confidence = []
    cIndex = 0
    improper_bits = false
    end = 0
    start = 0;
    bitPoints = []
    rx_bits_interp = []
    receivedPayload = []
}

var end = 0;
var start = 0;
var improper_bits = false
var bitPoints = []

if (!String.prototype.splice) {
    /**
     * {JSDoc}
     *
     * The splice() method changes the content of a string
by removing a range of
     * characters and/or adding new characters.
     *
     * @this {String}

```

```

        * @param {number} start Index at which to start
        changing the string.
        * @param {number} delCount An integer indicating the
        number of old chars to remove.
        * @param {string} newSubStr The String that is
        spliced in.
        * @return {string} A new string with the spliced
        substring.
        */
        String.prototype.splice = function(start, delCount,
        newSubStr) {
            return this.slice(0, start) + newSubStr +
            this.slice(start + Math.abs(delCount));
        };
    }

function bin2hex(str){
    var remainder = str.length % 8
    var zeroFill = []
    zeroFill.fill(0,8-remainder)
    str.concat(zeroFill.toString())

    var result = []

    var j = 0
    var rIndex = 0
    for (var i = 0 ; i < str.length; i = j) {
        j += 4; // specify radix--v
        var byte = str.slice( i, j )
        result[rIndex] = parseInt(byte, 2
    ).toString(16).toUpperCase()
        rIndex++
    }

    result = result.join("")
    return result
}

function formatBits(bits){
    var bitStr = bin2hex(bits.join(""))
    //bitStr_length = bitStr.length
    var bitIndex = 0
    for(var i = 0;i < bitStr.length;i++){
        if(bitIndex % 32 == 0){
            bitStr = bitStr.splice(i,0,"<br> \n")
            i += 4
        }
        else if(bitIndex % 8 == 0){

```

```

        bitStr = bitStr.splice(i,0," ")
        i++
    }
    bitIndex++
}

return bitStr
}

// document.getElementById("receivedBits").innerHTML =
formatBits([1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,
0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,
0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,
1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,
0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0]);

//-----Decoder-----
var baudRate = 100
var samplesPerBaud = null
var shift = 0
var bridgeBitSignal = []
var rx_bits = []
var rx_bitsIndex = 0
var receivedPayload = []
var gotData = 0//stays false until we decode a real data
bit
var sigProcessState = "idle"

var globalIndex = 0;
var observePoints = [[]];
var OPindex = 0;

var lastBuffer = [];

function forceStopRec(){
    sigProcessState = "endingTransmission"
}

function processBuffer(buffer){
    var phasor1700 = detectFreq(buffer, baudRate, 1700)
    var phasor1300 = detectFreq(buffer, baudRate, 1300)

    //document.getElementById("1700presence").innerHTML =
phasor1700.mag.toFixed(6);
    //document.getElementById("1300presence").innerHTML =
phasor1300.mag.toFixed(6);
    document.getElementById("1700percent").innerHTML =
phasor1700.percent.toFixed();

```

```

    document.getElementById("1300percent").innerHTML =
phasor1300.percent.toFixed();
    document.getElementById("totalPower").innerHTML =
phasor1300.totPower;

document.getElementById("signal-Process-State").innerHTML
= sigProcessState;

switch (sigProcessState)
{
    case "idle":
        //wait for a sync signal to be present. Otherwise do
nothing
        var transmissionPresent = detectSync(buffer,
phasor1700, phasor1300)
        if (transmissionPresent)
            sigProcessState = "transmissionDetected"

        globalIndex += bufferSize;
        break;

    case "transmissionDetected":
        //"Do nothing" We are just gathing the sync signal to
get a good sync
        sigProcessState = "syncing";
        globalIndex += bufferSize;
        break;

    case "syncing" :
        //We got two buffers. Find were a chip starts with
these buffers
        var bigBuf = new Float32Array(bufferSize *2 )
        bigBuf.set(lastBuffer)
        bigBuf.set(buffer, bufferSize)
        phasor1700 = detectFreq(bigBuf, baudRate, 1700)
        phasor1300 = detectFreq(bigBuf, baudRate, 1300)
        var first_shift = findChip2(phasor1300, phasor1700)

        //Also start decoding from here on out
        shift = decodeBuffer(buffer, first_shift, false)
        sigProcessState = "recieving"
        globalIndex += bufferSize;
        break

    case "recieving" :
        shift = decodeBuffer(buffer, shift, true)
        globalIndex += bufferSize;

```

```

        var payloadPresent = false
        var ppPattern = /01(4|2)(4|2)/;
        if(rx_bits.join("").search(ppPattern) != -1)
            payloadPresent = true;
        if(rx_bits[rx_bitsIndex - 1] == 2 &&
rx_bits[rx_bitsIndex - 2] == 2 && rx_bits[rx_bitsIndex - 3]
== 2 && payloadPresent)
            sigProcessState = "endingTransmission"
            break
        case "endingTransmission" :
            console.log("Ending transsmssion")
            rxTest_index++
            //processBits(rx_bits)
            sigProcessState = "awaitNothing"
            break;
        case "awaitNothing":
            var recButton = document.getElementById("recButton")
            audRecord(recButton)
            break;

    }
    lastBuffer = [];
    lastBuffer = buffer;
}

// function testAccuracy(arr){
//     var errCt = 0;
//     for(var i = 0; i < compbitarray.length; i++){
//         if(arr[i] != compbitarray[i]){
//             errCt++
//         }
//     }
//     var errRt = errCt/compbitarray.length
//     document.getElementById("error-rate").innerHTML =
errRt;
// }

var xferLength = 64
var rx_bits_interp = []

String.prototype.regexIndexOf = function(regex, startpos)
{
    var indexOf = this.substring(startpos ||
0).search(regex);
    return (indexOf >= 0) ? (indexOf + (startpos || 0)) :
indexOf;
}

```

```

function guessBadBits(bits,powONE,powZERO){
    rx_bits_interp = rx_bits
    var bits_str = rx_bits.join("");
    start = bits_str.regexIndexOf(/(1|0)/)
    start = bits_str.regexIndexOf(/(2|4)/,start)
    //search for the first non-2
    while(bits[start] == 2 || bits[start] == 4){
        start++
    }
    start--
    end = bits_str.search("4222")
    if(end == -1){
        end = bits_str.indexOf("222",start);
    }
    if(end == -1)
        bits_str.indexOf("3222",start);
    }else{
        end = start + xferLength
    }

    if((end - start) % 8 != 0){
        console.log("Warning: Bits lost - decoding from end
of transmission")
        improper_bits = true
        start = end - xferLength
        var bits_done = 0
        for(var i = end-1; bits_done < xferLength ; i-- ){
            if(rx_bits[i] != 1 && rx_bits[i] != 0)
                rx_bits_interp[i] = (powONE[i] > powZERO[i]) ? 1:0
            bits_done++
        }
    }
    else{
        for(var i = start; i < end; i++){
            var nextIsSync = (rx_bits[i+1] == 2)
            if(rx_bits[i] != 1 && rx_bits[i] != 0 && nextIsSync){
                rx_bits_interp[i] = (powONE[i] > powZERO[i]) ? 1:0
            }
        }
    }
    return rx_bits_interp;
}

function extractPayload(bits_arr){
    //Strip up to the second set of 2's
    //var start = 0

```

```

new_bits = bits_arr.slice(0,end)
// if(improper_bits){
//   start = end - xferLength
// }
// else{
//   start = new_bits.indexOf(1)
//   new_bits = new_bits.slice(start)
//   start = new_bits.indexOf(2)
//   new_bits = new_bits.slice(start)
//   //search for the first non-2
//   start = 0
//   while(new_bits[start] == 2){
//     start++
//   }
// }
for(var i = start; i < end;i++){
  bitPoints[i] = 1;
}

new_bits = new_bits.slice(start)
// var bits_str = new_bits.join("");

// var end = bits_str.search("42222")
// if(end == -1){
//   end = bits_str.search("2222")
// }

return new_bits
}

//processBits([0,1,1,0,1,0,0,0,0,1,1,0,0,1,0,1,0,1,1,0,1,1
,0,0,0,1,1,0,1,1,0,0,0,1,1,0,1,1,1,1])

function printRecievedBits(bits){
  var bitStr = formatBits(bits)
  document.getElementById("receivedBits").innerHTML =
bitStr;
  document.getElementById("numRecieved").innerHTML =
bits.length;

  var result = ""
  var j = 0;
  for (var i = 0 ; i < bits.length; i = j) {
    j += 8; // specify radix--v
    var byte = bits.slice( i, j )
    var cCode =parseInt ( byte.join(""), 2 )
    result += String.fromCharCode( cCode );
  }
}

```

```

    document.getElementById("received-string").innerHTML =
result
}

function processBits(bits_arr){
    //Strip the 2's from the bit array
    //We are assuming that all 2's (End/Start bits) are only
at the end
    var new_bits = []
    var j = 0
    var i = 0
    var u8ArrIndex = 0;
    var uint8 = new Uint8Array();

    new_bits =
guessBadBits(bits_arr.concat([2,2,2,2]),pow1700,pow1300);
    new_bits = extractPayload(new_bits);

    //All data is transferred in bytes. If the number of
bits are not divisible by 8 then there must be data
missing
    if((new_bits.length % 8) != 0){
        console.log("Warning: Not all bits recieved!!")
        //extractPayload(bits_arr);
        //Make the rest of the bits 0
        var remainder = new_bits.length % 8
        var zeroFill = []
        zeroFill.fill(0,8 - remainder)
        new_bits = new_bits.concat(zeroFill)
        //return ""
    }
    printRecievedBits(new_bits)
    receivedPayload = new_bits
    testAccuracy(new_bits)
}

function generateBits(length){
    //172 characters
    //32 characters
    rbits = []
    for(var i = 0; i < length; i++){
        var rand = Math.random()
        bit = rand > 0.5 ? 1:0
        rbits.push(bit)
    }
    return rbits
}

```

```

function setTestBits(bits){
    requestStr = "sentBits=" + JSON.stringify(bits)
    $.ajax({
        type: "POST",
        url: "transmissionTest.php",
        data: requestStr,
        cache: false,
        success: function(result){
            document.getElementById("tx-test-status").innerHTML =
result
        }
    });
}

function getTestBits(){
    requestStr = "getBits=1"
    $.ajax({
        type: "GET",
        url: "transmissionTest.php",
        data: requestStr,
        cache: false,
        success: function(result){
            //document.getElementById("rx-test-status").innerHTML
= result
            all_bits = JSON.parse(result);
            testRxSystem2()
        }
    });
}

var totCorrectBits = 0;
var totalBits = 0
var totCorrectPackages = 0;
var interpPayloadArr = []
var correctnessArr = []
var rxTestIndex = 0

function playTestBits(){
    getTestBits();
    playBits(all_bits);
}

function testAccuracy(myBits){
    var correctBits = 0

```

```

tBits = all_bits //[rxTestIndex]
totalBits += tBits.length;
for(var i = 0; i < tBits.length; i++){
    if(tBits[i] == myBits[i]){
        totCorrectBits++
        correctBits++
    }
}

if(correctBits == tBits.length)
    totCorrectPackages++

interpPayloadArr.push(myBits)
correctnessArr.push(correctBits/totalBits)
generateTestTable(rxTestIndex)
rxTestIndex++
}

function generateTestTable(tIndex){
    var tableData = [[]]
    for(var i = 0; i < tIndex+1; i++){
        var tBts = formatBits(all_bits)
        var rBts = formatBits(interpPayloadArr[i])
        var crct = correctnessArr[i]
        tableData[i] = [i, tBts, rBts, crct ]
    }
    createTable(tableData)
}

var longPLs = [[]]
var shortPLs = [[]]

function testTxSystem(){
    return
    //generate 100 172*8 bit payloads
    //generate 100 32*8 bit payloads
    longPLs = [[]]
    shortPLs = [[]]

    for(var i = 0; i < 100;i++){
        longPLs[i] = generateBits(172*8)
        shortPLs[i] = generateBits(32*8)
    }

    //setTestBits(longPLs.concat(shortPLs))
    ready = true
}

```

```

var ready = false

setInterval(testInit,1000)

function testInit(){
    return;
    if(ready && go){
        setInterval(testTxSystem2(),50)
        setInterval(testRxSystem2(),50)
    }
}

var txTest_index = 0

function testTxSystem2(){
    if(playing){return}
    for(var i = txTest_index; i < 100; i++){
        if(playing){return}
        playBits(longPLs[i])
        txTest_index++
        //setTimeout(function() {}, 200)
    }

    for(var i = 0; i < 200; i++){
        if(playing){return}
        playBits(shortPLs[i])
        txTest_index++
        //setTimeout(function() {}, 200)
    }
}

var go = false

document.addEventListener("keypress",continueTest,false)
function continueTest(e){
    return
    var valid = false
    if(!e){
        valid = true
    }
    if(e && e.keyCode == 32)
        valid = true

    if(valid)
        playing = false

    go = true
}

```

```

var all_bits = [[]]

function testRxSystem() {
    return
    getTestBits();
    ready = true
}

var rxTest_index = 0
var oldRxTI = 0

function testRxSystem2(){
    return
    var recButton = document.getElementById("recButton")
    if(rxTest_index != 0 || oldRxTI == rxTest_index
-1){return}
    oldRxTI = rxTest_index
    for(var i = rxTest_index;i < 2;i++){
        xferLength = all_bits[i].length
        audRecord(recButton,true)
        testAccuracy(receivedPayload,i)
    }
}

function testSystem(){
    return
    var role = $('input[name=test-role]:checked').val();

    // if(role == "tx"){
    //     testTxSystem2()
    // }else{
    //     testRxSystem()
    // }

    go = true
}

var syncAvg1700 = 0;
var syncAvg1300 = 0;
var manS

```