MASLA

Microphone Arrays for Source Location Applications

18551 – Group 1 – Spring 2004
Akhil Chauhan (achauhan@)
Bryan Chen (bryanche@)
Anup Doshi (adoshi@)
Vinay Kapur (vinayk@)

May 2, 2004

Table of Contents

Introduction

This report endeavors to comprehensively discuss the process and methodology employed in the conception and implementation of our project, Microphone Array Source Location Applications (MASLA). The following sections will delve into the details of the theory behind our project, and the process by which we proceeded to make this idea a reality. Later sections will discuss the implementation of this project on the TI TMS320C67.

In addition to the theory and the implementation of the theory on the TI, we will also discuss in detail numerous issues our group faced on the path to the successful implementation of our project, these include but are not limited to memory/speed issues, synchronization issues and accuracy issues.

The Problem

The original problem which lead us to the realm of Microphone arrays, was our desire to create a project in which we would track an intruder through a room. It was our desire to create a sound sensitive security system. Although being a very worthy problem, it was quickly realized that we did not have the technology to make our project sensitive enough to listen to and track the footsteps of an intruder. We then began to focus on the numerous other applications of Microphone arrays and found that our technology could be best used in terms of video conferencing.

Imagine a professor is giving a lecture which is being videotaped or in this day and age, broadcast over the internet. Our technology would allow us to detect the location of the professor as he walked around the room, and feed that location to a camera to follow the professor around. Also in a corporate environment, when videoconferences are held,

our technology would identify the location of the speaker and instruct a camera to point at that speaker. This would eliminate the need for an individual to manually move the camera around and would allow the speakers to move freely, without concern that they will fall out of the camera's view.

Numerous Applications

In addition the security and videoconferencing applications already mentioned microphone arrays for the purpose of source location have applications in numerous other fields and industries. One such application was for military purposes where research has proposed having automated gun turrets linked to a microphone array system in order to secure a boundary. Additional applications consisted of wildlife research, in terms of monitoring animals in the wild and tracking them based on their footsteps or any noise they create. However after analyzing these different applications, we feel that with the time and money at our disposal, pursuing a project to better videoconferencing would be the most valuable and effective use of our time.

Microphone Arrays

The study of Microphone arrays can be dated back twenty years and the field has matured to the point that microphone array technology has worked its way into numerous existing products and is the focus of on-going research. While there are numerous different uses for Microphone arrays, our project specifications require a simple setup for source location. Although later in this paper we will discuss different formations for the mics, at this point we will discuss using a straight-line formation which is also what we ended up using in our final demonstration.

Figure 1 below illustrates the basic concept behind our project. Figure 1 has two different microphone pairs, and by evaluating the time difference of arrivals of the sound signals at each of these microphones we can learn the location of the source.

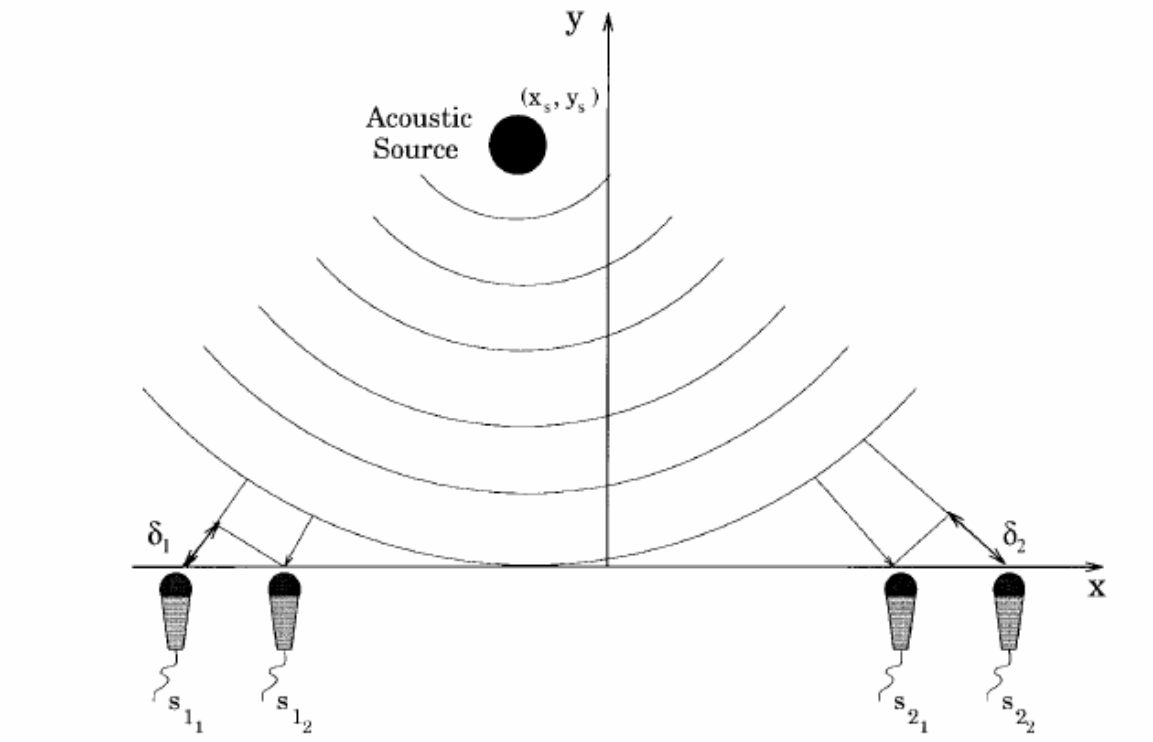

Figure 1: 4 Mic – Microphone Array[1]

Locating a sound source implies the estimation of mutual time delays between the direct-path wavefront arrivals at the sensors, in our case microphones. Once these delays are known the position can be calculated using geometry.

[1] Use of the crosspower-spectrum phase in acoustic event location, Omologo, M.; Svaizer, P., Speech and Audio Processing, IEEE Transactions on, Vol.5, Iss.3, May 1997, Pages:288-292

Given a linear microphone array and a source located in position (x_s, y_s) (Figure 1), the signal $s_{k\,(t)}$ acquired by the microphone "k" can be expressed as

$$s_k(t) = \alpha_k r(t - t_k) + n_k(t)$$

Where r()t indicates the waveform as generated at the source, $alpha_k$ is an attenuation factor due to certain propagation effects which end up not having that large an effect in our final testing environments. t_k is the propagation time of the direct wavefront and $nu_k(t)$ includes all the undesired components, which in a reverberant environment can also be correlated with $r(t)$[2].

Assuming we are operating on simply two dimensions, we can calculate the angle of the sound source. Knowing the mutual delay between the arrival between the mics of each pair we can derive the angle as such:

Theta = arc cos(c * mutual delay / distance between both mics)

c = speed of sound

Beamforming

A common use of phased arrays in general, and one that we looked into, is the concept of beamforming. As a general application of phased array radars, beamforming is used to selectively amplify 'interesting' signals coming from a certain direction, and potentially to reduce the amount of noise as well. Thus it is obvious how this concept could be applicable in our case; however, due to implementation issues as described below we were unable to extend our project to include this concept. Beamforming has

[2] Use of the crosspower-spectrum phase in acoustic event location
Omologo, M.; Svaizer, P.,
Speech and Audio Processing, IEEE Transactions on, Vol.5, Iss.3, May 1997
Pages:288-292

been fairly successfully implemented with microphone arrays though, and several discussions exist in literature[3][4].

This process of beamforming can be thought of as spatial filtering across a set of antennas. In other words, the incoming (assumed coherent) signals from an array of antennas are multiplied by a complex weighting function as described by a spatial filter. If designed correctly, the filter can be applied so as to essentially null out unwanted signals coming from certain directions, and amplify desired signals coming from other directions. Spatial-domain beamforming is analogous to time-domain windowing in that the spatial frequency response is shaped as a 'sinc' function whose main lobe amplitude, zeros, and side lobes can be suitably modified by changing the weights of the spatial function. A naïve approach is commonly referred to as a delay-and-sum, similar to what is described above: by simply applying an appropriate phase shift to each input and then summing them, the input signal coming from a certain direction will be amplified.

Several issues turned out to be critical in preventing us from using beamforming. Foremost of those was the fact that we were using only four microphones. As is obvious from the discussion above, four microphones only translates into four degrees of freedom when adjusting the spatial weighting function; this is certainly not enough to design a useful spatial frequency function. As implemented by Sullivan and others[5], the number of microphones needed to acquire decent results was at least twice the number that we had. Another issue is that speech signals are relatively wide-band signals; this issue may be

[3] Thomas M. Sullivan, Multi-Microphone Correlation-Based Processing for Robust Automatic Speech Recognition (2.2MB), Ph.D Thesis, ECE Department, CMU, August 1996.
[4] Microphone Arrays: Signal Processing Techniques and Applications, M. Brandstein & D. Ward
[5] Microphone Arrays: Signal Processing Techniques and Applications, M. Brandstein & D. Ward

compensated by logarithmically spacing the microphones[6]. Again, the number of microphones we used limited our abilities in this application since we were limited in the availability of inputs. Thus we decided against using traditional beamforming and instead went forward with source location.

The Algorithm

Extensive research has been done in the field of microphone arrays for various purposes such as sound location. Intuitively it seems that source localization can be achieved by using a simple Time-Delay of Arrival (TDOA) technique, which is the difference in phase the sound source reaches the two microphones. By simply correlating the two input signals, we should be able to estimate the time difference and thus deduce the angle of arrival, similar to how human ears work. However it is still fairly difficult to achieve a decent resolution with only two microphones, considering that spatial resolution depends on many factors, including source distance, sampling rate, and microphone accuracy. These variables may have a degrading effect on the estimation of time delays between microphones, thus affecting our ability to reliably predict location. By increasing the number of microphones, it may be possible to improve the resolution somewhat by increasing redundancy.

To overcome some of the problems discussed above such as resolution and due to the constraints we had, we used four microphones in an array like structure. In order to calculate the distance of the sound source from the designated microphone, the time difference of arrival was calculated for each of the two pairs of microphones. Then

[6] Microphone Arrays: Signal Processing Techniques and Applications, M. Brandstein & D. Ward

simple geometry, in particular the law of cosines, is applied to each pair of microphones, to get the angle the sound is incident on the pair of microphones. Once both the angles are known, we use the law of sines to calculate the speaker location from the designated origin microphone.

Cross Power Spectrum Phase

Many different methods can be applied to determine the time delay between two signals. Simple correlation could be used; however a drastic improvement can be achieved when the Cross Power Spectrum Phase is used to calculate the time difference of arrival. Our source for helping us reach the conclusion of using the CPSP algorithm was a paper written by Omologo and Svaizer[7], in which they talked about how the algorithm can be used to calculate the Time Difference of Arrivals for each microphone pairs.

The basic formula for the Cross Power Spectrum Phase is as follows:

$$CPSP = FFT^{-1}\left(\frac{FFT(x_1)FFT(x_2)^*}{\left|FFT(x_1)\right|\left|FFT(x_2)\right|} \right)$$

$$\hat{D} = i: \max_i (CPSP)$$

[8]

Here x1 and x2 are signals received in the left and right channel of the stereo microphones being used for this project. The Fast Fourier Transform of x1 is multiplied by the Fourier Transform of the complex conjugate of the other channel. The phase

[7] Use of the crosspower-spectrum phase in acoustic event location
Omologo, M.; Svaizer, P.
Speech and Audio Processing, IEEE Transactions on, Vol.5, Iss.3, May 1997
Pages:288-292

[8] Speaker localization using microphone array in a reverberant room
Qiyue Zou; Rahardja, S.; Zhibo Cai
Signal Processing, 2002 6th International Conference on, Vol.1, Iss., 26-30 Aug. 2002

information is preserved by dividing the result by the magnitudes of the FFTs of both the signals. Finally the delay is calculated by computing the inverse FFT of the result.

According to the Omologo and Svaizer paper, the resulting function is the transform of an allpass function and constant energy which is basically concentrated on the correct time delay.

Why Cross Power Spectrum Phase?

As discussed before simple correlation of the signals could be used to calculate the time delay of arrivals. But after analysis of the initial simulation of results using two microphones in Matlab and the knowledge we gained from our various references, we decided to use the CPSP instead of simple correlation because of the superiority of the quality of results achieved by using CPSP.

First, when time delays are being calculated they are solely based phase ad the sound signal reaching the both microphones is the same, but the only difference is in phase. When a simple correlation is performed, the magnitude is also considered in the FFTs and therefore this gives us a lot of redundant information causing the result of the correlation to be not so good.

Figure 2 shows the difference in results when we performed both CPSP and cross correlation on the same input of sound. As we can clearly see, the CPSP gives a much better signal to noise ratio (SNR). By SNR, we imply that it is the ratio between the highest peak and the average of all other peaks. Also it can be easily observed that the peak to side lobe ratio is much better in the Cross Power spectrum Phase method. Therefore the accuracy in the calculation of the time delays is superior when the CPSP algorithm is used instead of simple correlation.

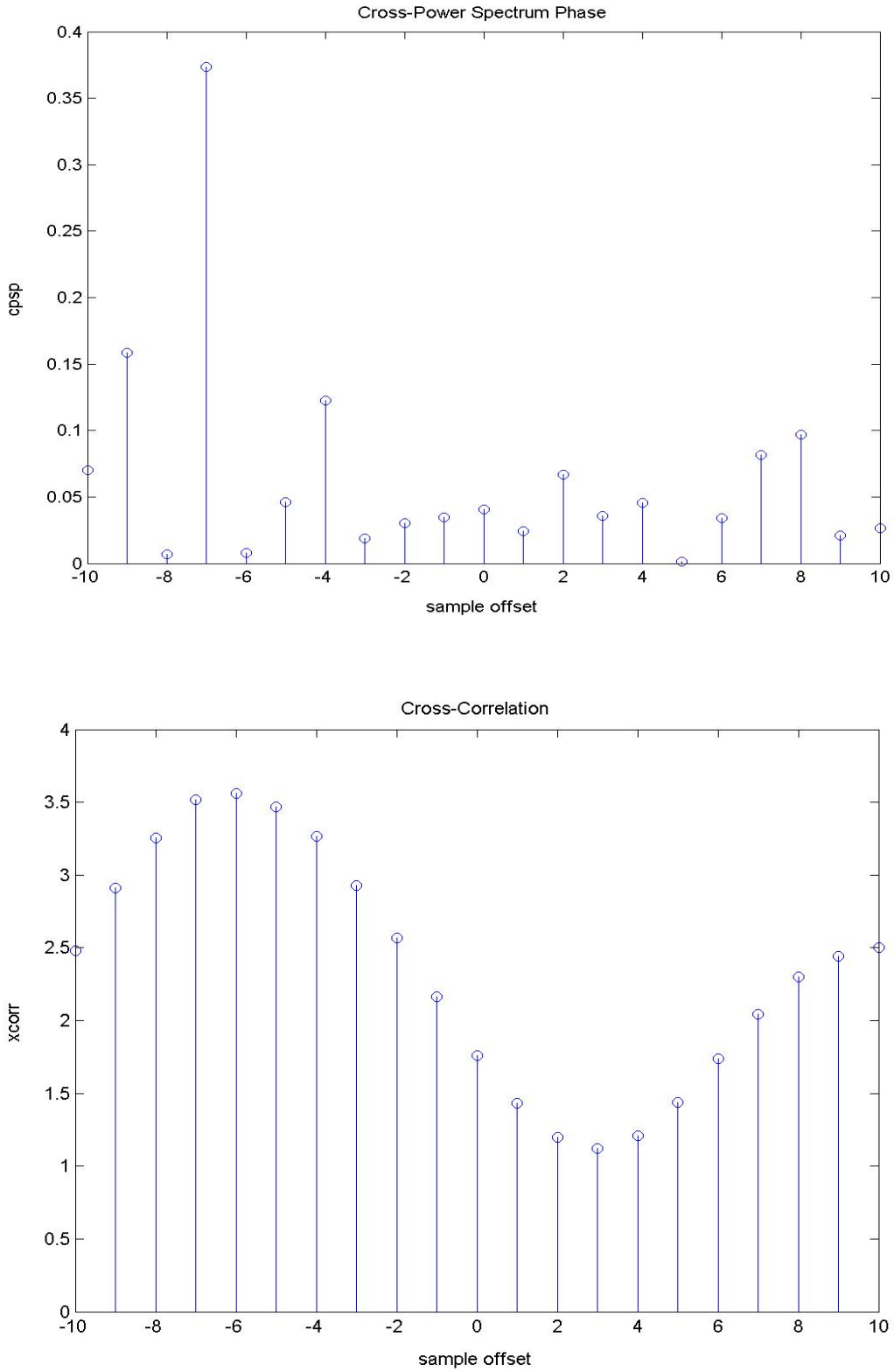

Figure 2: Comparison of results between CPSP and Cross Correlation

Geometry

Once the time difference of arrival is calculated for both pairs of microphones, the next step in our process of sound location is the calculation of the angle of the source relative to each pair of microphones. Figure 3 below shows the calculation of the angle for each pair of microphones.

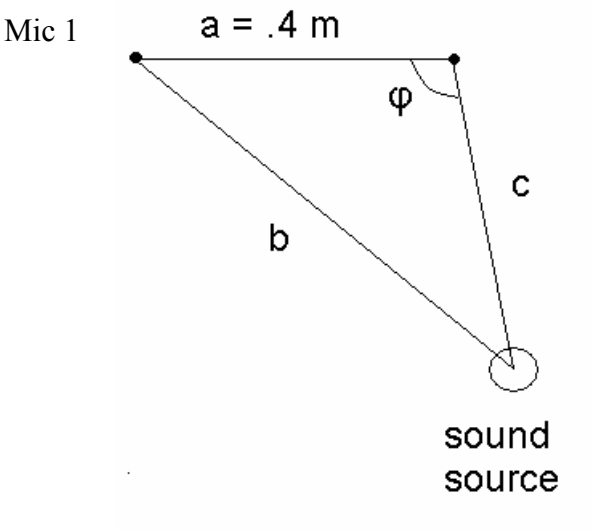

Figure 3: Simple Geometry

The microphone pairs in our project are always separated by .4 m, since experimental results had previously shown that this was a fairly optimal distance for using CPSP.[9] Basic calculations are as follows:

- a is the distance between the mics.
- b – c = d (Time Difference of arrival)
- Hence b = c + d, let c = 1.
- Applying Cosine Law

$$(diff + c)2 + c2 + 2ac(cos\varphi)$$

[9] Use of the crosspower-spectrum phase in acoustic event location, Omologo, M.; Svaizer, P. Speech and Audio Processing, IEEE Transactions on, Vol.5, Iss.3, May 1997, Pages:288-292

$$\varphi = \cos\text{-}1((a2 + c2 - (diff + c)2)/(2ac))$$

So once both angles are known we can calculate the distance of the sound source from the designated origin microphone as follows:

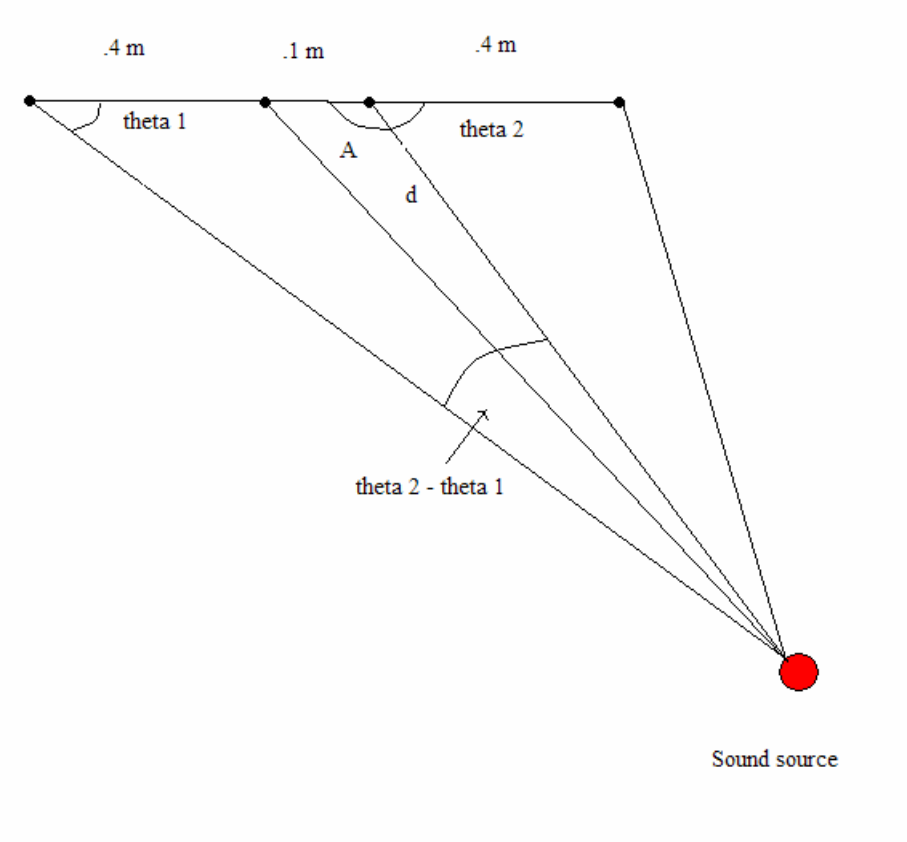

Figure 4: Geometry for Final Setup

Here, theta 1 and theta 2 have been calculated by the previous step. A is simply equal to 180 – theta2. We want to find the distance, d. This is accomplished by us using the law of sines, as follows:

$$D = .5 * \sin(\text{theta1}) / \sin(\text{theta2} - \text{theta1})$$

Therefore, by using the Cross Power Spectrum Phase to calculate the time difference of arrival and the using the trigonometry and geometry methods described above we were

able to effectively calculate the distance of the speaker location from designated origin microphone.

Algorithm Issues

The paper by M. Omologo and P. Svaizer claims that by using 2 mics pairs, a location error average of less than 10 cm can be achieved in a 6m * 6m area. As discussed above, a greater number of microphones implies better resolution. Due to hardware and budget constraints, we used 4 microphones and were able to get fairly accurate real time results.

Initially, we had planed to place the microphones in the shape of a square of sides of .4 m. However with the Cross Power Spectrum Phase giving angles between 0 and 180, this became a problem. For example, for inputs from both the angles 30 and 330, our algorithm output 30 degrees. Therefore, if the sound source is originating from anywhere between the two pairs of microphones in the square, the angle from one of the microphone pairs will be incorrect and this results in an incorrect location of the sound. For this reason, we changed our configuration so that both pairs are placed in a line and the distance between each pair is .1 m. Also due to hardware constraints and synchronization issues, the angles of only two pairs if microphones could be calculated, instead of calculating angles from multiple pairs and getting a more accurate answer. This will be discussed later in the Implementation and Synchronization sections.

The Cross Power Spectrum Phase[10] does a decent job getting rid of reverberations. However the quality of the signal and accuracy of TDOA estimates can be improved

[10] Microphone Arrays, Ward & Brandstein, Springer 2001

using Pre-Whitening by Linear Prediction.[11] Empirically we realized that reverberations were not a big problem.

Outside noise is also a problem because since our project is real time there are actually lots of times when we need to disregard external disturbances. We can use adaptive filtering or noise reduction techniques, but have ignored this problem, since our final results turned out to be pretty good. Also this problem was not tackled due to time constraints.

Implementation

Initial Implementation / Matlab

Prior to working with the EVM, we attempted an initial implementation of source location with just two microphones. By demonstrating the capabilities in Matlab, we were able to show the potential of the CPSP. The Matlab code for our midterm update can be viewed in Appendix A.

For the update we showed several different manners in which we could potentially have used the CPSP. One of the most important parts of the demonstration was our ability to show great degrees of accuracy by upsampling. After initially computing the CPSP, we generated an ideally upsampled version of the signal by zero-padding before taking the inverse FFT. This showed that simple ideal interpolation improved accuracy a great deal, as can be seen in Figure 5. A more detailed discussion on upsampling and accuracy as related to the EVM implementation can be found below.

[11] Microphone Arrays, Ward & Brandstein, Springer 2001

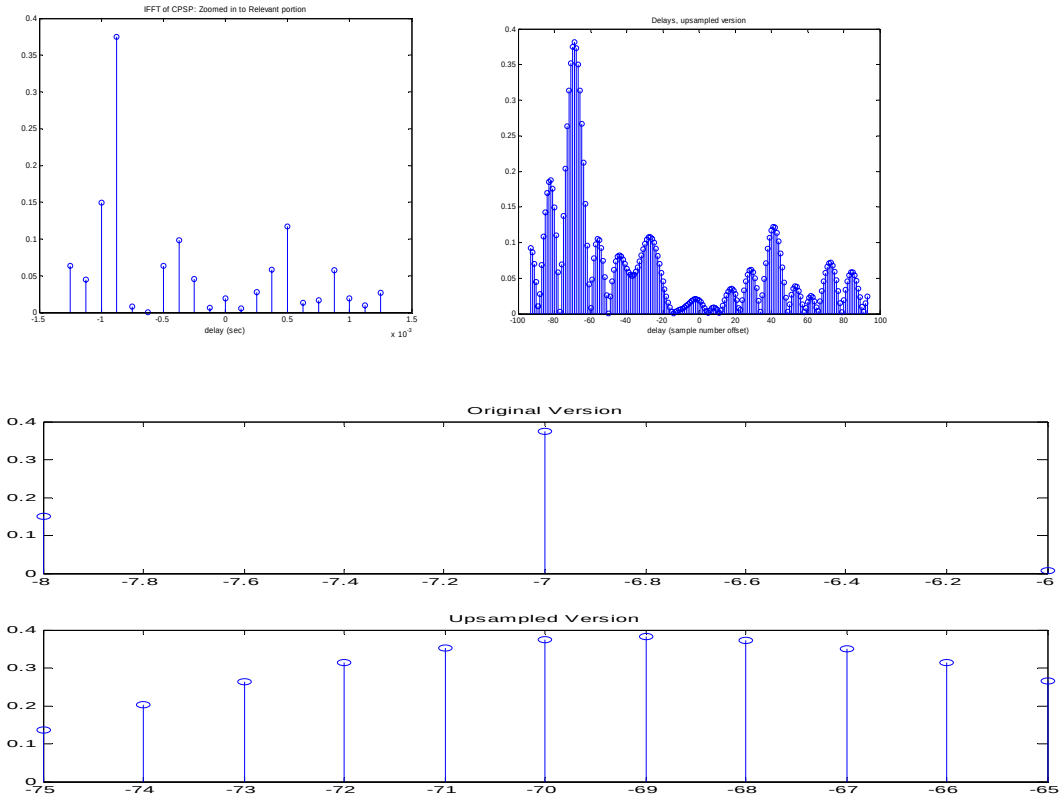

Figure 5: Simple Ideal Interpolation Improving Accuracy

Since the CPSP algorithm is based on finding a maximum peak corresponding to the primary angle of arrival, we realized that there was a potential to find multiple speakers by searching for local maxima. Figure 6 shows the output of the algorithm with two separate speakers; then by picking a certain noise floor threshold the local peaks were found. The two speakers were identified by zeroing out the noise below that threshold, and the results were promising. Unfortunately the major problem with locating several speakers was the identification of the threshold below which could be considered noise. In order to identify this number, we needed to have a stationary environment with many measurements in that environment; the 551 lab was anything but stationary due to the

variable number of people speaking and the music coming from the 545 side. Thus this feature was demonstrated in Matlab but we decided not to pursue it onto the EVM.

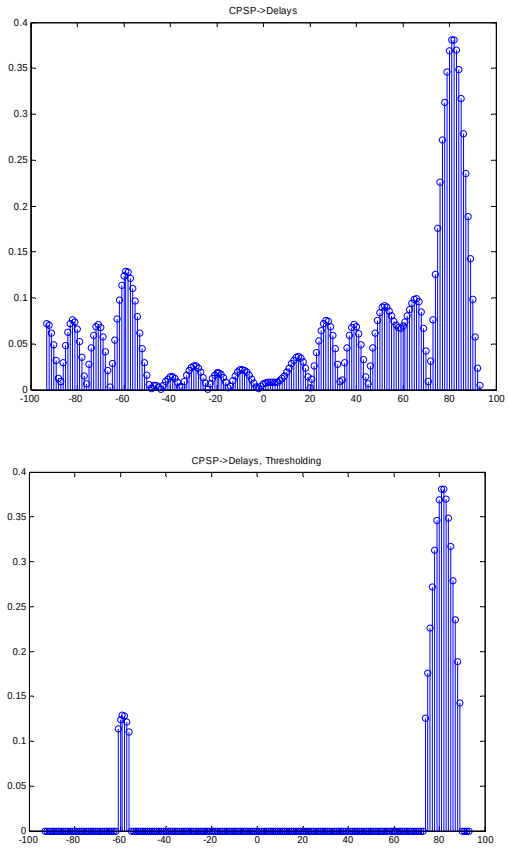

Figure 6: Two Separate Speakers

A final facet of our Matlab demonstration was showing the concept of tracking. By repeatedly recording and recalculating the angle from which a sound occurred, we essentially showed that tracking someone or something was extremely viable. A direct extension was made to our implementation on the EVM, so that our final demo involved tracking a speaker and continuously updating their position.

Data Flow

With four mics, and one EVM, we had to figure out how to be able to get all the data onto the EVM at the same time. The EVM only has a mic in and a line in to get data

from, so we needed to figure out how to get 4 mic inputs into the EVM that had two jacks for inputs. We found a set of coupled mono-mics that had one jack per pair of mono-mics that we could use to plug in directly to the EVM. This allowed us to have 4 inputs (mics) and still be able to plug it all into the EVM's two jacks (mic in / line in). However, we realized later, that the line in jack required more power on the input, so just plugging in the pair of microphones on that jack wouldn't suffice. We then needed to find an amplifier, but none could be found that fit our price range. Finally, we found a USB soundcard we could plug one pair of the mics into and then record input in a C++ program that would send data into the EVM.

Our final setup is as follows: two pairs of mono microphones each connected by a single jack, and a USB soundcard. One pair plugged into the USB soundcard which then plugged into the USB port on the PC. Another pair plugged in directly into the EVM's mic in port. A diagram of the data flow can be seen in Figure 7. We had the input that was directly plugged into the EVM transfer in by interrupts, while the input we had from the USB soundcard was sent to a C++ sound recording program. That program would then transfer the data to the EVM by a PCI transfer whenever one buffer of 4096 samples (0.5 seconds) worth of data was filled up. The EVM would receive these PCI transfers asynchronously through a callback function. Meanwhile, the C++ program will immediately await for a result from the transfer it just did. The EVM would do our calculations now from the input it obtained from the four mics and then send the two angles back via another PCI transfer to the C++ program. Once the C++ program received the results for a transfer, it would calculate the distance given theta1 and theta2, and display them on the GUI.

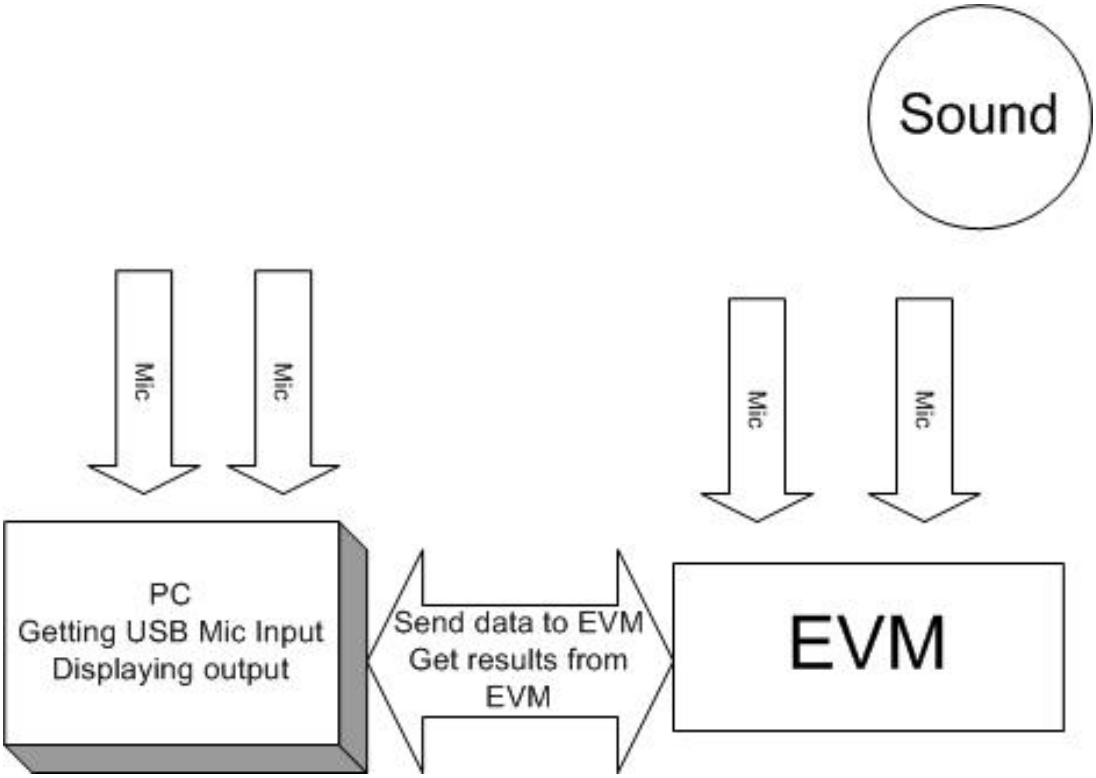

Figure 8: Diagram of Data Flow

Synchronization Issues

One of the major problems with any real-time system is the synchronization of the individual components, and our project was no different. Our setup was particularly challenging to resolve in this respect, as we had two inputs recorded directly to the EVM, and two recorded indirectly through the external sound card, the PC, and finally via asynchronous PCI transfer to the EVM. This problem was certainly not alleviated by our original intentions to align the recording of each buffer on the PC side with the each buffer on the EVM side such that we could calculate delays between all the pairs of microphones, instead of just between two pairs. Unfortunately such a perfect alignment was beyond the scope of this project.

The main deterring reason against this perfect alignment stems from the fact such an alignment needs to be extremely precise; even a very small time-delay calculation error translates to a large margin of error in the output accuracy.[12] The precision of this alignment is suspect to many variables, not the least of which includes human error in the training process. The relatively unreliable PC platform is also very difficult to control in such a precise manner as to be able to consistently schedule recordings precisely in concert with the EVM.

Thus we abandoned that notion and yet were still left with a different synchronization problem involving the data flow within the EVM and between the EVM and PC. In our first full implementations on the EVM, we noticed that a sort of race condition was occurring for controlling computational resources. The buffers coming from the PC were transferred asynchronously, and once they were completely transmitted a callback function called the necessary CPSP functions to compute an angle from the external input pair. Meanwhile the EVM was controlling its own pair of inputs, recording them into a buffer via interrupts and then finally trying to call the CPSP function when the buffer was full. However, we noticed that once one callback function completed, another PCI transfer had already completed and was waiting to use the processor for its own computations.

This resulted in a state in which the DSP performed CPSP calculations for the pair of inputs from the PC almost continuously, essentially ignoring the interrupts from the EVM. In fact, in retrospect we realized that this problem occurred because the PC was continuously recording into buffers, and then queuing up the buffers to be transferred

[12] Use of the crosspower-spectrum phase in acoustic event location, Omologo, M.; Svaizer, P. Speech and Audio Processing, IEEE Transactions on, Vol.5, Iss.3, May 1997, Pages 288-292

over to the EVM. Thus the buffers from the PC were able to maintain continuous control of the processor. On the other hand, the EVM filled up a single buffer, and waited for the CPSP calculations to occur before recording another buffer.

We were able to fix this "race" condition by simply delaying the transfers from the PC; the induced lag allowed the buffers coming into the EVM a chance to catch up. We found the appropriate delay in quite a manual and tedious way, by observing the sequence of events on the EVM and adjusting the delay such that the computational resources switched evenly between processes. Unfortunately this method had a somewhat adverse effect on our plans for optimization. As it turned out, the simultaneous control of the PCI transfer/callback and interrupt processes was not extremely robust to significant changes in the code. When we tried to increase the levels of optimization, the interplay between the processes changed drastically and our controlling method became obsolete. The extra effort necessary for redesigning this control would not have been offset by the added value of optimization - our code was already fast enough to produce real-time results with little noticeable delays. Thus we made an engineering decision to stick with lower levels of optimization.

When we really started taking measurements and testing out our algorithm, we found another slight, subtle, and seemingly inexplicable problem in the real-time updates. When a speaker started talking, the pair of microphones going into the EVM picked up the speaker almost instantly, but the pair of microphones going into the PC lagged behind by a couple of update cycles. This could not have been due to our induced lag time, since that only delayed the pertinent computation by a maximum of 1 update cycle. In fact we found, again in retrospect, that these delays were caused by the PC recording mechanism

which queued up the buffers waiting to be transferred. The new speaker would not really be “heard” by our system until several update cycles had occurred. Since this revelation dawned upon us after the fact, we were unable to fix the code on the PC side to forget the queue and instead just record and calculate sequentially as the EVM did. However we did find that for the application of tracking a speaker, the update lag was not very apparent (<1 sec) and it had minimal effects on the performance of the system.

Memory and Speed Issues

When first compiling our program, we had many memory issues trying to successfully compile the program and fit it on the DSP. We started out with the code from the Lab1 project, and added in the FFT code to do an FFT whenever the buffer would fill up during an interrupt. We had many errors with the complier complaining that the text region was truncated. This turned out now to be a “cannot fit on memory” issue, but rather an issue with the DSP unable to jump to instructions too far away in memory. After reallocating memory around, we finally were able to successfully compile the program and have the text region be on off chip program memory.

For our global buffers, we required lots of memory. We had three buffers each holding 4096 samples in complex float format to hold our input data from the two pairs of mics, along with having one buffer to serve as a temporary buffer to copy our data into when running our FFT and CPSP algorithm. We also required two more buffers of 4096 samples each in int format to hold our twiddle factors and bit reversal array to use when doing our FFT. Since this was all a hefty amount of memory, we put these in off chip memory using the far keyword and allocated far to be in SDRAM0.

We decided to go with buffers that were 4096 samples long each because that is what seemed to have worked best. Our CPSP algorithm would not work well with smaller sample sizes, as discussed in the next section on accuracy issues. We also could not increase our sample sizes because that would increase all our global buffer sizes by huge factors and would bring our program to a halt. We also could not up sample our buffers either since that too would significantly slow down our program.

Knowing that most of our data and program was in off chip memory, we also expected our program code to run slowly. In profiling our CPSP code that was used to calculate the angle, we saw that it took about 44 million CPU cycles each time the function was called. This is about 0.3 seconds since the DSP runs at 133 MHz. However though, this part of our code was not the bottleneck, as mentioned above. Since we received input from the PC every 0.5 seconds through PCI transfers, this 0.3 seconds could easily fit in between each successive transfer. The bottleneck though was synchronization issues we had in dealing with receiving data from two different types of sources, direct mic in input and PCI transfer. This synchronization issue is discussed in the prior section.

Variable	Size
`far int buffer[BUFFER_LEN];`	4096 * 4 = 16k
`far int buffer2[BUFFER_LEN];`	4096 * 4 = 16k
`far int buffer3[BUFFER_LEN];`	4096 * 4 = 16k
`far float x1[2*BUFFER_LEN];`	4096 * 4 * 2 = 32k
`far float x2[2*BUFFER_LEN];`	4096 * 4 * 2 = 32k
`far float xout[BUFFER_LEN];`	4096 * 4 = 16k
`far float tempbuf[2*BUFFER_LEN];`	4096 * 4 * 2 = 32k
`far int revtableup[BUFFER_LEN];`	4096 * 4 = 16k
`far float wup[2*BUFFER_LEN];`	4096 * 4 = 16k
Total Off Chip:	**224K**

Figure 8: Memory Usage calculations

Upsampling and Accuracy

With respect to accuracy, our initial intention was to at least replicate the results of Omologo et al[13] and get less than a 10cm error in a 6x6 meter area. We knew that to do this, would require an extremely high sampling rate so as to be able to resolve delays on a very fine scale. The simplest method to achieve such a high resolution was obviously upsampling; we had shown the potential of this method in our Matlab demo. Attempting to preplan for memory requirements, we presumed that upsampling a buffer of 0.1 seconds in length by a factor of 8 would allow for decent results. However we ran into several problems along the way.

The first problem, which took use a decent amount of time to debug, was that 1024 samples at an 8000Hz sampling rate was just not enough to capture a good signal and get reliable results. We were only able to get consistent results by using a minimum of 4096 samples, or approximately 0.5 seconds. Even with this significantly larger buffer, we tried implementing upsampling to try and get good results. However we found that upsampling by a small amount made almost no difference in the results, and using a factor larger than 4 caused significant memory problems (see memory section).

Thus we were left with our original resolution corresponding to 8kHz. Our geometry was such that the microphones within each pair were 40cm apart, corresponding to a maximum 11-sample delay between sound arriving at each microphone in the pair. Thus our resolution was only a maximum of 11 different angles over 90 degrees. As one might expect, this led to distinguishable errors in location calculations, as shown in Figure 9. The shaded region is the intersection of the range over which each microphone would

[13] Use of the crosspower-spectrum phase in acoustic event location, Omologo, M.; Svaizer, P. Speech and Audio Processing, IEEE Transactions on, Vol.5, Iss.3, May 1997, Pages 288-292

output the same angle. It is apparent then that if the speaker is anywhere within the shaded region, the same radial distance (from Mic 1) would be calculated, and with a small resolution this could cause fairly large errors.

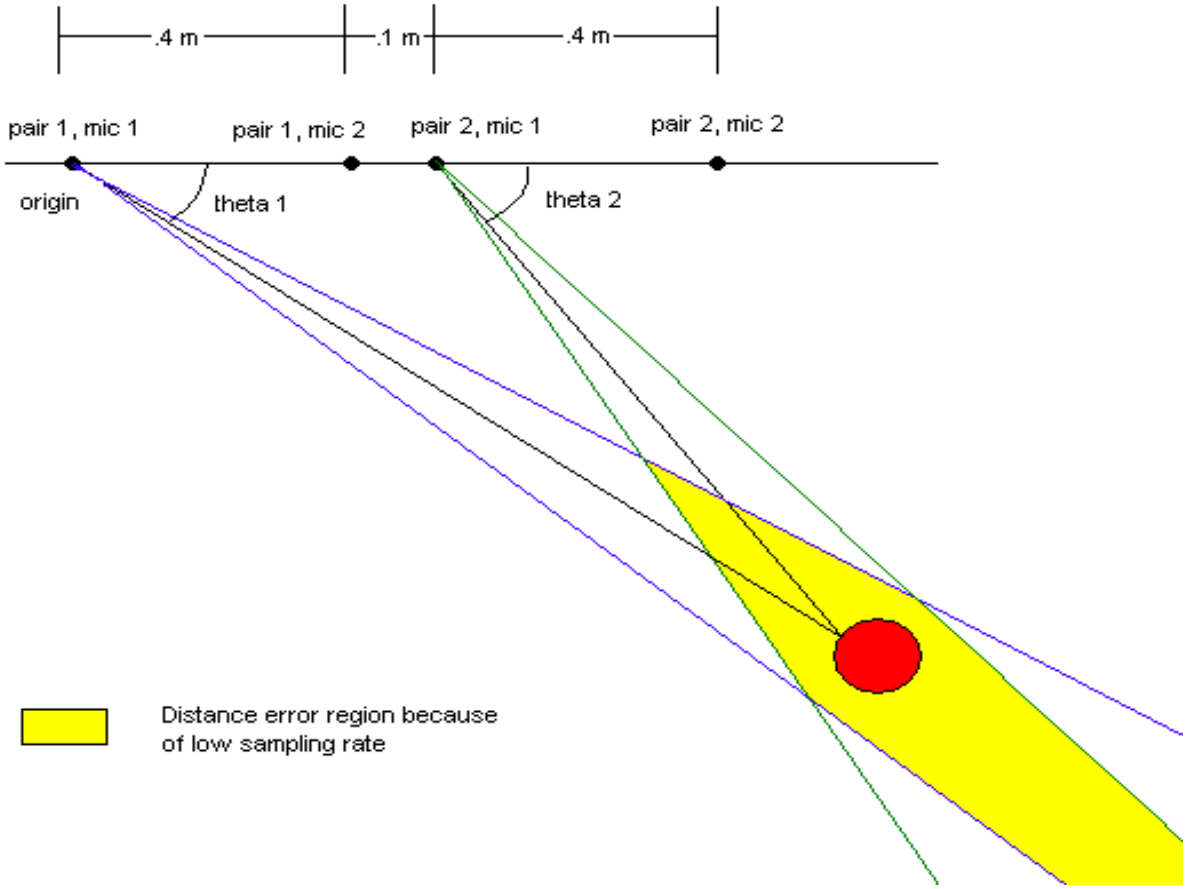

Figure 9: Demonstration of Error Region

Upon testing we realized that there would be singularities at the extremes, namely when the source was located at 0 or 180 degrees relative to the linear array. Radial distance can not be calculated for these values, and upon approaching these angles the distance calculation seemed to gradually worsen. This observation can be explained easily by looking at Figure 9; setting both angles near zero will cause the region of intersection to grow significantly. However we found that between radial distances of .4 and 2 meters, the algorithm produced fairly accurate results. So for the purpose of

tracking a speaker or identifying the location of a source of sound, our implementation seems fairly useful. More importantly, it is obviously scalable; with more memory or more microphones, the algorithm has the potential to be very accurate.

C++ Program

We obtained a Visual C++ program to record input on the PC[14]. This program gave us an interface to obtain our sound input from the two mics connected to the USB soundcard. We used this program as a basis for the C++ part of our project. Some modifications were made to fit our project such as adding the PCI transfer code to send/receive data to/from our EVM. We spent some time studying the program carefully so that we would know where our custom code would fit appropriately. After much analyzing, we found the place in the code that would process input buffers once they were full and record to a wav file. We removed the code to write to a wav file, and instead, we added code to transfer the filled buffer to the EVM. We also added code to wait and receive the results back from the EVM immediately after sending a filled buffer. The data we received consisted from two angle calculations, one for each pair of mics. The C++ program would then also do a simple calculation with the angles to obtain the distance of the sound source. We finally modified the GUI a bit to add a display to show the two angles and the calculated distance. The display would update each time we received results from the EVM.

[14] http://www.codeproject.com/audio/fister.asp?target=play%7C%2Brecord%7Csound
How to play and record sound
By Thomas Holme

Final Demo / Future Work

For the final demonstration, we were able to show our working algorithm in the context of solving our initial problem. We were successfully able to track in real time the position of a moving source of sound, specifically a person speaking, amidst the noisy conditions of the 551 lab. We outputted the two angles generated by each pair, along with the radial distance from the first microphone. Unfortunately due to the accuracy problem described above, had we calculated Cartesian coordinates, the results would have not been as obvious as the pair of angles. The GUI Interface displayed these results in real time, and can be seen in Figure 10.

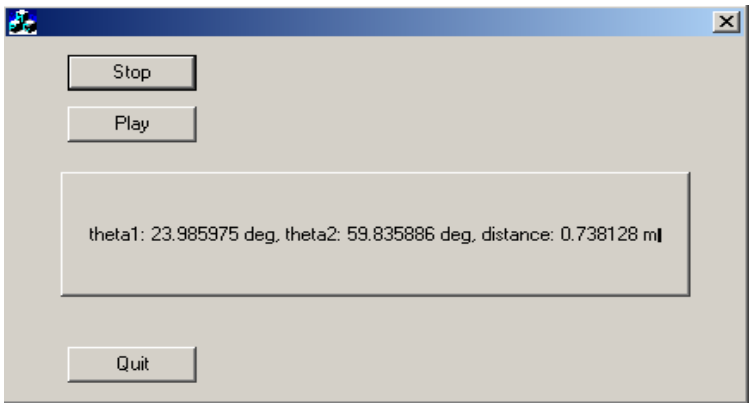

Figure 10: GUI for Demo

As described above, we were able to achieve fairly small errors, on the order of 10-20 cm, for radial distances between .4 and 2 meters from the first microphone. This is also assuming we stayed away from the boundary conditions as well. Thus we were able to effectively demonstrate our project; while held back by the many limitations described above we were still able to achieve very good results.

Future Work and Potential Improvements

In the future many of the issues above may be re-hashed to try and resolve them in some manner. As can be seen in our discussions above, memory issues were really the main problem which hampered us from achieving better results. We were unable to find a way to upsample a large enough buffer to provide accuracy on the order that we desired. Also, the synchronization issue may be dealt with in a way as to avoid the subtle problems which arose with our code. Noise canceling and multiple-speaker tracking could also potentially be implemented for applications with relatively stationary environments.

Finally, a good extension of this project would be to include more microphones. This would help on a lot of fronts, and would in fact make beamforming a distinct possibility. Most of the prior research[15][16] used relatively expensive equipment to mux the signals and port them into the EVM. Had we sufficient funding, we could have done the same and potentially implemented more sophisticated and robust applications of microphone arrays.

[15] Thomas M. Sullivan, Multi-Microphone Correlation-Based Processing for Robust Automatic Speech Recognition (2.2MB), Ph.D Thesis, ECE Department, CMU, August 1996.
[16] D. Rabinkin, R. Renomeron, A. Dahl, J. French, J. Flanagan, and M. Bianchi. A DSP Implementation of Source Location Using Microphone Arrays. Proceedings of the SPIE, 22(4):20 22, 1995.

Appendix A: MATLAB Code

Delay.m: Records sound and calculates angle from which speaker is speaking

```
% Find the delay and corresponding angle between two channels of a
recorded signal.
% Sampling rate and Length of recorded signal are specified.
clear all;
%Record Signal
d = .4;                                  %distance between mics, in
meters
C = 344.955;                             %speed of sound at room
temperature = 72Farenheit
Fs = 8000;                              %sampling rate
up = 10;
RecLength = 1;                           %how long to record in seconds
y = wavrecord(RecLength*Fs, Fs, 2);      %record wav; 2 channels

CH1   = y(:,1);       %pick off left and right
CH2   = y(:,2);

%Cross-Power Spectrum Phase
CPSP = (fft(CH1).*conj(fft(CH2))./( abs(fft(CH1)).*abs(fft(CH2)) );

CPSP = [up*CPSP; zeros((up-1)*Fs*RecLength,1)];
Fs = Fs*up;
delays = abs(fftshift(ifft(CPSP)));

%Get Delay
Maxdiff = ceil((d/C)*Fs);
zeroindex = Fs*RecLength/2 +1;
%stem(-Maxdiff:Maxdiff, delays((zeroindex-
Maxdiff):((zeroindex+Maxdiff))));
%title('Delays'); xlabel('sample number offset');
%figure; stem(-(Fs*RecLength)/2:((Fs*RecLength)/2-1), delays);
[maxvalue, maxindex] = max(delays((zeroindex-
Maxdiff):(zeroindex+Maxdiff)));

timedif = (maxindex-(Maxdiff+1)) / Fs;
%disp(sprintf('Delay between signals is %f sec: sample number %d',
timedif, maxindex))

Ddiff = C*timedif;
theta = acos((d^2+1-(Ddiff+1)^2)/(2*d));
disp(sprintf('Angle is %f degrees', theta*180/pi))
%prr = peaktofloor(delays);
%disp(sprintf('CPSP->Delays: Peak to Floor Ratio is %f db', prr));
```

Peaktofloor.m – calculates measure of how good the output will be.

```
function prr = peaktofloor(x)
```

```
[maxv, maxi] = max(x);
newx = [x(1:maxi-1); x(maxi+1:length(x))];;

prr = 20*log10(maxv/mean(newx));
```

Demo.m – Code for Midterm update demonstration

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Demo of CPSP
% 3/24/2003
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Preprocessing
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;

set(0,'Units','pixels')
scnsize = get(0,'ScreenSize');
pos1  = [scnsize(3)/3+2, -30, 2*scnsize(3)/3, scnsize(4)];
figure('Position', pos1);
disp('  ');
%Record Signal
d = .4;                                 %distance between mics, in
meters
C = 344.955;                            %speed of sound at room
temperature = 72Farenheit
Fs = 8000;                              %sampling rate
RecLength = 1;                          %how long to record in seconds
y = wavread('input.wav');               %input wav
soundsc(y);
CH1   = y(:,1);                         %arbitrarily pick left and
right
CH2   = y(:,2);

%View left and right channels
subplot(2,1,1), plot(CH1);
title('channel 1'); xlabel('sample');
subplot(2,1,2), plot(CH2);
title('channel 2'); xlabel('sample');
pause;

%Zoom in
subplot(2,1,1), stem(3070:3097, CH1(3070:3097));
title('channel 1 zoomed in'); xlabel('sample');
subplot(2,1,2), stem(3070:3097, CH2(3070:3097));
title('channel 2 zoomed in'); xlabel('sample');
pause;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Initial Implementation
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Cross-Power Spectrum Phase
CPSP = (fft(CH1).*conj(fft(CH2)))./( abs(fft(CH1)).*abs(fft(CH2)) );
delays = abs(fftshift(ifft(CPSP)));
subplot(1,1,1), stem(-(Fs*RecLength)/2:((Fs*RecLength)/2-1), delays);
```

```
title('IFFT of CPSP: Delays'); xlabel('delay (sample number offset)');
prr = peaktofloor(delays);
disp(sprintf('CPSP->Delays: Peak to Floor Ratio is %f db', prr));
pause;

%Get Delay
Maxdiff = ceil((d/C)*Fs);
zeroindex = Fs*RecLength/2+1;
stem([-Maxdiff:Maxdiff]./Fs, delays((zeroindex-
Maxdiff):((zeroindex+Maxdiff))));
title('IFFT of CPSP: Zoomed in to Relevant portion'); xlabel('delay
(sec)');

[maxvalue, maxindex] = max(delays((zeroindex-
Maxdiff):(zeroindex+Maxdiff)));
timedif = (maxindex-(Maxdiff+1)) / Fs;
disp('  ');
disp(sprintf('Delay between signals is %f sec: sample offset of %d',
timedif, maxindex-Maxdiff-1))
pause;
Ddiff = C*timedif;
theta = acos((d^2+1-(Ddiff+1)^2)/(2*d));
disp(sprintf('Angle is %f degrees', theta*180/pi))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Upsampling
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
pause;
up = 10;                                                %upsampling by 10

CPSP = [up*CPSP; zeros((up-1)*Fs*RecLength,1)];    %ideal upsampling:
interpolation by sin(x)/x
Fs = Fs*up;
delays2 = abs(fftshift(ifft(CPSP)));

%Get Delay
Maxdiff = ceil((d/C)*Fs);
zeroindex = Fs*RecLength/2 +1;
stem(-Maxdiff:Maxdiff, delays2((zeroindex-
Maxdiff):((zeroindex+Maxdiff))));
title('Delays, upsampled version'); xlabel('delay (sample number
offset)');
disp('  ');
disp('**Upsampled Version**');
pause;

[maxvalue, maxindex] = max(delays2((zeroindex-
Maxdiff):(zeroindex+Maxdiff)));
timedif = (maxindex-(Maxdiff+1)) / Fs;

disp(sprintf('Delay between signals is %f sec: sample offset of %d',
timedif, maxindex-Maxdiff-1))

Ddiff = C*timedif;
theta = acos((d^2+1-(Ddiff+1)^2)/(2*d));
```

```
disp(sprintf('Angle is %f degrees', theta*180/pi))
%Analyze difference
pause;
subplot(2,1,2); stem(-75:-65, delays2((40001-75):((40001-65))));
title('Upsampled Version');
subplot(2,1,1); stem(-8:-6, delays((4001-8):(4001-6)));
title('Original Version');

pause;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Multiple Speakers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
disp('  ');
disp('Multiple Speakers');
clear all;
%Record Signal
d = .4;                                 %distance between mics, in
meters
C = 344.955;                            %speed of sound at room
temperature = 72Farenheit
Fs = 8000;                              %sampling rate
up = 10;
RecLength = 1;                          %how long to record in seconds
y = wavread('input_mult.wav');          %record wav; 2 channels
soundsc(y);
CH1   = y(:,1);      %arbitrarily pick left and right
CH2   = y(:,2);       %should actually figure out which is which

%Cross-Power Spectrum Phase
CPSP = (fft(CH1).*conj(fft(CH2))./( abs(fft(CH1)).*abs(fft(CH2)) );

CPSP = [up*CPSP; zeros((up-1)*Fs*RecLength,1)];
Fs = Fs*up;
delays = abs(fftshift(ifft(CPSP)));

%Get Delay
Maxdiff = ceil((d/C)*Fs);
zeroindex = Fs*RecLength/2 +1;
subplot(1,1,1), stem(-Maxdiff:Maxdiff, delays((zeroindex-
Maxdiff):((zeroindex+Maxdiff))));
title('CPSP->Delays');
pause;
stem(-Maxdiff:Maxdiff, delays((zeroindex-
Maxdiff):((zeroindex+Maxdiff))) .* (  delays((zeroindex-
Maxdiff):((zeroindex+Maxdiff)))  > .1)  );
title('CPSP->Delays, Thresholding');

[maxvalue, maxindex] = max(delays((zeroindex-
Maxdiff):(zeroindex+Maxdiff)));
timedif = (maxindex-(Maxdiff+1)) / Fs;

Ddiff = C*timedif;
theta = acos((d^2+1-(Ddiff+1)^2)/(2*d));
disp(sprintf('Speaker 1: Angle is %f degrees', theta*180/pi))
```

```
[maxvalue, maxindex] = max(delays((zeroindex-Maxdiff):(zeroindex)));
timedif = (maxindex-(Maxdiff+1)) / Fs;

Ddiff = C*timedif;
theta = acos((d^2+1-(Ddiff+1)^2)/(2*d));
disp(sprintf('Speaker 2: Angle is %f degrees', theta*180/pi))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Demo it live
pause; disp('  ');
disp('Live Demo [press enter and record signal]');
pause; delay;

%%% Tracking
pause; disp('Tracking [press enter to begin, ctrl-c to quit]');
while(true) delay;
end
```

Appendix B: EVM Code

```
/*************************************************************
 * 18-551 Lab 1                                              *
 * echo.c                                                     *
 * A program to echo out on the output line whatever the EVM  *
 *  board gets on the the input.  We can choose to filter the *
 *  signal as well.                                           *
 * Author: 18551Spring2004Group1                              *
 *************************************************************/
#include <stdio.h>
#include <stdlib.h>

#include <mcbsp.h>              /* mcbsp devlib */
#include <common.h>
#include <mcbspdrv.h>           /* mcbsp driver */
#include <board.h>              /* EVM library */
#include <codec.h>              /* codec library */
#include <mathf.h>              /* math library */
#include <intr.h>               /* interrupt library */
#include <linkage.h>
#include <string.h>
#include <dspf_sp_bitrev_cplx.h>
#include <pci.h>

#define BUFFER_LEN 4096

int rindex=0;                   /* Index for receive ISR */
int xindex=0;                   /* Index for transmission ISR */
int done=0;
far int buffer[BUFFER_LEN];       /* Memory buffer for samples */
far int buffer2[BUFFER_LEN];      /* Buffer for mic in */
far int buffer3[BUFFER_LEN];      /* Buffer for PCI transfer */

far float theta1 = 0;
far float theta2 = 0;
float pos = 0;

int upf = 1;

far float x1[2*BUFFER_LEN*1/*up*/];      /* FFT input/output (complex
interleaved) */
far float x2[2*BUFFER_LEN];             /* FFT input/output (complex
interleaved) */
far float xout[BUFFER_LEN*1/*up*/];
far float tempbuf[BUFFER_LEN*2*1/*up*/];

/***** FFT global variables *****/
int revtable[BUFFER_LEN];     /* Look-up table for digit-reversing */
float w[2*BUFFER_LEN];        /* FFT twiddle factors (complex
interleaved) */
far int revtableup[BUFFER_LEN*1/*up*/];
far float wup[2*BUFFER_LEN*1/*up*/];
/***** done FFT global variables *****/
```

```
void cfftr4_dif(float* x, float* w, short n);

int request_transfer(void *buf, int size, int command);
int wait_transfer();


/*********************** FUNCTIONS ****************************/




/* This function creates the lookup table for digit reversing.  After
   it is run, revtable[n] equals the pairwise digit-reversal of n.
   n is the size of the FFT this table will be used for.*/
void mkrevtable(int n) {
  int bits, i, j, r, o;

  bits= (31 - _lmbd(1, n))/2;  /* _lmbd(1,n) finds leftmost 1 bit in n
*/
  for(i=0; i<n; i++) {
    r=0; o=i;
    _nassert(bits>=3);
    for(j=0; j<bits; j++) {
      r <<= 2;
      r |= o & 0x03;
      o >>= 2;
    }
    revtable[i] = r;
  }
}


/* Function for filling the table of FFT twiddle factors.  n is the
   size of the FFT to be used */
void fillwtable(int n) {
    int i;

    for (i=0; i<BUFFER_LEN; i++) {
        w[2*i] = cosf(((float)i/n)*2*PI);
        w[2*i+1] = sinf(((float)i/n)*2*PI);
    }

}

/* for FFT */
void mkrevtableup(int n) {
  int bits, i, j, r, o;

  bits= (31 - _lmbd(1, n))/2;  /* _lmbd(1,n) finds leftmost 1 bit in n
*/
  for(i=0; i<n; i++) {
    r=0; o=i;
    _nassert(bits>=3);
    for(j=0; j<bits; j++) {
```

```
      r <<= 2;
      r |= o & 0x03;
      o >>= 2;
    }
    revtableup[i] = r;
  }
}

/* for FFT */
void fillwtableup(int n) {
    int i;
    for (i=0; i<n; i++) {
        wup[2*i] = cosf(((float)i/n)*2*PI);
        wup[2*i+1] = sinf(((float)i/n)*2*PI);
    }
}

/* Main CPSP Algorithm */
int fftstuff(int typething) {
    int i=0, tempindex;
    float mag, temp1, temp2;
    float re1, im1, re2, im2;
    float max, Ddiff;
    int maxindex;


    buffer2[0] = buffer2[1];

    for(i=0; i<BUFFER_LEN; i++) {
        x1[i*2] = (float)abs((int)(buffer2[i] & 0xffff));
        if(x1[i*2] > 32000)
            x1[i*2] = x1[i*2]-65536;
        x1[i*2+1] = 0;
        x2[i*2] = (float)abs((int)((buffer2[i] & 0xffff0000)>>16));
        if(x2[i*2] > 32000)
            x2[i*2] = x2[i*2]-65536;
        x2[i*2+1]=0;
    }

    /*FFT*/
    cfftr4_dif(x1, w, BUFFER_LEN);
    cfftr4_dif(x2, w, BUFFER_LEN);

    /* Bit-reverse */
    memcpy(tempbuf, x1, 2*BUFFER_LEN*sizeof(float));
    for(i=0; i<BUFFER_LEN; i++) {
        tempindex = revtable[i];
        x1[2*i] = tempbuf[2*tempindex];
        x1[2*i+1] = tempbuf[2*tempindex+1];
    }

    memcpy(tempbuf, x2, 2*BUFFER_LEN*sizeof(float));
    for(i=0; i<BUFFER_LEN; i++) {
        tempindex = revtable[i];
        x2[2*i] = tempbuf[2*tempindex];
        x2[2*i+1] = tempbuf[2*tempindex+1];
```

```
}
for (i=2*BUFFER_LEN; i<upf*2*BUFFER_LEN; i++) {
    x1[i]=0;
}


/**CPSP**/
/*Conjugate x2*/
for(i=0; i<BUFFER_LEN; i++) {
    x2[2*i+1] = -x2[2*i+1];
}

/*Divide by magnitude*/
for(i=0; i<BUFFER_LEN; i++){
    temp1   = x1[2*i];
    temp2   = x1[2*i+1];
    mag     = sqrtf(temp1*temp1+temp2*temp2);
    x1[2*i]   = (temp1/mag);
    x1[2*i+1] = (temp2/mag);

    temp1   = x2[2*i];
    temp2   = x2[2*i+1];
    mag     = sqrtf(temp1*temp1+temp2*temp2);
    x2[2*i]   = (temp1/mag);
    x2[2*i+1] = (temp2/mag);
}

/*Multiply, save into x1*/
for(i=0; i<BUFFER_LEN; i++) {
    re1 = x1[2*i];
    im1                   = x1[2*i+1];
    re2                    = x2[2*i];
    im2                   = x2[2*i+1];

    x1[2*i]          = (re1*re2 - im1*im2);
    x1[2*i+1]        = (re1*im2 + im1*re2);
}

cfftr4_dif(x1, wup, ((short)upf*BUFFER_LEN)); //wupf

memcpy(tempbuf, x1, 2*sizeof(float)*BUFFER_LEN*upf);
for(i=0; i<BUFFER_LEN*upf; i++) {
    tempindex = revtableup[i];  //revupf
    x1[2*i] = tempbuf[2*tempindex];
    x1[2*i+1] = tempbuf[2*tempindex+1];
}

/*Bit-reverse*/
for(i=0; i<BUFFER_LEN*upf; i++) {
    xout[i] = fabsf(x1[2*i]);
}

/*Find max*/
max=0;
maxindex=0;
/* Search both sides of xout */
```

```
    for(i=0; i<upf*11; i++) {
        if(xout[i] > max) {
            max = xout[i];
            maxindex = i;
        }
    }
    Ddiff = 344.955*maxindex/(8000*upf);
    for(i=upf*BUFFER_LEN-upf*11; i<upf*BUFFER_LEN; i++) {
        if(xout[i] > max) {
            max = xout[i];
            maxindex = i;
            Ddiff = 344.955*(maxindex-upf*BUFFER_LEN)/(8000*upf);
        }
    }

    /*Get theta*/

    if (typething == 1) {
        theta1 = acosf(((1.16)-(Ddiff+1)*(Ddiff+1))/(.8));
    } else {
        theta2 = acosf(((1.16)-(Ddiff+1)*(Ddiff+1))/(.8));
        printf("theta1: %f deg, theta2: %f deg\n", 180*theta1/PI,
180*theta2/PI);
        if (pci_message_sync_send(1, FALSE) == ERROR) {
            printf("error sending sync send\n");
        }
        printf("sent message\n");
        pci_fifo_sync_send(0, (unsigned int *) &theta1, sizeof(float));
        pci_fifo_sync_send(0, (unsigned int *) &theta2, sizeof(float));

        printf("send sync\n");
    }

    return 1; /* Success */
}

/*************************************************************
 *    Name: rcvISR
 *  Inputs: none
 *  Output: none
 * Purpose: Interrupt vector to be called whenever a single
 *  sample of data is ready to be read.  For each sample, we
 *  simply store it in a buffer and increment the index into the
 *  buffer.
 *************************************************************/
interrupt void rcvISR(void) {
    /* MCBSP0_DRR is the hardware register where samples are stored */

    buffer[rindex++]=MCBSP0_DRR;
    /* circular buffer */
    if (rindex>=BUFFER_LEN ) {
        rindex=0;
        memcpy(buffer2, buffer, BUFFER_LEN*sizeof(int));
        if(fftstuff(1)!=666) {
            printf("fftstuff not completed properly");
        }
```

```
        /* Wait until it's done */
        done++;
    }
}


void done_receiving(int status) {
    memcpy(buffer2, buffer3, BUFFER_LEN*sizeof(int));
    fftstuff(2);
}

/************************************************************
 *    Name: main
 *  Inputs: none
 *  Output: none
 * Purpose:
 ************************************************************/
int main(void) {
    int i, j, k, dev1;
    Mcbsp_dev dev;               /* Serial port device */
    evm_init();                  /* Standard board initialization */

    mkrevtable(BUFFER_LEN);          /* Make the digit-reversal look-
up table */
    fillwtable(BUFFER_LEN);          /* Make the table of FFT twiddle
factors */
    mkrevtableup(upf*BUFFER_LEN);          /* Make the digit-reversal
look-up table */
    fillwtableup(upf*BUFFER_LEN);          /* Make the table of FFT
twiddle factors */
    done = 0;
    //DMA_AUXCR = 0x00000010;/* Set priority of HPI over CPU to avoid
crashing */

    printf("Program Running\n");
    printf("Configuring MCBSP...");
    mcbsp_drv_init();            /* Call this before using McBSP
functions */

    /* Open serial port */
    if (!(dev=mcbsp_open(0))) {
        return(ERROR);
    }
    /* Configure McBSP */
    mcbsp_setup(dev);   /* See bottom of this file */
    printf("done\n");

    /******************* configure CODEC *********************/
    printf("Configuring codec...");
    /* EXIT_ERROR is a macro which jumps to exit_err if the function
       returns an ERROR */
    EXIT_ERROR(codec_init());
    codec_change_sample_rate(8000, TRUE);
    /* A/D 0.0 dB gain, turn off 20dB mic gain, sel (L/R)LINE input */
    EXIT_ERROR(codec_adc_control(LEFT,0.0,TRUE,MIC_SEL));
    EXIT_ERROR(codec_adc_control(RIGHT,0.0,TRUE,MIC_SEL));
```

```
    /* mute (L/R)LINE input to mixer */
    EXIT_ERROR(codec_line_in_control(LEFT,MIN_AUX_LINE_GAIN,TRUE));
    EXIT_ERROR(codec_line_in_control(RIGHT,MIN_AUX_LINE_GAIN,TRUE));
    /* D/A 0.0 dB atten, do not mute DAC outputs */
    EXIT_ERROR(codec_dac_control(LEFT, 0.0, FALSE));
    EXIT_ERROR(codec_dac_control(RIGHT, 0.0, FALSE));
    printf("done\n");

    /*************** setup interrupt routines ******************/
    printf("Initializing interrupts...");
    intr_init();
    /* Hook up serial transmit interrupt to CPU Interrupt 14 */
    intr_map(CPU_INT14,ISN_XINT0);
    INTR_CLR_FLAG(CPU_INT14);     /* Clear any old interrupts */
    intr_hook(xmitISR,CPU_INT14); /* Hook our own xmitISR into chain
for 14 */
    /* Repeat the same process for the receive interrupt */
    intr_map(CPU_INT15,ISN_RINT0);
    INTR_CLR_FLAG(CPU_INT15);
    intr_hook(rcvISR,CPU_INT15);
    /* Enable all necessary interrupts */
    INTR_ENABLE(CPU_INT_NMI);     /* Non-maskable interrupt */
    INTR_ENABLE(CPU_INT14);
    INTR_ENABLE(CPU_INT15);
    INTR_GLOBAL_ENABLE();         /* Controls whether ANY interrupts
function */
    printf("done\n");

    /****************** Turn on the serial port
**********************/
    MCBSP_ENABLE(dev->port,MCBSP_RX|MCBSP_TX);

    /* At this point, the program leaves main and enters an infinite
     *  idle loop.  Interrupts continue to function */


    pci_driver_init();        /* Call before using any PCI code */

    dev1 = pci_fifo_open();    /* Open FIFO */

    if (pci_message_sync_send(1, TRUE) == ERROR) {
                printf("error sending sync send\n");
    }

    pci_fifo_async_receive(dev1, (unsigned int*)buffer3, BUFFER_LEN *
sizeof(int), done_receiving);

    while(1){
        for (i = 0; i < 500000; i++) {
        }
        pci_fifo_async_receive(dev1, (unsigned int*)buffer3, BUFFER_LEN
* sizeof(int), done_receiving);
    }


exit_err:
```

```
    return(ERROR);
}

/************************************************************
 *    Name: mcbspSetup
 *  Inputs: Mcbsp_dev
 *  Output: none
 * Purpose: McBSP stands for Multi-Channel Buffered Serial Port.
 *  It is build onto the C67 processor itself, and is how the
 *  codec communicates with the processor.  This function sets
 *  up the serial port for communication with the codec, and
 *  should never need to be modified.
 ************************************************************/
int mcbsp_setup(Mcbsp_dev dev) {
  /* Structure with all configuration parameters for serial port */
  Mcbsp_config mcbspConfig;
  memset(&mcbspConfig,0,sizeof(mcbspConfig)); /* Initialize everything
to 0 */

  mcbspConfig.loopback              = FALSE;
  mcbspConfig.tx.update             = TRUE;
  mcbspConfig.tx.clock_polarity     = CLKX_POL_RISING;
  mcbspConfig.tx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.tx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.tx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.tx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.tx.frame_length1      = 0;
  mcbspConfig.tx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.tx.frame_ignore       = FRAME_IGNORE;
  mcbspConfig.tx.data_delay         = DATA_DELAY0;

  mcbspConfig.rx.update             = TRUE;
  mcbspConfig.rx.clock_polarity     = CLKR_POL_FALLING;
  mcbspConfig.rx.frame_sync_polarity= FSYNC_POL_HIGH;
  mcbspConfig.rx.clock_mode         = CLK_MODE_EXT;
  mcbspConfig.rx.frame_sync_mode    = FSYNC_MODE_EXT;
  mcbspConfig.rx.phase_mode         = SINGLE_PHASE;
  mcbspConfig.rx.frame_length1      = 0;
  mcbspConfig.rx.word_length1       = WORD_LENGTH_32;
  mcbspConfig.rx.frame_ignore       = FRAME_IGNORE;
  mcbspConfig.rx.data_delay         = DATA_DELAY0;

  /* Pass entire structure to mcbsp_config, a library function which
   *  sets registers according to the contents of the structure */
  if(mcbsp_config(dev,&mcbspConfig) != OK) {
    printf("Couldn't configure McBSP device %i\n", dev);
    return(ERROR);
  }
  return(OK);
}


/* Use mailbox 1 for address, 2 for size, and 3 for command */
int request_transfer(void *buf, int size, int command)
{
    amcc_mailbox_write(2, size);
```

```
    amcc_mailbox_write(3, command);
    pci_message_sync_send((unsigned int)buf, FALSE);
    return(0);
}

/* The PC will send a message when the transfer is complete.  Wait
                          for that to happen */
int wait_transfer()
{
    unsigned int value;

    pci_message_sync_retrieve(&value);
    return(value);
}
```

Appendix C: C++ Code that was modified

```
//http://www.codeproject.com/audio/fister.asp?target=play%7C%2Brecord%7
Csound
// How to play and record sound
// By Thomas Holme
//
// Pipe.cpp: implementation of the CPipe class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include <stdio.h>
#include "fister.h"
#include "Pipe.h"
#include "evm6xdll.h"
#include <windows.h>
#include <math.h>


#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

#define PI 3.14159265

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CPipe::CPipe()
{
    hBd    = NULL;
    h_event;
    hHpi = NULL;

    unsigned int message;
    int i;
    char s_buffer[80];

    m_SoundIn.DataFromSoundIn = DataFromSoundIn;    // assign pointer
to callback function
    m_SoundIn.m_pOwner = this;
    m_SoundOut.GetDataToSoundOut = GetDataToSoundOut;  // assign
pointer to callback function
    m_SoundOut.m_pOwner = this;

    runOnce = 1;

/************** start evm board initialization *************/
    /* Open board */
    TRACE("starting initialization\n");
    hBd = evm6x_open(0, 0);
    TRACE("opened evm\n");
    if ( hBd == INVALID_HANDLE_VALUE ) {
```

```
        TRACE("Couldn't open board\n");
        exit(1);
    }

    /* Set the transfer timeout to 5 seconds */
    if(!evm6x_set_timeout(hBd, 5000)) {
        evm6x_close(hBd);
        TRACE("Couldn't set timeout\n");
        exit(1);
    }

    /* reset PCI FIFO */
    if((hHpi = evm6x_hpi_open(hBd)) == NULL) {
        TRACE("error opening hpi\n");
        exit(1);
    }
    evm6x_hpi_write_single(hHpi, 0x0e000000, 4, 0x0170003C);
    evm6x_hpi_close(hHpi);
    TRACE("Opened connection to board...\n");

    if(!evm6x_clear_message_event(hBd)) {
        evm6x_close(hBd);
        TRACE("Cannot clear message event\n");
        //getch();
        exit(1);
    }

    TRACE("cleared message\n");
    /* Setup the Windows event that comes when a message is received */
    sprintf( s_buffer, "%s%d",EVM6X_GLOBAL_MESSAGE_EVENT_BASE_NAME, 0);
    h_event = OpenEvent( SYNCHRONIZE, FALSE, s_buffer );
    if(h_event == NULL) {
        //evm6x_close(hBd);
        TRACE("\n!!!!!  h_event IS NULL  !!!!!\n");
        //getch();
        //exit(1);
    }

    TRACE("waiting for object\n");
    /* wait for event signaling a message from the DSP */
    WaitForSingleObject( h_event, INFINITE );

    /* we got a message, now read it */
    if(!evm6x_retrieve_message(hBd, (unsigned long*) &message)) {
        evm6x_close(hBd);
        //      TRACE("Error retrieving message from EVM\n");
        //getch();
        exit(1);
    }
    TRACE("Received sync message: %i.  Writing...\n", message);
    TRACE("done constructor\n");

/************************** end initialization********************/
}

CPipe::~CPipe()
```

```
{

}

void CPipe::DataFromSoundIn(CBuffer* buffer, void* Owner)
{
    ((CPipe*) Owner)->WriteSoundDataToFile(buffer);
}

void CPipe::WriteSoundDataToFile(CBuffer* buffer)
{
    unsigned int ulLength;
    float theta1=0;
    float theta2=0;
    float pos;
    char s_buffer[80];
    char buf1[256];
    char buf2[256];

    //if(m_pFile)
    //{
    //      if(!m_pFile->Write(buffer))
    //      {
    //              m_SoundIn.Stop();
    //              AfxMessageBox("Unable to write to file");
    //      }
    // write data to a 2nd file to make sure this is it


    //}
    if (m_testFile)
    {
        //if (runOnce) {
        //      for (int i = 0; i < buffer->ByteLen/2; i++)
        //      {
        //              TRACE("%d\n",*(buffer->ptr.s + i));
        //      }
        //}
        //m_testFile->Write(buffer);

        /*************  WRITING THE DATA VECTOR TO EVM
*****************/
        /* Set the length of the transfer */
        ulLength = buffer->ByteLen;
        /* Actually send the stuff.  Note the name is x_pc.  It MUST be
typecast
           to (unsigned int *), even though we're sending floats. The
data isn't
           changed */
        //if (runOnce) {
        //TRACE("writing sample to evm\n");
        if (!evm6x_write(hBd, (unsigned long*) buffer->ptr.s, (unsigned
long*) &ulLength)) {
            evm6x_close(hBd);
            TRACE("Couldn't write to EVM\n");
            //getch();
```

```
            exit(1);
        }
        //TRACE("done writing sample to evm\n");

        ulLength = sizeof(float);
        /* Setup the Windows event that comes when a message is
received */
        sprintf( s_buffer, "%s%d",EVM6X_GLOBAL_MESSAGE_EVENT_BASE_NAME,
0);
        h_event = OpenEvent( SYNCHRONIZE, FALSE, s_buffer );

        /* wait for event signaling a message from the DSP */
        WaitForSingleObject( h_event, INFINITE );

        /* we got a message, now read it */
        evm6x_retrieve_message(hBd, (unsigned long*) &theta1);
        //TRACE("Received sync message: %i.  Reading...\n", (unsigned
long) theta1);

        if (!evm6x_read(hBd, (unsigned long *) &theta1,(unsigned long*)
&ulLength)) {
            evm6x_close(hBd);
            TRACE("couldn't read theta1\n");
            exit(1);
        }
        if (!evm6x_read(hBd, (unsigned long *) &theta2,(unsigned long*)
&ulLength)) {
            evm6x_close(hBd);
            TRACE("couldn't read theta2\n");
            exit(1);
        }

        //TRACE("theta1 %f, theta2 %f\n", 180*theta1/PI,
180*theta2/PI);
        if (theta2!=theta1) {
            pos = .5*sinf(PI-theta2) / sinf(theta2-theta1); //.4 m
separation between pairs
            if(pos>0) {
                TRACE("theta1: %f deg, theta2: %f deg, distance: %f
m\n", 180*theta1/PI, 180*theta2/PI, pos);
                sprintf(buf1, "theta1: %f deg, theta2: %f deg,
distance: %f m\n", 180*theta1/PI, 180*theta2/PI, pos);
                UpdateDisplay(buf1);
                UpdateBackground(FALSE);
            } else {
                UpdateBackground(TRUE);
            }
        } else {
            if (theta1 > 1.35 && theta1 < 1.37) {
                UpdateBackground(TRUE);
            } else {
                TRACE("theta1: %f deg, theta2: %f deg\n",
180*theta1/PI, 180*theta2/PI);
                sprintf(buf1, "theta1: %f deg, theta2: %f deg\n",
180*theta1/PI, 180*theta2/PI);
                UpdateDisplay(buf1);
```

```
                UpdateBackground(FALSE);
            }
        }
        //}
        runOnce = 0;

    }

    /*
      TRACE("InTop: Full=%2d, Empty=%2d\n", m_FifoFull.GetCount(),
m_FifoEmpty.GetCount());

      // add to fifo buffer
      m_FifoFull.Add(buffer);
      buffer = (CBuffer*) m_FifoEmpty.Consume();

      TRACE("InEnd: Full=%2d, Empty=%2d\n", m_FifoFull.GetCount(),
m_FifoEmpty.GetCount());
    */
}

void CPipe::GetDataToSoundOut(CBuffer* buffer, void* Owner)
{
        ((CPipe*) Owner)->ReadSoundDataFromFile(buffer);
}

void CPipe::ReadSoundDataFromFile(CBuffer* buffer)
{
    if(m_pFile)
    {
        if(!m_pFile->Read(buffer))
        {
            // enf of file -> tell the GUI
            OnEndOfPlayingFile();
            delete m_pFile;
        }
    }

    if(m_testFile)
    {
        delete m_testFile;
    }
    /*
      TRACE("OutTop: Full=%2d, Empty=%2d\n", m_FifoFull.GetCount(),
m_FifoEmpty.GetCount());

      // consume from fifo buffer
      m_FifoEmpty.Add(buffer);
      buffer = (CBuffer*) m_FifoFull.Consume();

      TRACE("OutEnd: Full=%2d, Empty=%2d\n", m_FifoFull.GetCount(),
m_FifoEmpty.GetCount());
    */
}

void CPipe::StartRecordingToFile()
```

```
{
    m_pFile = new CSoundFile("sound1.wav", m_SoundIn.GetFormat());
    m_testFile = new CSoundFile("test.wav", m_SoundIn.GetFormat());
    if(m_pFile && m_pFile->IsOK())
        m_SoundIn.Start();
}

void CPipe::StopRecordingToFile()
{
    m_SoundIn.Stop();
    // close output file
    if(m_pFile)
        delete m_pFile;
}

void CPipe::StartPlayingFromFile()
{
    m_pFile = new CSoundFile("sound1.wav");
    if(m_pFile && m_pFile->IsOK())
        m_SoundOut.Start(m_pFile->GetFormat());
    else
        ErrorMsg("Unable to open file");
}

void CPipe::StopPlayingFromFile()
{
    m_SoundOut.Stop();
    // close output file
    if(m_pFile)
        delete m_pFile;
    if(m_testFile)
        delete m_testFile;
}

void CPipe::OnEndOfPlayingFile()
{
    // implement this function in the GUI to change things when EOF is
reached
}

void CPipe::UpdateDisplay(CString str)
{

}

void CPipe::UpdateBackground(bool b)
{

}
```

```
// SoundBase.cpp: implementation of the CSoundBase class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "SoundBase.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CSoundBase::CSoundBase()
{
    m_Format.wFormatTag    = WAVE_FORMAT_PCM;
    m_Format.cbSize              = 0;
    m_BufferSize    = 4096;  // samples per callback
    SetBitsPerSample(16);
    SetSamplesPerSecond(8000);
    SetNumberOfChannels(2);
}

CSoundBase::~CSoundBase()
{

}

void CSoundBase::SetBitsPerSample(int bps)
{
    m_Format.wBitsPerSample = bps;
    Update();
}

int CSoundBase::GetBitsPerSample()
{
    return m_Format.wBitsPerSample;
}

void CSoundBase::SetSamplesPerSecond(int sps)
{
    m_Format.nSamplesPerSec = sps;
    Update();
}

int CSoundBase::GetSamplesPerSecond()
{
    return m_Format.nSamplesPerSec;
}

void CSoundBase::SetNumberOfChannels(int nchan)
{
```

```
    m_Format.nChannels = nchan;
    Update();
}

int CSoundBase::GetNumberOfChannels()
{
    return m_Format.nChannels;
}

void CSoundBase::Update()
{
    m_Format.nAvgBytesPerSec =
m_Format.nSamplesPerSec*(m_Format.wBitsPerSample/8);
    m_Format.nBlockAlign     = m_Format.nChannels
*(m_Format.wBitsPerSample/8);
}

void CSoundBase::SetBufferSize(int NumberOfSamples)
{
    m_BufferSize = NumberOfSamples;
}

int CSoundBase::GetBufferSize()
{
    return m_BufferSize;
}

WAVEFORMATEX* CSoundBase::GetFormat()
{
    return &m_Format;
}
```

```
// fisterDlg.cpp : implementation file
//

#include "stdafx.h"
#include "fister.h"
#include "fisterDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif


/////////////////////////////////////////////////////////////////////////////
//////
// CFisterDlg dialog

CFisterDlg::CFisterDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CFisterDlg::IDD, pParent)
{
    //{{AFX_DATA_INIT(CFisterDlg)
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in
Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CFisterDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CFisterDlg)
    DDX_Control(pDX, IDC_BUTTON1, m_Distance);
    DDX_Control(pDX, IDC_PLAY, m_PlayButton);
    DDX_Control(pDX, IDC_RECORD, m_RecButton);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CFisterDlg, CDialog)
    //{{AFX_MSG_MAP(CFisterDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_RECORD, OnRecord)
    ON_BN_CLICKED(IDC_PLAY, OnPlay)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
//////
// CFisterDlg message handlers

    BOOL CFisterDlg::OnInitDialog()
{
    CDialog::OnInitDialog();
```

```
    // Set the icon for this dialog.  The framework does this
automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);             // Set big icon
    SetIcon(m_hIcon, FALSE);        // Set small icon

    return TRUE;  // return TRUE  unless you set the focus to a control
}

// If you add a minimize button to your dialog, you will need the code
below
//  to draw the icon.  For MFC applications using the document/view
model,
//  this is automatically done for you by the framework.

void CFisterDlg::OnPaint()
{
    if (IsIconic())
    {
        CPaintDC dc(this); // device context for painting

        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }
}

// The system calls this to obtain the cursor to display while the user
drags
//  the minimized window.
HCURSOR CFisterDlg::OnQueryDragIcon()
{
    return (HCURSOR) m_hIcon;
}

void CFisterDlg::OnRecord()
{
    CString string;
    m_RecButton.GetWindowText(string);
    if(string == "Record")
    {
        StartRecordingToFile();
        m_RecButton.SetWindowText("Stop");
```

```
    }
    else
    {
        StopRecordingToFile();
        m_RecButton.SetWindowText("Record");
    }

    m_Distance.SetWindowText("dude");
}

void CFisterDlg::OnPlay()
{
    CString string;
    m_PlayButton.GetWindowText(string);
    if(string == "Play")
    {
        StartPlayingFromFile();
        m_PlayButton.SetWindowText("Stop");
    }
    else
    {
        StopPlayingFromFile();
        m_PlayButton.SetWindowText("Play");
    }
}

void CFisterDlg::OnEndOfPlayingFile()
{
    m_PlayButton.SetWindowText("Play");
}

void CFisterDlg::UpdateDisplay(CString str)
{
    m_Distance.SetWindowText(str);
}

void CFisterDlg::UpdateBackground(bool b)
{
    m_Distance.SetState(b);
}
```

Appendix D: Purchases

Equipment Purchased	Amount	Vendor Name	Price
Miniature Binaural Mic	2	Microphones.com	$55.95
iMic – USB External Soundcard	1	Buy.com	$40.00