Acoustic Beamforming using a TDS3230 DSK:
Final Report

Steven Bell  Nathan West
Student Member, IEEE  Student Member, IEEE

Electrical and Computer Engineering
Oklahoma Christian University
## CONTENTS

I  Introduction  1

II  Theory  1
   II-A  Delay and Sum Method  1
   II-B  Microphone Array Design  1
      II-B1  Microphone spacing  1

III  Simulation  2
   III-A  Source localization  2
   III-B  Spatial Filtering  3

IV  System Requirements  3

V  Design  4
   V-A  Hardware  4
   V-B  DSP Software  4
   V-C  Interface Software  5
      V-C1  DSK-PC Interface  5
      V-C2  Interface GUI  5

VI  Test Methods  6
   VI-A  Source localization  6
   VI-B  Spatial filtering  6

VII  Results  6

VIII  Discussion  6
     VIII-A  Sources of difficulty  6

IX  Conclusion  7

References  7

Appendix A: MATLAB code for beam-sweep source localization  8

Appendix B: MATLAB code for spatial filtering  11

Appendix C: C code for main processing loop  14

Appendix D: C code for summing delays  16

Appendix E: C code for calculating delays  17

Appendix F: Java Code - Main  17

Appendix G: Java Code for Beamformer Communication  18

Appendix H: Java Code for Beam Display  21
Abstract—Acoustic beamforming is the use of a microphone array to determine the location of an audio source or to filter audio based on its direction of arrival. For this project, we simulated and implemented a real-time acoustic beamformer using MATLAB for simulations and the TDS3230 DSK for the real-time implementation. Although the final system does not meet all of our initial goals, it does successfully demonstrate beamforming concepts.

I. INTRODUCTION

Most broadly, beamforming is the use of an array of antennas - or in the case of audio, microphones - to perform signal processing based on the spatial characteristics of a signal. We will discuss two primary forms of beamforming in this document: source localization and spatial filtering. Source localization attempts to determine the location in space a signal originated from, while spatial filtering creates an electronically-steerable narrow-beam antenna, which has gain in one direction and strong attenuation in others. Spatial filtering systems and corresponding transmission techniques can replace physically moving antennas, such as those used for radar.

Acoustic source localization is familiar to all of us: we have two ears, and by using them together, our brain can tell where sounds come from. Similarly, our brains are able to focus on one particular sound and tune out the rest, even when the surrounding noise is much louder than the sound we are trying to hear.

II. THEORY

A. Delay and Sum Method

If we have an array of microphones and sufficient signal-processing capability, we can measure the time delay from the time the sound strikes the first microphone until it strikes the second. If we assume waves originate far enough away that we can treat the edge of its propagation as a plane, then the delays are simple to model with trigonometry as shown in Figure 1.

Suppose we have a linear array of microphones, $m_1$ through $m_n$, each spaced $D_{mic}$ meters apart. Then $D_{delay}$, the extra distance the sound has to travel for each successive microphone, is given by

$$D_{delay} = D_{mic} \cdot \cos(\theta)$$  \hspace{1cm} (1)

At sea level, the speed of sound is $340.29 \frac{m}{s}$, which means that the time delay is

$$T_{delay} = \frac{D_{mic}}{340.29 \frac{m}{s}} \cdot \cos(\theta)$$  \hspace{1cm} (2)

By reversing this delay for each microphone and summing the inputs, we can recover the original signal. If a signal comes from a different direction, the delays will be different, and as a result, the individual signals will not line up and will tend to cancel each other when added. This essentially creates a spatial filter, which we can point in any direction by changing the delays.

To determine the direction a signal came from, we can sweep our beam around the room, and record the total power of the signal received for each beam. The source came from the direction with the highest-power signal.

B. Microphone Array Design

Microphone arrays can be nearly any shape: linear, circular, rectangular, or even spherical. A one-dimensional array allows beamforming in one dimension; additional array dimensions allow for 2-dimensional beamforming. Given the limited number of microphones and amount of time we have, a linear array is the best choice.

1) Microphone spacing: The spacing of the microphones is driven by the intended operating frequency range.

For spatial filtering, a narrower beam width is an advantage, because signals which are not directly from the intended direction are attenuated. A narrow beam width is analogous to a narrow transition band for a traditional filter. Lower frequencies will correlate better with delayed versions of themselves than high frequencies, so the lower the frequency, the broader the beam. Conversely, a longer array will result in a greater delay between the end microphones, and will thus reduce the beam width.

At the same time, the spacing between microphones determines the highest operating frequency. If the wavelength of the incoming signal is less than the spacing between the microphones, then spatial aliasing occurs. An example is shown in Figure 2.

The spacing between microphones causes a maximum time delay which, together with the sampling frequency, limits the number of unique beams that can be made.
The maximum number of beams, $max_{beams}$, is shown mathematically in Equation 3.

$$max_{beams} = 2 \cdot F_s \cdot time_{spacing}$$  \hspace{1cm} (3)

The variable $time_{spacing}$ is the maximum amount of time it takes for sound to travel from one microphone to an adjacent microphone, as the case when the source is along the line created by the array.

### III. SIMULATION

Source localization and spatial filtering both use essentially the same simulation code. First, we define a coordinate system where our microphones are placed, with the origin at the center of our linear array. We defined the distance between microphones to be 25 cm, making total array lengths of 75 cm for a four microphone array and 25 cm for a two microphone array.

#### A. Source localization

In the source localization code, a single source is located 10 meters away from the center of the array. A loop creates thirty beams which cover a 180-degree range, and the power for each beam is computed. Figure 3 shows the power sweeps for five different source angles. Note that in each case, the maximum power is exactly at the angle the sound originates from.

Using the beam-sweeping technique, we examined the relative effects of the number of microphones and the source frequency. Figure 4 shows a comparison of the beamwidth as a function of the number of microphones. Figure 5 shows a comparison of beam width versus frequency. Note that higher frequencies produce narrower beams, so long as spatial aliasing does not occur.
Figure 6. Decibel-scale plot of a 4-microphone array beampattern for a 600 Hz signal at 90 degrees. Note the relatively large sidelobes on both sides of the main beam.

Figure 6 shows a plot of the beam in decibels, which brings out a pair of relatively large sidelobes. These can be reduced by windowing the microphone inputs, in the same way that a filter can be windowed to produce lower sidelobes in exchange for a wider transition band. However, this windowing is unlikely to be useful for our small array.

The MATLAB source code is in Appendix A.

B. Spatial Filtering

For spatial filtering, one source is placed at 10 m normal to the array and 10 m tangentially from the center of the array. Using this location as a reference, we can place a secondary source equidistant from the center of the microphone array but some degree offset.

For this simulation set we used two offsets, 60 degrees and 90 degrees. The formed beam was looking 45 degrees away from the array (towards the source at 10, 10). Two microphones were used in simulations because of suggestions by Dr. Waldo and four were used because that is the maximum number of input channels on our hardware. In Figure 7 the simulation has a variable number of input channels for the same sources separated by 60 and 90 degrees. This simulation was the influence for the specification calling for 3dB down on a 60 degree source separation. The SNR is improved drastically when the same sources are separated by 90 degrees still using a 2 microphone array, as seen in Figure 8.

Figures 9 and 10 show similar but improved results when the array is increased to four microphones.

The MATLAB source code is in Appendix B.

IV. SYSTEM REQUIREMENTS

- The system must have at least two microphones as its input.
The system must have a GUI running on a computer which communicates with the DSK board.

In localization mode:
- The system will be able to locate the direction of a 400 Hz tone between -90 and +90 degrees, with +/- 5 degrees error. This target tone will be significantly above the noise threshold in the room.
- The GUI must display the direction of the incoming sound with a latency less than 1 second.

In directional enhancement mode:
- The GUI must allow the user to select a particular direction to listen from, between -90 and +90 degrees, and with a resolution of at least 10 degrees.
- The system should play the selected directional output via one of the line out channels.
- A 300 Hz noise source with the same power will be placed 60 degrees apart from the desired signal source. The directional output should give a signal-to-noise ratio of at least 3 dB.
- The system will operate properly when the sound sources are at least 4 meters away from the microphone array.

V. Design

The acoustic beam forming system consists of three high level subsystems: the microphone array, the DSK board, and a GUI running on a PC. A block diagram showing the interconnections of these subsystems is shown in Figure 11. The microphone array should contain two to four microphones placed in a straight line 25 cm apart. Theses microphones will be connected to the microphone inputs on the DSK_AUDIO4 daughtercard. The DSK will be programmed to process a 256 sample block at a time. The software delays each microphone’s input by a different number of samples then adds each input to create the output. In source localization mode, the power of the output is calculated and sent to the PC GUI. In spatial filtering mode, the output is passed to one of the board’s output channels.

A. Hardware

In addition to the standard DSK board provided to the DSP class we will be using the Educational DSP DSK_AUDIO4 daughtercard, which has four input channels. Each of these channels will be connected to one of the generic mono microphones provided to the class by Dr. Waldo. The placement of the microphones introduces a physical variable to the system that is important to the operation of hardware: according to our specifications and simulations the microphones should be spaced 25 cm apart. Ideally, they will face the direction of incoming acoustic sources for maximum input power.

B. DSP Software

The foundation of our DSP software is the sample code provided by Educational DSP and modified by Dr. Waldo. This code initializes the DSP and the codec and calls a processing function that provides samples from all four input channels. Using this as a base we will implement our main function inside of the processing function provided. In this main file we will also include a file that defines our sum and delay functions which will exist in their own files.

The main software branch opens a RTDX channel with the PC GUI and keeps track of the current operating mode. The mode is a boolean value where a TRUE means spatial filtering mode and a FALSE means source localization mode. In the spatial aliasing mode we do a simple delay and sum of the input data. The delay is given by the GUI. In localization mode the current output is also the delay and sum of the input; however, the delay sweeps from a minimum possible delay to a maximum possible delay defined by the number of samples it takes to go from $-90^\circ$ to $90^\circ$. After every delay and sum operation the delay/powen pairs will be sent to the GUI for display via the RTDX channel. The flow of this is shown in Figure 12.
To determine how much to delay each microphone by, we get the number of samples each microphone should be delayed by from the GUI and multiply by the order of that microphone in the array. The first microphone in the array is defined as microphone 0 and has a 0 delay. If the delay we get from the GUI is negative the order is reversed so that the earliest input always has a 0 delay. The calcDelay function will return a pointed to an array called delays. Figure 13 shows the structure and flow for calcDelay.

The sum function accepts the current four input samples and delay from the GUI and passes it to calcDelays. Figure 14 shows a program flow for this function. It will have a reverse index tracked buffer so that if $i=5$ is the current input $i=6$ is the oldest input and $i=4$ is the second most recent input. This requires fewer rollover-error-checking comparisons relative to a forward-tracking buffer, which should give us an incremental speed improvement.

### C. Interface Software

1) **DSK-PC Interface:** The PC will communicate with the DSK board using RTDX 2.0. For speed, the DSK will only perform integer operations; thus, the PC will perform the floating-point work of translating delays to angles and vice-versa. There will be two RTDX IO channels, one going each direction.

   a) **Power-vs-Delay to PC:**
   - Delay will be a 2’s complement 8 bit signed number, which represents the relative delay between microphones. This will be calculated from the microphone spacing and the speed of sound.
   - Power will be a 16-bit unsigned number, which holds the relative power of the signal for a particular beam.

   In order for these two values to remain correlated, we will interleave both on one channel, and signal the start of each new delay-power pair using the value 0xFF. This will allow our system to recover from lost or delayed bytes.

   This data will only be sent when the system is in source localization mode.

   ![Byte transmission order for DSK → PC communication](image)

<table>
<thead>
<tr>
<th>Use</th>
<th>Code</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microphone time spacing</td>
<td>0-100</td>
<td></td>
</tr>
<tr>
<td>Number of microphones</td>
<td>2-4</td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td>1-2</td>
<td></td>
</tr>
<tr>
<td>Beam selection</td>
<td>-100 to +100</td>
<td></td>
</tr>
</tbody>
</table>

   Figure 16. Byte transmission order for PC → DSK communication

   We found that we did not need any synchronization bytes, and we trimmed the communication down to a single byte for the power paired with a single byte for delay. This was intended to speed up RTDX communication, but was not successful.

   b) **Settings to DSK Board:** The GUI will control several settings, summarized in Table I. Physical variables such as microphone spacing and number of microphones need to be easily changed by software in real time, and the GUI provides an interface for users to the DSK to change the operation as required.

   The byte stream will follow the same format as the DSK to PC stream. Data will be sent in 3 byte packets as shown in Figure 16. Packets will only be sent when there is new data, so we expect the number of packets per second to be very low.

   We did not implement this due to time constraints.

2) **Interface GUI:** The interface software will be written in Java. An overview of the main classes is shown in Figure 17.

   The Beamformer class handles the low-level translation and communication with the DSK board. It converts from sample delays to angles and vice-versa, converts booleans to code values, and floating-point values to integers for high-speed communication.

   The BeamDisplay is a GUI class which displays the beam power levels and sound direction on a polar plot, similar to the simulation plots. It will use the Java AWT/Swing toolkit and associated drawing classes to implement a custom drawable canvas. Clicking on the display will select the direction to use when in spatial filtering mode.

   The Settings panel will contain a series of standard Swing components to select the operating mode, number of microphones, microphone spacing, the number of beams per sweep. It will have a pointer to the Beamformer object, so it can configure it directly.

   The MainWindow class creates the main GUI window, the display and control panels, and the connection to the DSK board. Once running, it will regularly poll the beamformer class for new data, and update the
We were unable to get any kind of beamforming to work with four microphones. After getting the 2-microphone system working, we switched to 4 by changing a `#define` in our code and connecting two additional microphones to the DSK expansion board. When we ran the GUI, we were unable to observe any kind of localization.

Each adjacent pair of microphones worked in a 2-microphone beamformer, so we believe that it was not a simple case of one microphone not working properly or a result of mixing up the order of the microphones. We performed a test where we merely averaged the inputs from the four microphones and sent that value to the output, which is equivalent to forming a beam perpendicular to the array. Using one of the speakers as an audio source and measuring the RMS voltage of the output on the oscilloscope, we manually checked the directionality of the array. We were unable to measure any significant difference between the sound source being directly orthogonal to the array and at a small angle to it. This indicates that at least part of the problem is not due to our software. There may have been additional problems with our software for four microphones, but because we were unable to get this simple test working, we did not test our software further.

VIII. DISCUSSION

A. Sources of difficulty

We identified several factors which contributed to our system’s failure to meet its requirements.

First, there were several acoustic problems. We found that the supplied microphones are very directional - they are specifically labeled “unidirectional”. Because the beamforming system assumes omnidirectional input, unidirectional microphones will cause oblique signals to be attenuated. This is fine when the desired beam is perpendicular to the array, but is suddenly a problem when the goal is to amplify oblique signals over orthogonal ones.

The audio input level from the microphones seemed to be rather low, producing averaged output values in the single-millivolt range. We are not sure if there is a way to configure additional gain on the DSK board, or if using different microphones would help. Due to the low input level, we had to play the signal very loudly on the speakers and keep them within one or two meters of the array.

A 1-meter source distance clearly violates the farfield plane-wave assumption - our initial specification assumed that 4 meters was farfield. This may have caused some problems, but re-running our MATLAB simulation using a source only 1m away produced a beam virtually indistinguishable from a 10-meter beam.
We also experienced very strong echoes in the room, because of its cement construction and solid wood lab benches. Working in an environment with reduced echoes would probably have yielded better results. This was evident during late nights in the lab when people working in the ASAP room and janitorial staff would peek into the room to find the very loud noise echoing throughout the HSH.

On the software side, we experienced difficulties getting the bandwidth we desired over RTDX. I had also seen poor RTDX performance before, when sending data from a PC to the DSK during the cosine generation lab. There are several possible causes of this: One is that we only send a few bytes at a time, but do it dozens of times per second. Recommendations we found online suggested that filling the RTDX queue and sending large packets of data is the fastest way to send data.

Another likely cause is our DSK boards. We read a mailing list correspondence between someone who is familiar with this family of DSKs and a hapless hobbyist having similar issues. The “expert” claimed that there is obfuscation hardware on the DSK boards introduced by Spectrum Digital that slows down RTDX communication. It was mentioned that a possible workaround is using the JTAG connector on the DSK board, but the hardware required for this is not readily available (and the JTAG port is blocked by the DSP_AUDIO_4). We looked at high-speed RTDX, but it is not supported on this platform.

IX. Conclusion

Although we did not meet our initial specifications, our project was still somewhat successful. We were able to implement a working beamforming system, and most of the problems we encountered were limitations of our equipment which were not under our control.

References

% General setup
clear;

% Speed of sound at sea level
speedOfSound = 340.29; % m/s

% Source setup
% Frequency of the audio source
sourceFreq = 700; % Hz

% Just so we know - since this will affect our mic placement
wavelength = speedOfSound / sourceFreq;

% xy location of audio source
sourceLocX = 0; % in meters
sourceLocY = 10;

% Microphone setup
numMicrophones = 4;

% Distance between microphones - determines the highest frequency we can
% sample without spatial aliasing
micSpacing = 0.25; % meters

% Minimum amount of time it takes for sound to travel from one mic to the next
timeSpacing = micSpacing/speedOfSound;

% Total width of the array - determines the lowest frequency we can
% accurately locate
arrayWidth = (numMicrophones - 1) * micSpacing;

% xy locations of the microphones
micLocX = linspace(-arrayWidth/2, arrayWidth/2, numMicrophones); % in meters
micLocY = zeros(1, numMicrophones);

% Sampling rate
Fs = 44100; % Hz

% Distance from the source to the mic
propDistance = hypot(sourceLocX - micLocX, sourceLocY - micLocY);
timeDelay = propDistance/speedOfSound;

% Create some of the signal
soundTime = 0:1/Fs:.125;
sourceSignal = sin(2 * pi * sourceFreq * soundTime);
% Delay it by the propagation delay for each mic
for ii = 1:numMicrophones
    received(ii,:) = delay(sourceSignal, timeDelay(ii), Fs);
end

% Plot the signals received by each microphone
% plot(received');

% Now it’s time for some beamforming!
% Create an array of delays
numBeams = 60;
beamDelays = linspace(-timeSpacing, timeSpacing, numBeams);

for ii = 1:numBeams
    summedInput = zeros(1, length(sourceSignal));
    for jj = 1:numMicrophones
        if(beamDelays(ii) >= 0)
            % Delay the microphones by increased amounts as we go left to right
            summedInput = summedInput + ...
                delay(received(jj,:), beamDelays(ii) * (jj - 1), Fs);
        else
            % If the beam delay is negative, that means we want to increase the
            % delay as we go right to left.
            summedInput = summedInput + ...
                delay(received(jj,:), -beamDelays(ii) * (numMicrophones - jj), Fs);
        end
    end
    % Calculate the power of this summed beam
    power(ii) = sum(summedInput .^ 2);
end

% directions = linspace(pi, 0, numBeams);
directions = acos(linspace(-1, 1, numBeams));
polar(directions, power/max(power), ’-b’);
function [outputSignal] = delay(inputSignal, time, Fs)
% DELAY - Simulates a delay in a discrete time signal
%
% USAGE:
% outputSignal = delay(inputSignal, time, Fs)
%
% INPUT:
% inputSignal - DT signal to operate on
% time - Time delay to use
% Fs - Sample rate

if (time < 0)
    error('Time delay must be positive');
end

outputSignal = [zeros(1, floor(time * Fs)), inputSignal];

% Crop the output signal to the length of the input signal
outputSignal = outputSignal(1:length(inputSignal));
% General setup

% The frequencies we are interested in are:
% Right now this is optimized for sources between 100 and 500 Hz
% Change this to whatever range you are interested in.

fmin = 0; % Hz
fmax = 700; % Hz

% Speed of sound at sea level
speedOfSound = 340.29; % m/s

% Source setup
% Frequency of the audio sources
source1Freq = 300; % Hz
source2Freq = 400; % Hz

% Just so we know — since this will affect our mic placement
wavelength1 = speedOfSound / source1Freq;
wavelength2 = speedOfSound / source2Freq;

degreesApart = 60;

% xy location of audio source
source1LocX = sqrt(10^2 + 10^2) * cosd(45 + degreesApart); % in meters
source1LocY = sqrt((10^2 + 10^2) - source1LocX^2);

source2LocX = 10; % Secondary source
source2LocY = 10;

% Microphone setup
numMicrophones = 2;

% Distance between microphones — determines the highest frequency we can
% sample without spatial aliasing
micSpacing = 0.25; % meters

% Minimum amount of time it takes for sound to travel from one mic to the next
timeSpacing = micSpacing / speedOfSound;

% Total width of the array — determines the lowest frequency we can
% accurately locate
arrayWidth = (numMicrophones - 1) * micSpacing;

% xy locations of the microphones
micLocX = linspace(-arrayWidth/2, arrayWidth/2, numMicrophones); % in meters
micLocY = zeros(1, numMicrophones);

% Sampling rate
Fs = 96000; % Hz
% Distance from the source to the mic
prop1Distance = hypot(source1LocX - micLocX, source1LocY - micLocY);
prop2Distance = hypot(source2LocX - micLocX, source2LocY - micLocY);

time1Delay = prop1Distance/speedOfSound;
time2Delay = prop2Distance/speedOfSound;

% Create some of the signal
soundTime = 0:1/Fs:.125;

source1Signal = sin(2 * pi * source1Freq * soundTime);
source2Signal = sin(2 * pi * source2Freq * soundTime);

% Delay it by the propagation delay for each mic
for ii = 1:numMicrophones
    received(ii,:) = delay(source1Signal, time1Delay(ii), Fs) +
    delay(source2Signal, time2Delay(ii), Fs);
end

% Direct the beam towards a location of interest
angleWanted = 45; % Degrees (for simplicity)
angleToDelay = angleWanted * pi/180; % Convert to radian

% We want to take the fft of the signal only after every microphone is
% getting all of the data from all sources
deadAirTime = (max([time1Delay, time2Delay]));
deadAirSamples = deadAirTime * Fs;
endOfCapture = length(received(1,:));

% Start off with an empty matrix
formedBeam = zeros(1, max(length(received)));

% For each microphone add together the sound received
for jj = 1:numMicrophones
    formedBeam = formedBeam +
    delay(received(jj,:), + timeSpacing*sin(angleToDelay) * (jj-1), Fs);
end

% Get the PSD object using a modified covariance
beamPSD = psd(spectrum.mcov, formedBeam,'Fs',44100);
% Get the magnitude of the PSD
formedSpectrum = abs(fft(formedBeam));
% The fft sample # needs to be scaled to get frequency
fftScaleFactor = Fs/numel(formedSpectrum);

% The frequencies we are interested in are:
% Right now this is optimized for sources between 100 and 500 Hz
fmin = 0; % Hz
fmax = 600;% Hz

% Plot the PSD of the received signal
% Get the frequencies and data out of the PSD object
beamPSDfreqs = beamPSD.Frequencies;
beamPSDdata = beamPSD.Data;

% Get the indexes that correspond to frequencies specified above
indexesToPlot = find(beamPSDfreqs > fmin - 1); find(beamPSDfreqs > fmax, 1);

% Actually plot it (in a log10 scale so we have dB)
plot(beamPSDfreqs(indexesToPlot), 20*log10(beamPSDdata(indexesToPlot)));
title('PSD using a modified covariance of the received signal');
ylabel('Power/Frequency (dB)');
xlabel('Frequency (Hz)');

% Plot the fft of our received signals
figure(2);
maxLimit = round(fmax/fftScaleFactor);
minLimit = round(fmin/fftScaleFactor);
if minLimit <= 0
    minLimit = 1;
end
f = linspace(0, 44100, 44100/fftScaleFactor);

% This gets all of the fft frequencies we want to look at
fOfInterest = f(minLimit:maxLimit);
% Grab the portion of the fft we want to look at
spectrumOfInterest = formedSpectrum(minLimit:maxLimit);
% Normalize this so that the max amplitude is at 0db.
spectrumOfInterest = spectrumOfInterest/max(formedSpectrum);
% Plot it
plot(fOfInterest, 20*log10(spectrumOfInterest));
title('FFT of the received signal');
ylabel('Relative magnitude of signal (dB)');
xlabel('Frequency (Hz)');
void processing(){
    Int16 twoMicSamples[2];

    Int32 totalPower; // Total power for the data block, sent to the PC
    Int16 tempPower; // Value used to hold the value to be squared and added
    Int16 printCount; // Used to keep track of the number of loops since we last sent data
    unsigned char powerToSend;

    Int8 firstLocalizationLoop; // Whether this is our first loop in localization mode
    Int8 delay; // Delay amount for localization mode

    QUE_McBSP_Msg tx_msg, rx_msg;
    int i = 0; // Used to iterate through samples

    RTDX_enableOutput( &dataChan );

    while(1){
        SEM_pend(&SEM_McBSP_RX,SYS_FOREVER);

        // Get the input data array and output array
        tx_msg = QUE_get(&QUE_McBSP_Free);
        rx_msg = QUE_get(&QUE_McBSP_RX);

        // Spatial filter mode

        // Localization mode
        if(firstLocalizationLoop){
            delay = DELAYMIN;
            firstLocalizationLoop = FALSE;
        }
        else{
            delay++;
            if(delay > DELAYMAX){
                delay = DELAYMIN;
            }
        }

        // Process the data here
        /* MIC1R data[0]
         MIC1L data[1]
         MIC0R data[2]
         MIC0L data[3]
         * OUT1 Right - data[0]
         * OUT1 Left - data[1]
         * OUT0 Right - data[2]
         * OUT0 Left - data[3]
         */
totalPower = 0;

    for(i=0; i<QUE_McBSP_LEN; i++){
        // Put the array elements in order
        twoMicSamples[0] = *(rx_msg->data + i*4 + 2);
        twoMicSamples[1] = *(rx_msg->data + i*4 + 3);
        tx_msg->data[i*4] = 0;
        tx_msg->data[i*4 + 1] = 0;
        tx_msg->data[i*4 + 2] = sum(twoMicSamples, delay);
        tx_msg->data[i*4 + 3] = 0;
C code for summing delays

Listing 1. sum.h

```c
#ifndef __SUM_H_BFORM__
#define __SUM_H_BFORM__
extern Int16 sum(Int16* newSamples, int delay);
#endif
```

Listing 2. sum.c

```c
#include <std.h>
#include "definitions.h"
#include "calcDelay.h"

/* newSamples is an array with each of the four samples.
* delayIncrement is the amount to delay each microphone by, in samples.
* This function returns a "beamed" sample. */
Int16 sum(Int16* newSamples, int delayIncrement){
    static int currInput = 0; // Buffer index of current input
    int delays[NUMMICS]; // Amount to delay each microphone by
    int mic = 0; // Used as we iterate through the mics
    int rolloverIndex;
    Int16 output = 0;
    static Int16 sampleBuffer[NUMMICS][MAXSAMPLEDIFF];

    // Calculate samples to delay for each mic
    // TODO: Only do this once
    calcDelay(delayIncrement, delays);

    // We used to count backwards - was there a good reason?
    currInput++; // Move one space forward in the buffer

    // Don't run off the end of the array
    if(currInput >= MAXSAMPLEDIFF){
        currInput = 0;
    }

    // Store new samples into sampleBuffer
    for(mic=0; mic < NUMMICS; mic++)
        // Divide by the number of microphones so it doesn’t overflow
        // when we add them
        sampleBuffer[mic][currInput] = newSamples[mic]/NUMMICS;

    // For each mic add the delayed input to the current output
    for(mic=0; mic < NUMMICS; mic++){
        if(currInput - delays[mic] >= 0){// Index properly?
            output += sampleBuffer[mic][currInput - delays[mic]];
        }
        else{
            // The delay index is below 0, so add the length of the array
            output += sampleBuffer[mic][currInput - delays[mic]];
        }
    }
}
```
rolloverIndex = MAXSAMPLEDIFF + (currInput - delays[mic]);
output += sampleBuffer[mic][rolloverIndex];

return output;

Listing 3. calcDelay.h
#ifndef _CALC_DELAY_H_
define _CALC_DELAY_H_
extern void calcDelay(int delayInSamples, int* delays);
#endif

Listing 4. calcDelay.c
/* calcDelay
* Accepts delays in samples as an integer
* and returns a pointer to an array of delays
* for each microphone.
* 
* Date: 9 March 2010
*/
#include "definitions.h"

void calcDelay(int delayInSamples, int* delays){
    int mic = 0;
    if(delayInSamples > 0){
        for(mic=0; mic < NUMMICS; mic++){
            delays[mic] = delayInSamples*mic;
        }
    }
    else{
        for(mic=0; mic < NUMMICS; mic++){
            delays[mic] = delayInSamples*(mic-(NUMMICS-1));
        }
    }
}

Listing 5. MainWindow.java
/* MainWindow.java - Java class which constructs the main GUI window
* for the project, and sets up communication with the DSK. It also
* contains the main() method.
* 
* Author: Steven Bell and Nathan West
* Date: 9 March 2010
*/

APPENDIX E
C code for calculating delays

APPENDIX F
Java code - Main
package beamformer;

import java.awt.*; // GUI Libraries
import javax.swing.*;

public class MainWindow
{
    JFrame window;
    DisplayPanel display;
    ControlPanel controls;
    static Beamformer beamer;

    MainWindow()
    {
        beamer = new Beamformer();
        window = new JFrame("Acoustic Beamforming GUI");
        window.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        window.getContentPane().setLayout(new BoxLayout(window.getContentPane(), BoxLayout.LINE_AXIS));

        display = new DisplayPanel();
        display.setPreferredSize(new Dimension(500,500));
        window.add(display);

        controls = new ControlPanel();
        controls.setAlignmentY(Component.TOP_ALIGNMENT);
        window.add(controls);

        window.pack();
        window.setVisible(true);
    }

    public static void main(String[] args)
    {
        MainWindow w = new MainWindow();
        while(true)
        {
            beamer.update();
            w.display.updatePower(beamer.getPower());
        }
    }
}

APPENDIX G
JAVA CODE FOR BEAMFORMER COMMUNICATION

Listing 6. Beamformer.java
/* Beamformer.java - Java class which interacts with the DSK board
 * using RTDX.
 */
package beamformer;

//Import the DSS packages
import com.ti.ccstudio.scripting.environment.*;
import com.ti.debug.engine.scripting.*;

public class Beamformer {
    DebugServer debugServer;
    DebugSession debugSession;
    RTDXInputStream inStream;
    int[] mPowerValues;

    public Beamformer() {
        mPowerValues = new int[67]; // TODO: Change to numBeams
        ScriptingEnvironment env = ScriptingEnvironment.instance();
        debugServer = null;
        debugSession = null;

        try {
            // Get the Debug Server and start a Debug Session
            debugServer = (DebugServer) env.getServer("DebugServer.1");
            debugServer.setConfig("Z:/2010_Spring/dsp/codecomposer_workspace/dsp_project_trunk/dskboard.ccxml");
            debugSession = debugServer.openSession(".*");

            // Connect to the target
            debugSession.target.connect();
            System.out.println("Connected to target.");

            // Load the program
            debugSession.memory.loadProgram("Z:/2010_Spring/dsp/codecomposer_workspace/dsp_project_trunk/Debug/beamformer.out");
            System.out.println("Program loaded.");

            // Get the RTDX server
            RTDXServer commServer = (RTDXServer) env.getServer("RTDXServer.1");
            System.out.println("RTDX server opened.");

            RTDXSession commSession = commServer.openSession(debugSession);

            // Set up the RTDX input channel
            inStream = new RTDXInputStream(commSession, "dataChan");
            inStream.enable();
        }
        catch (Exception e)
public void start()
{
    // Start running the program on the DSK
    try{
        debugSession.target.restart();
        System.out.println("Target restarted.");
        debugSession.target.runAsynch();
        System.out.println("Program running....");
        Thread.currentThread().sleep(1000); // Wait a second for the program to run
    }
    catch (Exception e)
    {
        System.out.println(e.toString());
    }
}

public void stop()
{
    // Stop running the program on the DSK
    try{
        debugSession.target.halt();
        System.out.println("Program halted.");
    }
    catch (Exception e)
    {
        System.out.println(e.toString());
    }
}

public void setMode()
{
}

public void setNumBeams()
{
}

public void update(){
    try{
        // Read some bytes
        byte[] power = new byte[1];
        byte[] delay = new byte[1];
        inStream.read(delay, 0, 1, 0); // Read one byte, wait indefinitely for it
        inStream.read(power, 0, 1, 0); // Read one byte, wait indefinitely for it
    }
**Appendix H**

**Java Code for Beam Display**

Listing 7. DisplayPanel.java

```java
/* DisplayPanel.java - Java class which creates a canvas to draw the beamformer output on and handles the drawing. */

* Author: Steven Bell and Nathan West
* Date: 9 March 2010
* $LastChangedBy$
* $LastChangedDate$
*

package beamformer;

import java.awt.*; // GUI Libraries
import javax.swing.*;
import java.lang.Math;

public class DisplayPanel extends JPanel {  

  int mDelay[] = {-3, -2, -1, 0, 1, 2, 3};  
  double mPower[] = {0, .25, .5, 1, .5, .25, 0};  
  int mNumPoints = 67;
  double mSpacing = 33;

  DisplayPanel() {  
```
mDelay = new int[mNumPoints];
mPower = new double[mNumPoints];

for (int i = 0; i < mNumPoints; i++) {
    mDelay[i] = i - 33;
    mPower[i] = .5;
}

void updatePower(int[] newPower) {
    for (int i = 0; i < mNumPoints; i++)
    {
        mPower[i] = (double)newPower[i] / 255;
        if (mPower[i] > 1){
            mPower[i] = 1;
        }
    }
    this.repaint();
}

static final int FIGURE_PADDING = 10;

// Overrides paintComponent from JPanel
protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    Graphics2D g2d = (Graphics2D)g; // Cast to a Graphics2D so we can do more advanced painting
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);

    // Determine the maximum radius we can use. It will either be half
    // of the width, or the full height (since we do a semicircular plot).
    int maxRadius = this.getWidth()/2;
    if (maxRadius > this.getHeight()){
        maxRadius = this.getHeight();
    }
    maxRadius = maxRadius - FIGURE_PADDING;

    // Pick our center point
    int centerX = this.getWidth() / 2;
    int centerY = this.getHeight() - (this.getHeight() - maxRadius) / 2;

    // Calculate all of the points
    int[] px = new int[mNumPoints];
    int[] py = new int[mNumPoints];

    for (int i = 0; i < mNumPoints; i++)
    {
        double angle = delayToAngle(mDelay[i]);
        px[i] = centerX - (int)(maxRadius * mPower[i] * Math.sin(angle));
        py[i] = centerY - (int)(maxRadius * mPower[i] * Math.cos(angle));
    }
g2d.setPaint(Color.BLUE);
g2d.drawPolygon(px, py, mNumPoints);

// Draw the outline of the display, so we have some context
g2d.setPaint(Color.BLACK);
float[] dash = {5, 4};
g2d.setStroke(new BasicStroke((float).5, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER, 1, dash, 0));
g2d.drawLine(10, centerY, this.getWidth() - 10, centerY);
g2d.drawArc(10, (this.getHeight() - maxRadius) / 2, 2*maxRadius, 2*maxRadius, 0, 180);
}

// Takes a delay and converts it to the equivalent in radians
private double delayToAngle(int delay)
{
  return (Math.PI/2 - Math.acos(delay/mSpacing));
}

// Takes an angle in radians (-pi/2 to +pi/2) and converts it to a delay
private int angleToDelay(double angle)
{
  return (int)(mSpacing * Math.cos(angle + Math.PI/2));
}