University of Cape Town
Department of Electrical Engineering

EEE4120F
Sea Sounds Data Caster Project
Final Report (Milestone 4)

Stefan Schröder - SCHSTE054
Elle Mouton - MTNELL004

15 May 2019

Abstract

This project is aimed at investigating the calming sounds of the ocean and understanding how to recreate these sounds. The aim is then to design and build a transmitter that will produce and play relaxing sea sounds that include a hidden message. A receiver will also be implemented and it will take the input produces by the transmitter and decode the original message. This report will detail the implementation of both the encoding and modulating techniques used for the transmitter as well as the demodulating and decoding techniques used for the receiver. Investigations will also be made regarding the necessary steps that would need to be taken in order to write the code in HDL and implement the system on an FPGA.

CONTENTS

I. Introduction

A. Aim of the project

The aim of the Sea Sounds Data Caster project is to encode messages within audio that sounds like relaxing ocean sounds. This will be done by first investigating the sources and structure of the sounds of the ocean so that the sound can be replicated. Steps will then be taken to design and implement both a transmitter and a receiver for this system. The transmitter will take a text message as an input and will hide this data in wave sounds. The receiver will take the wave sounds as an input and will reconstruct the original message. The transmitter and receiver will both be implemented in MATLAB as a software solution but the report will outline the necessary steps of how these systems could be implemented on an FPGA.

B. Research from Milestone 1 regarding sea sounds

Milestone 1 was aimed at investigating the causes and structure of ocean sounds as well as the reason why these sounds are relaxing to humans. After doing extensive research and mathematical analysis it was found that ocean sounds are very similar to white noise which means that in there is a gaussian distribution in the spread of energy in the frequencies of the sounds. It was also found that the range of audible frequencies in the ocean sounds are mainly between 50Hz and 20kHz. It was also found that the reason that humans find the ocean sounds so relaxing is that the sound is slow changing and predictable. There are slow build-ups and low build-downs and no sudden changes in volume.

C. Outline of this report

This report will first discuss the development of the receiver sub-system. The general structure and workings of the receiver algorithm will be presented and then further detail will be discussed regarding how the receiver could be implemented on an FPGA. Following the receiver, the transmitter design will be presented. The MATLAB solution will first be discussed and then a further investigation of how the sub-system would be implemented on an FPGA will be discussed.
The report will then discuss the general implementation of the entire system and will provide important code snippets that were used for both the transmitter and receiver.
Lastly, the report will discuss the results that the final implementations produced and then reflections will be made on the system development and any possible improvements for further development will be identified.

D. A summary of the major design decisions that were made

One of the major decisions that was made was the modulation scheme that would be used. We decided to combine frequency modulation as well as amplitude modulation, as it allowed us to have a higher baud rate while still sounding like the ocean. We achieved this by encoding each bit at a certain frequency band, and having the amplitude of the frequency band depict the information that we wanted to send.

E. Reflections on the development process

Initially we wanted to use more frequency bands so that we could send a larger number of bits at once. However, this would mean that we would need to use higher frequency bands and vary the amplitude of these bands. This caused the sound of the audio to stray too far from that of ocean sounds and so this option of using higher frequency bands was abandoned and instead four frequency bands on the lower side of the spectrum were chosen. The lower frequencies also produced more predictable and rhythmic sea sounds. Thus we sacrificed a high baud rate for better quality sound.

II. Receiver Concept Design

A. Block Diagram

The diagram shown in figure 2 below illustrates the concept of the receiver. The block diagram can be read from right to left with the input on the left being the audio file and the output on the right being the decoded text message. The pictures under the blocks illustrate the process visually.

B. Schematic Design

1) Description of necessary components: In table I we list the components that are needed within the system, and the reasons for them being included in the system.

2) Block diagram of final Receiver: The reciever's final block diagram is more clearly defined in figure 1.

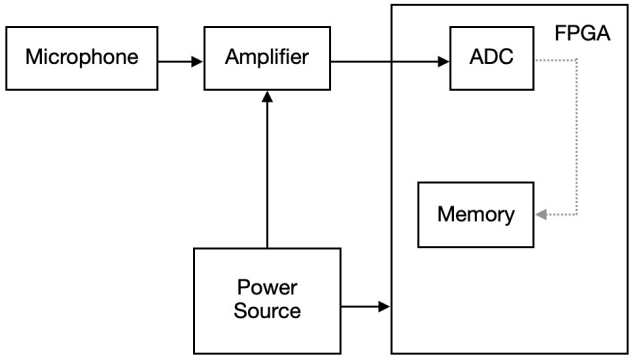

Fig. 1: Final Receiver block diagram

3) Component list: The components that were selected are shown in table II

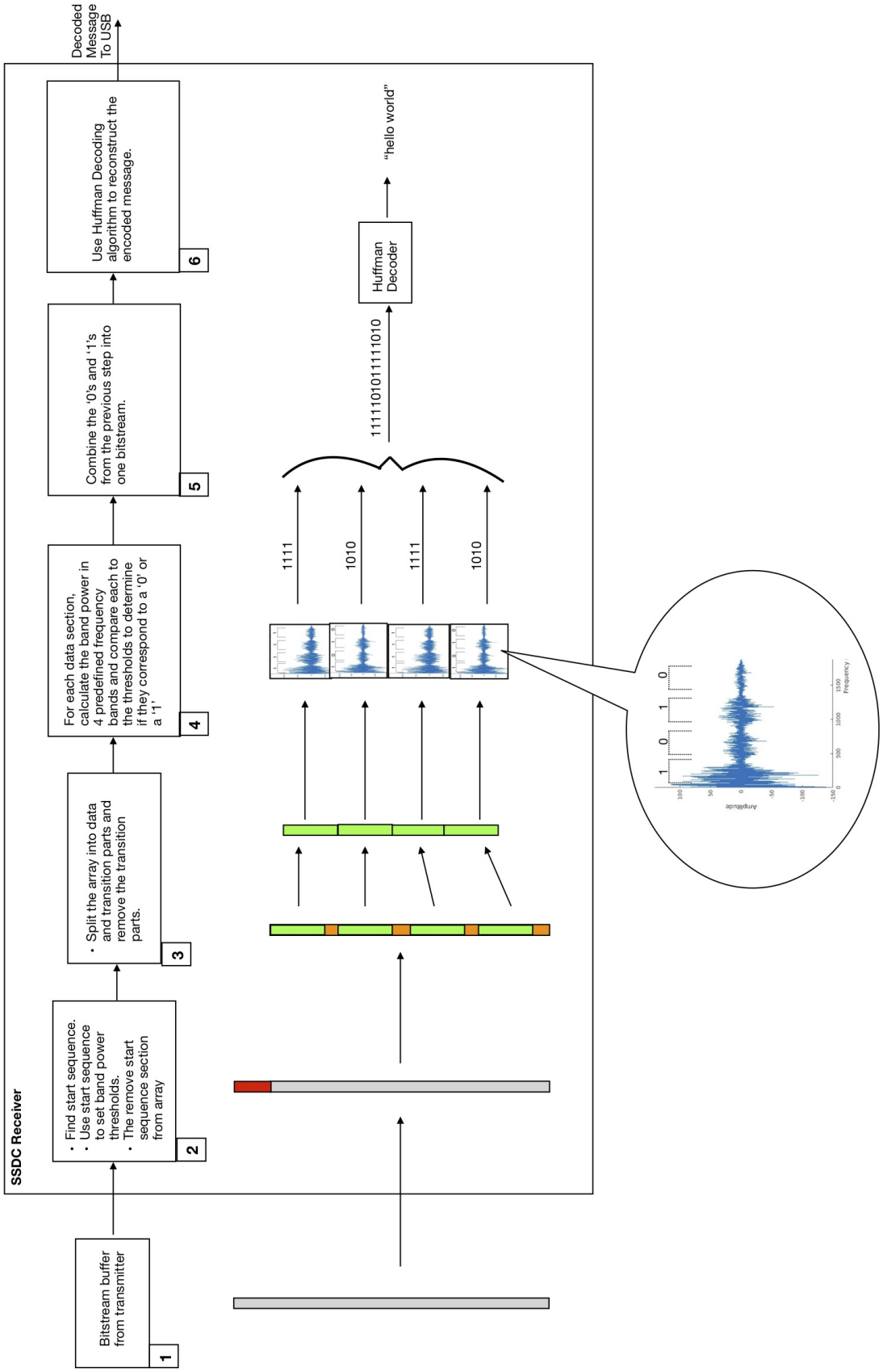

Fig. 2: Block Diagram of the Receiver implementation

TABLE I

DESCRIPTION OF COMPONENTS

Component	Description of components
Microphone:	The microphone is required to pick up the ocean sounds from the transmitter. The microphone should ideally have a far and narrow pick-up range so as to exclude any unwanted noise. For this specification a shotgun condenser microphone is ideal.
Amplifier:	The system requires an amplifier in order to increase the power of the input signal to fully utilize the ADCs bit length and quantization. The amplifier should ideally minimize the amount of noise present in the system.
ADC:	An ADC will be used to convert the analog signal from the amplifier to a digital signal that the FPGA will be able to read and store in memory. An ADC with a high resolution will result in fewer quantisation errors. In the transmitter, we only have 8 different levels the bits can be at, so we don't need a ADC with a resolution higher than 8 bits, because an 8 bit ADC will provide 256 levels of resolution.
Memory:	Since the transmitter will be sending audio signals of variable length, a large amount of memory is required for the receiver system. As we are most likely using an 8-bit ADC and a recording sampling rate of 44.1kHz, we will receive 44k Bits per second, which translates to 2.65 MBs per minute. Thus we will not need too much storage to satisfy the requirements.
Power Source:	A power source is required to power the FPGA and Amplifier. As the FPGA only needs 3.3V @ 1.0-1.5A to power on, it can be powered on with most cell phone battery banks. Which is useful because they are cheap enough, and easy to come by. The amplifier has a input voltage regulation of between 2.7v to 5.5v so could also be powered with the same battery bank as it provides 5V.
FPGA:	The FPGA is going to be the heart and sole of the receiver, as it is the device that will do the all of the processing. Since we have not used many FPGAs it is prefered to use the Nexys 4 ddr. The FPGA does not need to be particularly fast, as we are only sending 4 bits every 2 seconds.

TABLE II

RECEIVER COMPONENT CHOICE

Component Type	Specific Component	Reason for choice
Microphone	Electret mic	Even though a shotgun microphone would be preferable, they are usually very expensive and range from prices of R1500 - R4000, and thus is not necessary for this project. A simple circuit microphone can be used to achieve a satisfactory result. A microphone such as the electret mic can be used and connected directly to the Amplifier circuit.
Amplifier	MAX9714	For the amplifier, it is easy to make a simple circuit that can connect to the ADC directly. The amplifier circuit till make use of the MAX9814, which is a nice IC that has variable gain control and a low amount of noise.
ADC	Nexys 4 onboard	The FPGA that we are using (Nexys 4) contains a built in FPGA that can perform the function that we need, this is another reason why we chose to use the Nexys 4.
Memory	Nexys 4 onboard	The Nexys 4 FPGA has 2 build in memory chips, 128Mbit Cellular RAM and a 128Mbit non-volatile serial Flash device, which is perfectly sufficient for the purpose of this project.
Power Source	Macally 13000mah	For the power supply we could just use a power bandk that provides enough output power and two output USBs. The Macally 13000mah Power Bank with 2 USB Ports does this, and has 1 USB port that can output 2.1A and another that can output 1A. The 2.1A will be used to power the amplifier, and the 1A used to power the FPGA.
FPGA	Nexys 4	The FPGA that we are using is the Nexys 4, the reason for choosing this is because we are already acquainted with this device, and it has a built in ADC and enough memory to power the whole system.

III. Receiver Design

The code for the MATLAB receiver program is presented in the code listing below. The process for the receiver is also described and illustrated in figure 2.

One section of the code that would need to be adapted for an FPGA implementation is a synchronization step. This step would involve waiting for the transmitter to send the start sequence. The receiver would record the audio and would note the initial band-power levels in each of the pre-defined band and then would wait for all the band-power levels to drop to half of the original values and then would start recording. The original band-power levels would also be used to set the thresholds for the receiver program. The synchronization process is further described in the Transmitter Design section.

```
function receive()
    filename = 'encoded_data.wav';      % file to read audio from
    bitRate = 2;                                % four Bits sent every x seconds
    transRate = 1;                              % transition periods in seconds
    thresh1 = 8; thresh2 = 1.6; thresh3 = 0.6; thresh4 = 0.1; %bandpower thresholds are set
    % ================================================================
    % define the frequencies that the bands will be centered around
    % ================================================================

    p1 = 250; p2 = 2250; p3 = 4250; p4 = 6250;

    % ================================================================
    % load the sound file
    % ================================================================

    [y,sampleRate]=audioread(filename);  %load audio into array y
    len_bit_piece = sampleRate*bitRate; %length of time that each group of 4 bits will be sent for
    len_transition = sampleRate*transRate;   %length of time that each transition period will be
    num_chunks = ceil(length(y)/(len_bit_piece+len_transition)); % number of 4 bit groups that are being received

    % ================================================================
    % create a new array, y_cut, that will not include the transition periods.
    % ================================================================

    y_cut = zeros(num_chunks*len_bit_piece,1);

    for i = 1:num_chunks
        if(i<num_chunks)
            new_index = ((i-1)*len_bit_piece)+1;
            old_index = ((i-1)*(len_bit_piece+len_transition))+1;
        else
                        new_index = ((i-1)*len_bit_piece);
            old_index = ((i-1)*(len_bit_piece+len_transition));
        end

       y_cut(new_index:new_index+len_bit_piece-1) =  y(old_index: old_index+len_bit_piece-1);
    end
    % ================================================================
    % Reshape the y_cut array so that it is a 2D array where every column
    % represents a new data segment (where a new set of 4 bits is encoded)
    % ================================================================

    y_split = reshape(y_cut, len_bit_piece, num_chunks);

    % ================================================================================
    % loop over each chunk, calculate the bandpower in all the predefined
    % frequency bands and classify each band based on the thresholds.
    % The "bandpower" function takes a section of the audio, the sampling rate
    % and the range of frequencies to calculate bandpower over for that audio sample
    % ================================================================================

    final = "";

    for i = 1:num_chunks
       sample =  y_split(:,i);
       bp1 = bandpower(sample, sampleRate, [p1-400 p1+400])*10000;
       bp2 = bandpower(sample, sampleRate, [p2-400 p2+400])*10000;
       bp3 = bandpower(sample, sampleRate, [p3-400 p3+400])*10000;
       bp4 = bandpower(sample, sampleRate, [p4-400 p4+400])*10000;

       if(bp1<=thresh1)
           final = final + "0";
       else
           final = final + "1";
       end
       if(bp2<=thresh2)
           final = final + "0";
       else
           final = final + "1";
       end
       if(bp3<=thresh3)
           final = final + "0";
       else
           final = final + "1";
       end
       if(bp4<=thresh4)
           final = final + "0";
       else
           final = final + "1";
       end
    end
    % ================================================================
    % pass the final bitstream to the huffman decoding algorithm to get the decoded message back
    % ================================================================

    message = huffmanDecode(char(final));
    fprintf(final+"\n");
    fprintf("Decoded Message: "+message+"\n");
end
```

IV. Transmitter Design

A. Encoding Scheme

In order to include the message into the wave sounds it must first be converted into binary format. We have decided to use the Huffman Tree Encoding technique in order to achieve this. A Huffman tree is made by creating a binary tree of the letters of the alphabet as well as any other necessary characters and assigning each letters unique string of bits. The more frequent the letter appears in words, the higher up it will be in the binary tree and the shorter its uniquely assigned bit string will be. Figure 3 below shows the relative letter frequency and Figure 4 shows an example of what a Huffman tree looks like.

By using the Huffman encoding technique, the number of bits per letter that needs to be encoded and transmitted becomes drastically less. For example, if the message being sent is testing then then the Huffman Encoding of this message using the letter frequencies shown in Figure x is the following string of 36 bits: ”010110110100000010110010000010001011”. If ASCII encoding was used then for each character in the string being sent, 8 bits would be required and so 56 bits would have been required to send the testing message.
In the FPGA implementation we would need to send start bits so that the receiver can determine power in the bands before the data is sent and so that the receiver and and transmitter can synchronize. These start bits can be added in the Huffman encoding so that the transmitter and encoding section does not have to be made aware of these starting bits and thus do not need to be changed. As discussed above the start bits will first set all the bands to high, and then low. So at the start of the Huffman bit stream, the program can just add ”11110000” and then have the transmitter behave as normal. These start bits will allow the receive to synchronize its timing and then the power thresholds of each 1 and 0.

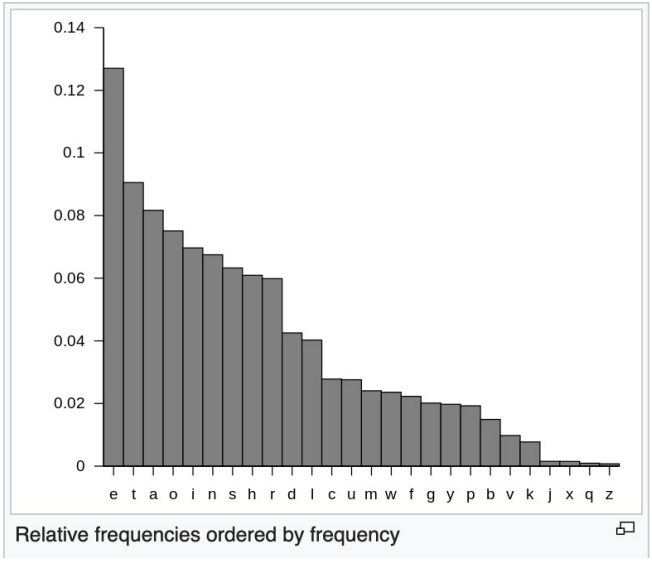

Fig. 3: Relative frequencies of different letters [1]

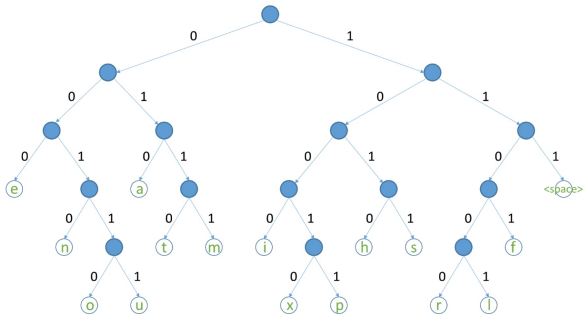

Fig. 4: An example of a Huffman binary tree [2]

B. Modulation Scheme

To encode each section of 4 bits, we first convert the Huffman encoding output into 4 waves, each at their own frequency. Each of these waves have half the amplitude of the previous (lowest frequency being the highest), this is done to emulate the sound of the ocean better. These waves will be encoded as; full amplitude being considered a 1 and 1/3 amplitude being considered a 0. Each wave will encode the Huffman output by splitting it into groups of 4, and then giving each wave frequency one of the bits. After these waves each have their amplitudes encoded, they are added together and made into array. After this the wave frequencies are convolved with noise bubbles, which is a band of noise. This is so that each encoded peak sounds similar to the ocean. Since we are sampling at 16 kHz we need to ensure that the data is only created with in the frequency band of of 16kHz (according to Nyquist). So all of the waves and their noise bubbles need to be less than 8kHz.

Figure 5 below shows the process of creating the audio section that would encode the bitstream 1010.

C. Description and block diagram of a possible FPGA implementation

The block diagram that describes the system can be seen in figure 6. There is quite a lot going on here and color codes were added to make things slightly easier. The system starts with the input of a Huffman bit stream, being stored into one long fixed register. The system then goes on to split this long register into groups of 4 bits and sends them to the next blocks as they are required.
The first group (red group) is the section where each bit is encoded. A group of 4 bits is set to a register and each bit connects to a Wave bit select block. This block is responsible for the encoding each bit as a 1 or a 0, where a 1 is represented by a full amplitude sine wave, and a zeros is represented by a half amplitude sine wave. Each sine wave register goes into one of these blocks, so that each wave is encoding one of the 4 bits. The output of the wave bit select is added together and

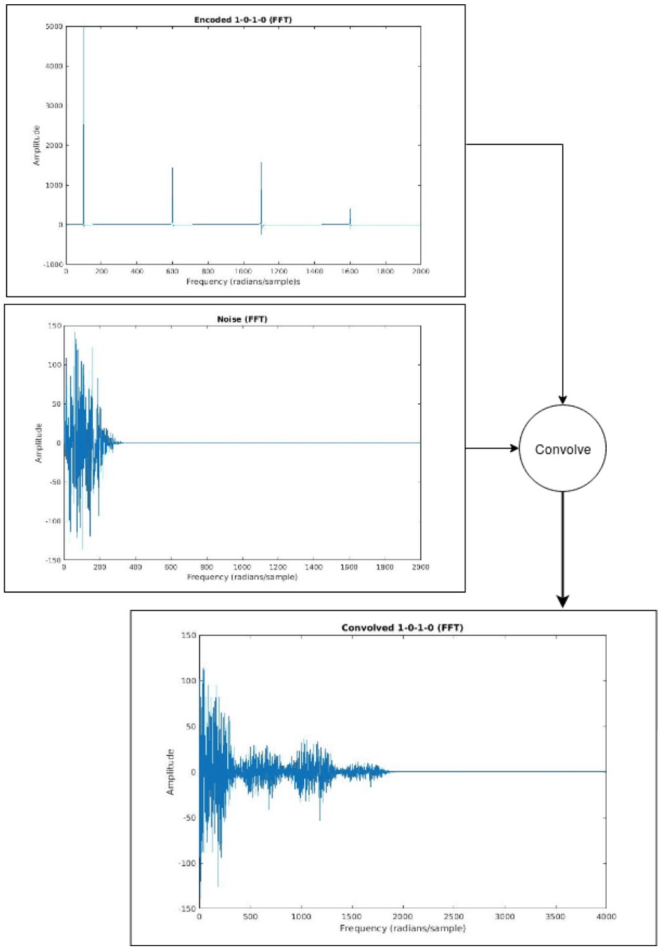

Fig. 5: Diagram showing the modulation process

then stored in the first part of a final register.

The next group (green group) is the section where the transitions between bits are created. This groups purpose is to ensure that when bits change there is a smooth transition between a 1 and a 0. Inside the Transition multiplier there are 4 options that can occur: the first option is when the previous bit is a 0 and the next bit is a 0, this case will make a transition wave with flat amplitude of 1/2 of the original wave. The second option is when the first bit is a 0 and the next is a 1, in this case the wave will go through a transition where its amplitude will start from a 0 level (1/2 amplitude represents a 0) and go to a 1 level. The third option is just the opposite, where the wave starts at 1 and goes down to 1/2 amplitude. The third option is when the previous bit is 1 and the next bit is 1, this option is simple as the transition wave is just the normal input wave. Each bit will need to have a transition and all of their outputs are added together and stored in the second part of the output register.

Once the encoded wave and the transition wave has been stored in the final wave the system is now able to proceed to the final stage. In the final stage the whole wave is multiplied by noise (which is convolution in frequency), this multiplication of noise creates the ”bubbles” of noise in the frequency domain, that simulate the sound of the ocean. Before the noise is multiplied to the output, it needs to be band limited. This is done by convolving the generated noise with the impulse response of a FIR low pass filter. The system explanation is now complete, and shows how the sound is generated.

V. Implementation

A. Transmitter: Encoding

To begin the encoding process a few constants constants had to be predefined. These redefinition’s are explained by their comments and are used throughout the rest of the code.

```
ampDiff = 3/5;          % Difference between size of first and second peak
maxAmp = 1/(1+ampDiff+ampDiff^2+ampDiff^3);% Total amplitude made by output x/(dont touch)
sampleRate = 48000;     % sample rate
bitRate = 2;            % Bits sent every x seconds
heightMult = 1/3;       % The multiplier creating the difference between 1 and 0
peakOne = 250;          % frequency of first  peak
peakTwo = 2250;         % frequency of second peak
peakThree = 4250;       % frequency of third peak
peakFour = 6250;        % frequency of the fourth peak
oscillatorFreq = 0.25;  % Frequency of the oscillator
trans = 1;              % length of the transition byte (seconds)
```

To ensure that the compilation of sound could occure quickly, the first step was to create the arrays of predefined sine waves, and can be seen in figure 6, with each of the Sine wave blocks. The code for each of these sine wave is simple enough and shown in the next listing.

```
rang = transpose(linspace(1,bitRate,sampleRate*bitRate));
waveOne     = sin(2* pi * peakOne   * rang);
waveTwo     = ampDiff*sin(2* pi * peakTwo * rang);
waveThree   = ampDiff*ampDiff*sin(2* pi * peakThree * rang);
waveFour    = ampDiff*ampDiff*ampDiff*sin(2* pi * peakFour  * rang);
```

It can be seen from this listing that each successive wave is half (ampDiff) the size of the previous, and that each wave is 2 seconds (bitRate) long. The next step that was taken was to predifine waves combinations from these waves, and create variable for each of the possible combinations of 4 bit values. This was not done in the block diagram of figure 6, however it was done in matlab to produce quicker results. An example of a few of these predifined 4bit was is shown in the next listing.

```
zzzz = (maxAmp*heightMult)*waveOne +
       (maxAmp*heightMult)*waveTwo +
       (maxAmp*heightMult)*waveThree +
       (maxAmp*heightMult)*waveFour;
...
zozo = (maxAmp*heightMult)*waveOne +
       (maxAmp           )*waveTwo +
       (maxAmp*heightMult)*waveThree +
       (maxAmp           )*waveFour;
...
oooo = (maxAmp           )*waveOne +
       (maxAmp           )*waveTwo +
       (maxAmp           )*waveThree +
       (maxAmp           )*waveFour;
```

The options that are shown in the listing corresponse to the four bits being z=0 o=1, so the options 0000 and 0101 and 1111 are displayed. It can be seen that the heightMult is used to reduce the amplitude of a sine wave to create a logical 0, and no multiple is added when creating a logical 1.

The next phase that was implemented was to create the predefined transition waves, only the transition from a logical 0 to a logical 1 was predefined, as a downward transition can

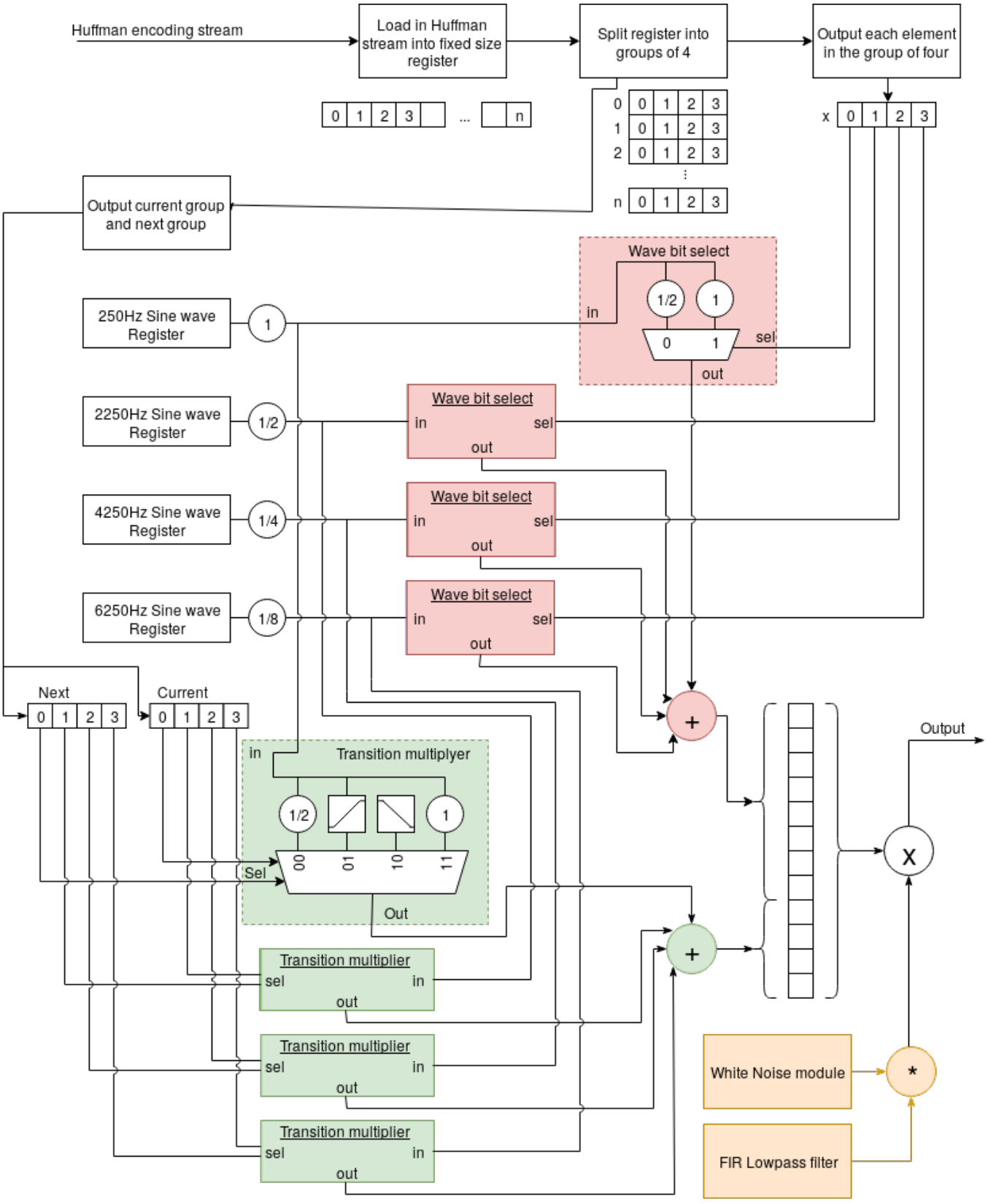

Fig. 6: Block diagram showing the implementation of the transmitter section of the system

be created by simply revering the upward transition array.

```
transLine = linspace(heightMult, 1, trans*sampleRate);
for i=1:trans*sampleRate
    peakOneTrans(i)     = maxAmp*transLine(i)*waveOne(i);
    peakTwoTrans(i)     = maxAmp*transLine(i)*waveTwo(i);
    peakThreeTrans(i)   = maxAmp*transLine(i)*waveThree(i);
    peakFourTrans(i)    = maxAmp*transLine(i)*waveFour(i);
end
```

To create final unmodulated output the next step was simple, all that needed to be done was break up the huffman bit string into groups of 4, and provide a switch case for each situation. This is not the most code efficient way to do this, but it is the best way to produce results the quickest, instead of having to mathematically define the arrays each time.
After a group of 4 bits is process, the transition waves need to be added, a code snippet showing the full creation is listing 1. OutputArray contains the final encoded waves, but they still need to be convolved with noise, so that they sound like the ocean.

B. Transmitter: Modulation

The modulation of the transmission signal is quite simple as all that needs to be done is to multiply the outputArray with band limited noise. The band limited noise is created with the following MATLAB function.

```
function y=noise_bubble(len)
    mu=0;
    sigma=1;
    x=sigma*randn(len,1)+mu; %creates full spectrum white noise

    load("Filters/lowpass48k900.mat");
    y=conv(x,impresp); % convoles the noise with the filter (ie filters the noise)
    y = y(1:len);

end
```

The noise that is being create is full spectrum and wideband, as that is what sea sound was discovered to present. This noise is then convolved with a lowpass filter's impulse responses (load filter creates the variable impresp with all of the impulse responses of the filter). Since the convolution process creates more samples than the original signal had, the output has to be truncated to ensure that it is the same length as the outputSignal.

The final step is to multiply these noise bubbles to our OutputArray afterwhich the sound and encoding is complete, and ready to be played for the listener.

```
noise = noise_bubble(finalLength);

finalOut = outputArray .* noise; % convole noise bubbles in frequency domain.
```

However a final touch was added to give the sound the little extra effect that made it sound more like the ocean. A final wave of very low frequency was multiplied over the whole wave, to create the sound of the ocean pulling in and out. This multiplication of a final wave does not have any effect in the frequency domain as it is such low frequency that it is ignored in the decoding process.

```
len = linspace(1,length(stream)*bitRate/2,finalLength);
oscillator = (0.25*cos(2*pi*oscillatorFreq*len)+0.75)';
finalOut = finalOut .* oscillator
```

Listing 1: MATLAB code to show the creation of the final unmodulated wave

```
outputArray = [];
for i=1:4:length(stream)
    % get first 4 bits
    string = strcat(stream(i),stream(i+1),stream(i+2),stream(i+3));

    switch(string)
    case "0000"
        outputArray = [outputArray; zzzz];
    case "0001"
        outputArray = [outputArray; zzzo];
    ...
    case "0110"
        outputArray = [outputArray; zooz];
    case "0111"
        outputArray = [outputArray; zooo];
    ...
    case "1111"
        outputArray = [outputArray; oooo];
    end

    % transition waves are added
    if(i<length(stream)-4) % check to see if there are the next four bits
        transArray = zeros(sampleRate*trans,1); % preallocate array space
        nextString = strcat(stream(i+4),stream(i+5),stream(i+6),stream(i+7)); % next four bits
        this = char(string);
        next = char(nextString);

        % transition for wave at 250 Hz
        if(this(1)<next(1)) % transition from logical 0 to logical 1
            transArray = peakOneTrans;
        elseif(this(1)==next(1)) % no transition
            if(this(1)=='0') % both low
                transArray = heightMult*maxAmp*waveOne(1:sampleRate*trans);
            else % both high
                transArray = maxAmp*waveOne(1:sampleRate*trans);
            end
        else % transition from logical 1 to logical 0
            transArray = flip(peakOneTrans);
        end
        ....
        % transition for wave at 6250Hz
        if(this(4)<next(4))
            transArray = transArray + peakFourTrans;
        elseif(this(4)==next(4))
            if(this(4)=='0')
                transArray = transArray + heightMult*maxAmp*waveFour(1:sampleRate*trans);
            else
                transArray = transArray + maxAmp*waveFour(1:sampleRate*trans);
            end
        else
            transArray = transArray + flip(peakFourTrans);
        end

        % add transition array
        outputArray = [outputArray; transArray];
    end
end
```

C. Receiver: Demodulation

The entire receiver program is shown in section 3 (Receiver Design). In this section the aim is thus to extract and explain the main sections of code shown in section 3. This section will also make reference to the block diagram shown in section 2.

The code below shows how the bandpower thresholds are set. Ideally these thresholds would not be hardcoded but rather would be determined from the start sequence of the audio file which corresponds to step 2 in figure 2.

```
%setting of the bandpower thresholds
thresh1 = 8; thresh2 = 1.6; thresh3 = 0.6; thresh4 = 0.1;
```

The code for loading the audio file and determining how much data to expect from it is shown in the code snippet below. The code shows first how the audio file is read into an array called y and the sample rate of the audio is also loaded. Then the number of array elements that correspond to a piece of data (len_bit_piece) is calculated in line 2 and in line 3 the number of array elements that corresponds to a transition section is calculated (len_transition). Based on these calculated lengths, the total number of data segments that the audio will carry can be calculated which is what is done in line 4. The variable num_chunks holds the number of 4-bit segments that the audio contains.

```
 %load audio into array y
[y,sampleRate]=audioread(filename);
%length of time that each group of 4 bits will be sent for
len_bit_piece = sampleRate*bitRate;
%length of time that each transition period will be
len_transition = sampleRate*transRate;
% number of 4 bit groups that are being received
num_chunks = ceil(length(y)/(len_bit_piece+len_transition));
```

The code below shows how a new array y_cut is created from the original array by removing the transition periods. This reconstruction of the data array corresponds to step 3 in figure 2.

```
 y_cut = zeros(num_chunks*len_bit_piece,1);

   for i = 1:num_chunks
       if(i<num_chunks)
           new_index = ((i-1)*len_bit_piece)+1;
           old_index = ((i-1)*(len_bit_piece+len_transition))+1;
       else
           new_index = ((i-1)*len_bit_piece);
           old_index = ((i-1)*(len_bit_piece+len_transition));
       end

      y_cut(new_index:new_index+len_bit_piece-1) =  y(old_index: old_index+len_bit_piece-1);
   end
```

The receiver code then goes on to loop over each data segment in the data array and for each segment, the bandpower for certain predefined frequency bands is calculated as seen in the code below.

```
bp1 = bandpower(sample, sampleRate, [p1-400 p1+400])*10000;
bp2 = bandpower(sample, sampleRate, [p2-400 p2+400])*10000;
bp3 = bandpower(sample, sampleRate, [p3-400 p3+400])*10000;
bp4 = bandpower(sample, sampleRate, [p4-400 p4+400])*10000;
```

The bandpower values are then compared to the set thresholds and then based on whether or not the bandpower values are above or below the threshold for that frequency, a 0 or 1 is added to the final bitstream. This corresponds with step 4 and 5 in Figure 2.

D. Receiver: Decoding

The same Huffman Tree used to encode the data in the transmitter is used to decode the data in the receiver. The code for the Huffman Tree decoding function is shown below. This corresponds to step 6 in Figure 2.

```
function y = huffmanDecode(x)
y = "";
keys = {'e','t','a','o','i','n','s','h','r','d','l','c','u','m','w','f','g','y','p'
        ,'b','v','k','j','x','q','z',' '};
values = {'01101','01011','01001','00110','00100','00010','00000','011111','011110',
          '011101','011100','011000','011001','010001','010101','010100','001011',
          '000111','001010','000011','000010','0100001','0001100','0001101',
          '01000001','01000000', '00111'};

dict2 = containers.Map(values, keys);
temp = "";

for i = 1: length(x)
    temp = temp+x(i);
    if(isKey(dict2, char(temp)))
        y = y+dict2(char(temp));
        temp = "";
    end
end
end
```

VI. Results

After developing both the transmitter and receiver in MATLAB, the system as a whole was tested. This was done by using the transmitter program to create audio containing various messages of different sizes. The audio would be saved and listened to to ensure that the sounds produces were similar to that of relaxing sea sounds. Then the receiver program would be run with the audio file as an input. The accuracy of the system would be a measure of how similar the decoded message was to the original message. Because of the extra layer of noise that we add to the audio in the transmitter program, the band-power levels in each of the predefined frequency bands was non-deterministic and so some adjustments of the band-power thresholds were required so that the '0' and '1's would be accurately decoded. After this was done, the system worked very well and all of the messages that were used to test the system were transmitter and decoded correctly.

Figure 7 shows the FFT of a recording of actual sea waves and figure 8 shows the FFT of the produced audio for the same frequency range. As can be seen in these figures, the profile of the FFTs are very similar. Both are similar to a Gaussian distribution with the amplitude of the frequency components decreasing for higher frequencies.

One thing to note is that if this system were to be developed in hardware, with the audio being played through a speaker and a microphone recording the sound, then much more noise would be introduced and this would affect the band-power thresholds being used and so could result in errors in the decoded message. In order to accommodate for this, it would be necessary to implement a starting sequence that the receiver would know when to start recording the audio and also so that the band-power thresholds could be set dynamically based on the band-power valued recorded in the start sequence.

In an FPGA implementation it would also be necessary to test the accuracy of the system for varying distances between the transmitter and reciever.

VII. Discussion and Conclusions

The system was used to send information and it came with some significant drawbacks. Because of the requirement to have the data transmission occur in the audible range it allows

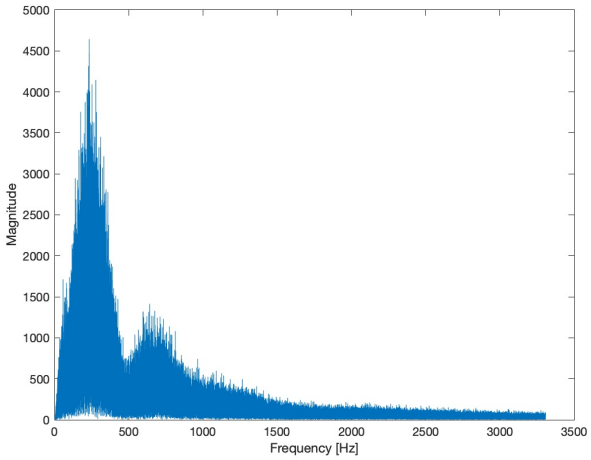

Fig. 7: FFT of the recording of actual waves [1]

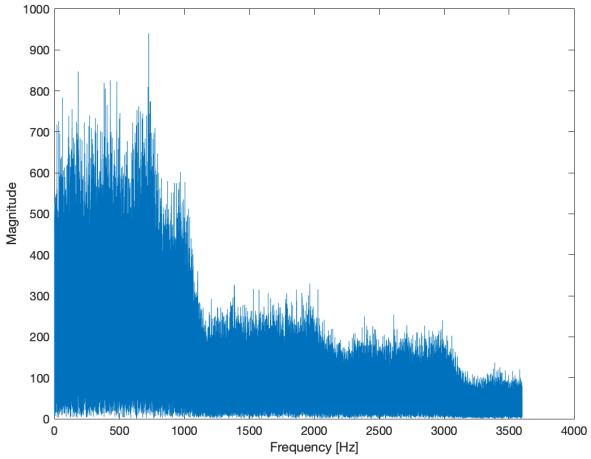

Fig. 8: FFT of the audio produced by the transmitter program [2]

for noise to interfere with the information very easily. It also meant that the data had to be sent at a very slow rate so that it emulate the sound of the ocean. However as these were the requirements of the project, they aren't really drawbacks to the specific system we designed. The implementation of "bubbles" of noise in the frequency domain allowed the emulation of ocean sounds to be done quite easily, but it did also mean that one bit would take up a large band, in this case 1000Hz band. This meant the total frequency spectrum that could be used was cut down into sections of 1000Hz, which leads to a maximum transmission of 24 bits, as 48000Hz was the chosen baud rate. The method of encoding bits in different frequency bands allowed the system to transmit more than one bit at a time, which gave the system an advantage when compared to one that only used amplitude modulation. The method of using "noise bubbles" with such a wide spectrum allowed the detection algorithm to easily identify the logical 1s and logical 0, as the the power is more distributed and thus the effect of randomness of the noise is lost, and can be more deterministic. The fact that each 4 bits is sent for 2 seconds also helped this fact.

In conclusion, the original aim of creating sea sound data caster was designed and successfully implemented.

REFERENCES

[1] "Wikipedia Letter Frequency," http://https://en.wikipedia.org/wiki/Letter$_frequency/$.
[2] "Huffman coding," https://dodona.ugent.be/en/exercises/601074743/.