ABSTRACT:

Notably, sound and sight are considered diametrically incompatible. That is to say sound cannot be seen and an image cannot be heard. The intent of this project is to so change this perception. Only one direction is under test in this endeavor; the operation to turn sound into an image.

GOAL:

Take a sample of sound digitally; perform a series of manipulations to transform the data obtained into a visual format. The purpose is to perform a visual comparison between a pure sample of sound and that of a random sample.

METHOD:

- Utilize code constructed on the Texas Instruments OMAP L-137 Development Board to retrieve or create a sample of sound.
- Have the board perform filtering techniques to separate the sample into three ranges of frequencies.
- Have the board execute mathematical manipulations to format the data to fit into an image control.
- Send the image data via the serial RS-232 connection to a computer, on which is a serial port receiver.
- The serial port receiver, coded in Visual Basic, retrieves the image data and processes it into 5 image controls.
- The image controls show red, green and blue content individually, as well as showcasing a conglomerate image with and without an alpha value.

CONCEPT:

Sound data when digitally stored, specifically, for the development board, is a discrete value ranging from -32767 to 32767. Image data, when digitally stored is comprised of 3 channels (red, green, and blue) of integers ranging from 0 to 255. These channels placed together form a unit called a pixel. There is additionally a fourth optional channel of opacity or alpha value; also ranging from 0 to 255. To convert the sound into data it is required to develop a scheme to decide what information in sound defines what information in sight.

For the purpose of this project the following guidelines were established to create the development process for the conversion. Firstly, the sound will be separated in to three ranges of frequencies to correspond to the three channels of color in an image. The frequency ranges chosen are to allow a maximum amount of data in each section, given a random sample of audio. The purpose in this is to prevent biasing the resultant image against high or low frequencies. The bias of any given sample is expected to be seen, however. The ranges of the frequencies and corresponding color channels are set forth in Table 1, below.

Range Title	Frequency Range (Hz)	Image Channel
Bass	< 320	Red
Mid	320 to 1280	Green
Treble	> 1280	Blue

Table 1: Ranges of frequencies as compared to the image channels

Secondly, the sound data must be altered in value: to reduce its magnitude and change its sign, where negative. The data must be reduced to allow for storage in the smaller color container. The color channel range is exactly $1/256^{th}$ of the sound's amplitude range. Three formats will be used to change the sound data.

1. The absolute value of the sound magnitude will be divided by 256. This will result in a range of values from 0 to 127.
2. The absolute value of the sound magnitude will be divided by 256. This will result in a range of values from 0 to 255.
3. The sound value will have 32768 added to it and then will be divided by 256. This will result in a range of values from 0 to 255.

Thirdly, for a value must be determined for the opacity of the image. The idea behind this value when implemented is to maintain a sense of how the sound changes over time. Two calculations are used in construction of this channel. The first is an average of the values of the corresponding red, green and blue changes. This concept is to provide a smoothing effect wherein a range of frequencies was not as present. The second is similar to a compounding average; however the sum total of components is not used in the calculation. Instead each sample's color values are averaged and this value is added to the previous alpha value and divided by two. The values of alpha this way slowly scale across the entire sample, as to minimize the effect of sharp changes in value.

Finally, how all the values obtained thus far will be implemented into an image must be determined. A simple approach may be to place the values in linearly. The issue with this is that an image is a series of columns and rows made of pixels, and data placed line after line will not provide an image that adequately represents the sound, from which it was converted. The approach instead is to develop an algorithm for starting the center of the image and spiraling outward from there.

Overall, the concept is to produce two images, one of a sample of a pure sound; sine wave. And the other will be from a random sample of sound. The sine wave is expected to make a smooth looking image as the wave progresses. The random sound image should compare to the sine's image as much as it is smooth in overall frequency content.

IMPLEMENTATION:

The project was developed in two parts. The first was created in Code Composer Studio for usage on the Texas Instruments OMAP L-137 Development Board. The second was made in Visual Studio 2010. The language used was Visual Basic. As discussed above, the development board code retrieves or creates a sample of audio. This sample is filtered, processed and sent via a serial cable to the computer. On the PC, the program created in VS2010 receives the data, now formatted for an image. The data is constructed into pixels and thereby into an image, which is shown on screen.

The following sections break down the code developed for this project.

Texas Instruments OMAP L-137 Development Board Code[1]:

The code for the development board is separated into stages as the information undergoes its transformation. Primarily, the code consists of the following: audio retrieval, frequency filtering, mathematical manipulation and transferring. All stages except the final also include storage.

Information Storage[2]:

The total storage for this project exceeds that of the internal memory (256KB) and therefore it was determined to place all data on the SDRAM (64MB). The program links a command phrase to allow the storage to the SDRAM; show in Figure 1.

```
SECTIONS
{
        mySDRAM :> SDRAM
}
```

Figure 1: myLink.cmd File allows the value “mySDRAM” to be interpreted as SDRAM.

This code is maintained independently in file, myLink.cmd. The purpose of this linking phrase is to allow the complier to interpret “mySDRAM” as storage to the SDRAM. This is used in conjunction with a #PRAGMA tag and DATA_SECTION () to place some the required buffers for this project on the SDRAM. The implementation is shown in Figure 2.

```
#pragma DATA_SECTION(rec_buffer, "mySDRAM");
short rec_buffer[SD_BUF_SIZE];

#pragma DATA_SECTION(bass_filt_buffer, "mySDRAM");
short bass_filt_buffer[SD_BUF_SIZE];
#pragma DATA_SECTION(mid_filt_buffer, "mySDRAM");
short mid_filt_buffer[SD_BUF_SIZE];
#pragma DATA_SECTION(treble_filt_buffer, "mySDRAM");
short treble_filt_buffer[SD_BUF_SIZE];

#pragma DATA_SECTION(red_buffer, "mySDRAM");
unsigned char red_buffer[SD_BUF_SIZE]; //bass
#pragma DATA_SECTION(green_buffer, "mySDRAM");
unsigned char green_buffer[SD_BUF_SIZE]; //mid
#pragma DATA_SECTION(blue_buffer, "mySDRAM");
unsigned char blue_buffer[SD_BUF_SIZE]; //treble
#pragma DATA_SECTION(alpha_buffer, "mySDRAM");
unsigned char alpha_buffer[SD_BUF_SIZE]; //compounding average

#pragma DATA_SECTION(filt_buffer, "mySDRAM");
unsigned char filt_buffer[SD_BUF_SIZE];
```

Figure 2: Code from audioSample_io.c that shows storage for the majority of the buffers used in this project

Sound Retrieval[3]:

The board uses a Digital Signal Processor (DSP) to retrieve an analog audio sample and convert it into the digital values discussed above. The frequency for digital conversion is 44100 samples per sec, or Hertz. The data is stored in two input buffers. One buffer is used to store incoming information, while the other is being read from. As long as the reading and processing does not take longer than the storage phase no information is lost. To help assist in this end, this project records an audio sample prior to processing it. The large amount time required performing the filtering, conversion and transmission make it improbable to complete this project in real-time. The code utilized to perform the tasks as described is not under dissection in the lab report. The main focus is the process integral to the image conversion.

In figure 3, the code used in the Audio_Task function of the audioSample_io.c file is shown. This is the code used to obtain a sample of audio. The code is divided into three sections. The first and last sections work with the aforementioned parts of code that communicate whether the DSP or user have control of the input and output buffers. The first claims the buffers for the code and the latter release the buffers back to the DSP's control. The code in the center is part of a Switch statement and a nested 'IF' block. The code is repeated as necessary to fill the rec_buffer array. The size of which is determined by the operator. For this experiment a size 32 input buffers was used. The size is 65536 samples or approximately 0.75 seconds of audio. The reasons behind this size will be discussed later. When the required

amount of samples is obtained the 'state' variable is changed to 1, allowing the code to progress through the nested 'IF' block.

```
while(1)
{
        // reclaim full input buffer and empty output buffer from streams
        SIO_reclaim(inStream, (Ptr *)&pIn, NULL);
        SIO_reclaim(outStream, (Ptr *)&pOut, NULL);
……..

if (sample_count == 0)
        printf ("Recording Audio. Please wait...\n");

memset(pOut, 0, BUFSIZE);
memcpy(&rec_buffer[sample_count * BUFLEN * NUM_BUFS], pIn, BUFSIZE);
sample_count++;

if (sample_count == BUF_NUMS)
{
        printf("Done recording\n");
        sample_count = 0;
        state = 1;
}

…….
        //issue used input buffer and full output buffer to streams
        SIO_issue(outStream, pOut, BUFSIZE, NULL);
        SIO_issue(inStream,  pIn,  BUFSIZE, NULL);
}
```

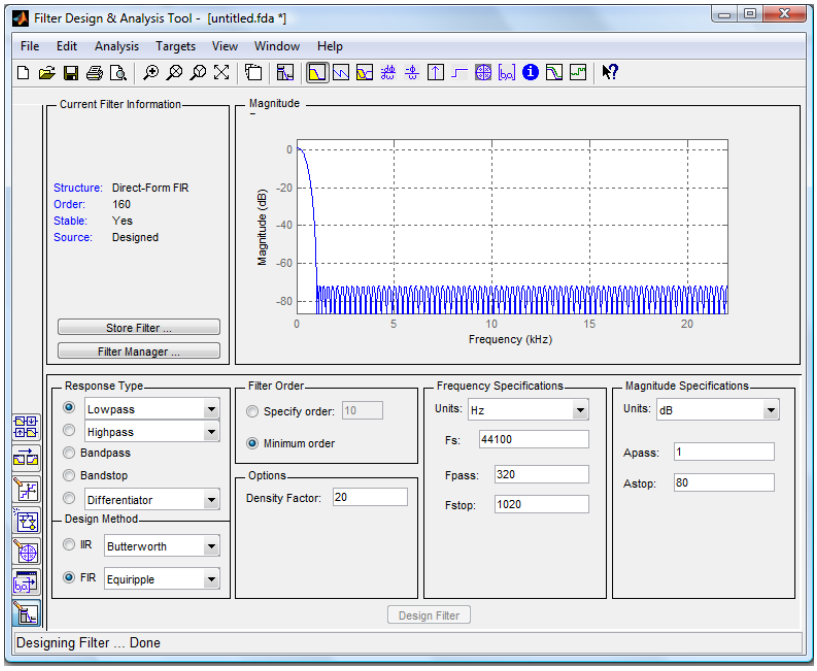
Figure 3: Code from audioSample_io.c that shows the main while loop for the audio task; the higher and lower parts are used to retrieve and release the input and output buffers for use by the code and then by the DSP.

Filtering the sample[4, 5]:

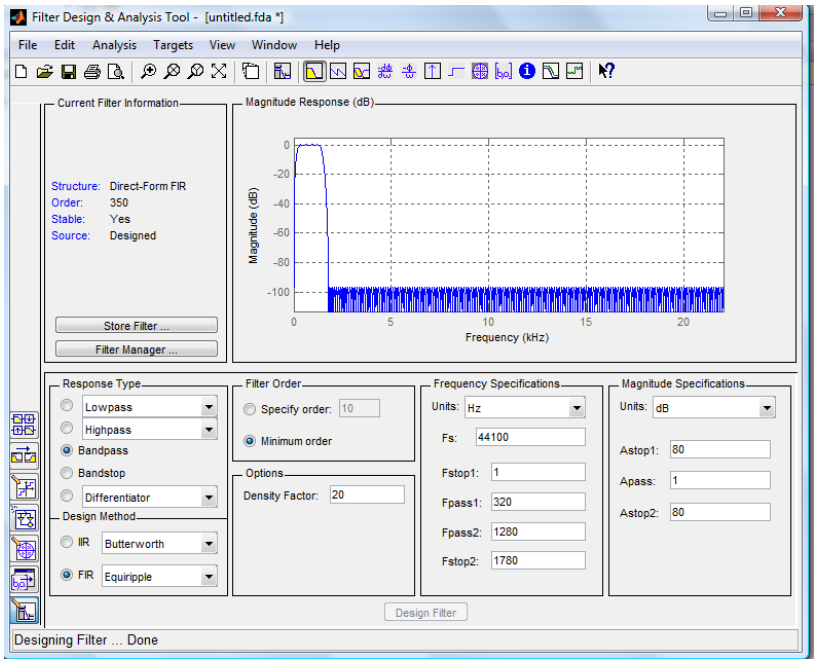
This project requires that the audio sample be separated into three distinct ranges of frequency. The ranges are bass (< 320 Hz), mid (320 to 1280 Hz) and treble (> 1280 Hz). To create these ranges, the MATLAB Filter Design and Analysis Tool (fdatool) was used. The bass filter was constructed of a low pass filter with an order of 160. Screenshot 1 gives all the details of the bass filters construction. Screenshot 2 is the fdatool display for the mid range filter design. And finally, the treble filter design is shown in Screenshot 3. All are displayed on the following page.

After each filter was designed, an M-file was generated that held the information on how to construct the filter. Using a supplied[4] file, the M-file for each was used to construct a

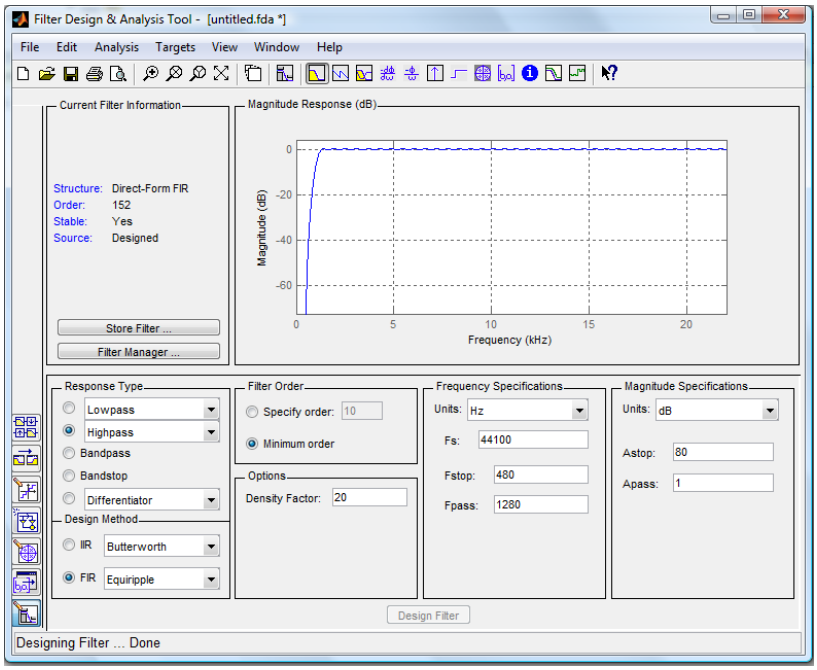
header file containing all the coefficients of the filter. These header files were included in the Code Composer project.



Screenshot 1: Bass filter design using the fdatool in MATLAB. The order (# of coefficients) is 160.



Screenshot 2: Mid filter design using the fdatool in MATLAB. The order (# of coefficients) is 350.



Screenshot 3: Treble filter design using the fdatool in MATLAB. The order (# of coefficients) is 152.

In the program for the DSP, the filters are included as header files containing the size and all the coefficients of the filter. The usage of each filter included performing a set of two nested 'for' loops. The internal 'for' loop cycles through the current sample and subsequent samples multiplying each by successive filter coefficients. The summation of this 'for' loop is the resulting value for that current sample. The rest of the entire recorded audio buffer is looped through. For the purpose of this project the resulting value's absolute value was stored at this point. Figure 4 shows the portion of the Audio_Task function that performs the filtering then changes the 'state' variable to allow the program to continue. Figure 5 is the filtering function, wherein the dual nested loops perform the mathematical operation and store the result.

```
if(state == 1)
{
        printf ("Performing Bass Filtering. Please wait...\n");
        FIR_Filter ( rec_buffer, bass_filt_buffer, bass_filt, BFILTER_SIZE );
        printf ("Performing Mid Filtering. Please wait...\n");
        FIR_Filter ( rec_buffer, mid_filt_buffer, mid_filt, MFILTER_SIZE );
        printf ("Performing Treble Filtering. Please wait...\n");
        FIR_Filter ( rec_buffer, treble_filt_buffer, treble_filt, TFILTER_SIZE );
        state = 2;
}
```

Figure 4: Code from audioSample_io.c that shows the portion of the main while loop when the filtering function is called to create the bass, mid and treble ranges.

```
Void FIR_Filter ( short *sdBuf, short *fsdBuf, const float *coeff, int filt_size )
{
	int i, k;
	double sum = 0;

	for (i=0; i < (SD_BUF_SIZE / NUM_BUFS); i++)
	{
		sum = 0;

		for ( k=0; k < filt_size; k++  )
		{
			sum += coeff[k] * sdBuf[2*(i-k)];
		}

		fsdBuf[i*2] = abs(sum);
		fsdBuf[i*2 + 1] = abs(sum);

		// can be without absolute value, add 32768
		// fsdBuf[i*2] = (sum + 32768);
		// fsdBuf[i*2 + 1] = (sum + 32768);

	}
}
```

Figure 5: Code from audioSample_io.c: function for filtering; receives the rec_buffer, an output filtered buffer, the coefficients for filtering, and a filter size.

Performing Numerical Conversion:

At this point the values in each sample range from 0 to 32767. Or when the result is not the absolute value, the range is -32767 to 32767. These values need to be reduced to fit into the integer size expected for a color scheme. This process was attempted in three ways. The first two are essentially the same; simply different size divisors. The first way divided each sample by 128. This corresponds to a resultant range of 0 to 255. The second divided by 256 resulting in a range of 0 to 128. The final way maintained the original negative values, but added 32768 to each sample. This process would adjust each value to more positive ensure that the lowest value was still positive and the greatest value was the top of the range. Once again each of these samples was divided by 256 returning a range of 0 to 256. The code to perform these tasks is shown in Figure 6; the alternative coding is shown but commented out to prevent operation. Upon completion the 'state' variable is again changed to allow the process to continue to the next operation.

```
if(state == 2)
{
        printf ("Performing Numerical Conversion. Please wait...\n");
        int k;
        for(k = 0; k < SD_BUF_SIZE; k++)
        {
                //absolute value of filter output divided by 128
                red_buffer[k] = bass_filt_buffer[k] / 128;
                green_buffer[k] = mid_filt_buffer[k] / 128;
                blue_buffer[k] = treble_filt_buffer[k] / 128;
                alpha_buffer[k] = ((( red_buffer[k] + green_buffer[k] + blue_buffer[k] ) / 3)
                    + (alpha_buffer[k-1])) / 2;

                // absolute value of filter output divided by 256
                /*red_buffer[k] = rec_buffer[k] / 256;
                green_buffer[k] = rec_buffer[k] / 256;
                blue_buffer[k] = rec_buffer[k] / 256;
                alpha_buffer[k] = rec_buffer[k] / 256;*/

                // add 32768 to each sample to make positive
                //red_buffer[k] = bass_filt_buffer[k] / 256;
                //green_buffer[k] = mid_filt_buffer[k] / 256;
                //blue_buffer[k] = treble_filt_buffer[k] / 256;
        }
        state = 3;
}
```

Figure 6: Code from audioSample_io.c: this is the portion of code to provide numerical conversion from a short sized variable to the range of an unsigned integer.

Construction of Sound[6]:

To provide a comparison from random sound results to anything, a pure sound or harmonic wave was created. A sine wave is used as the basis for what a smooth sample would appear as. Figure 7 showcases how this data is created.

The program creates a sine wave by iterating over the period of the wave. This is defined by dividing the sampling frequency by the fundamental frequency; specifically shown is a 1 kHz sine wave. The sine wave is then repeated until the storage buffer for each of the colors is filled. The process of shifting the sine wave into the positive (adding 32768) is shown. The value is then divided by 256 prior to being stored. The alpha value is developed as discussed before. This information is not filtered; only created, stored, and transmitted.

```
void Sine ()
{
	int i;
	unsigned short value;
	static short ctr = 0;

	for (i=0; i <= 16384; i++)
	{
		value = (32767*sin((2*PI*1000.0*ctr++)/44100)) + 32768;
		red_buffer[i] = (value / 256);
		green_buffer[i] = (value / 256);
		blue_buffer[i] = (value / 256);
		alpha_buffer [i*2] = (((red_buffer[i*2] + green_buffer[i*2] + blue_buffer[i*2])/3) +
(alpha_buffer [i*2 -1]))/2;

		if (ctr>=44100/1000.0)
		{
			ctr=0;
		}
	}
}
```

Figure 7: Code from audioSample_io.c: function for constructing a Sine wave to be processed into an image.

Serial Transmission of Data[7]:

Until this point the board has actually been performing two tasks, relatively, in parallel. The Audio_Task function being discussed until now, and second echo task are the two functions it is performing. The echo task performs UART (universal asynchronous receiver/transmitter) control. This is how data will be transmitted serially. Both tasks are dependent upon a flag variable. The audio task only operates when the flag variable is 0 and the echo task only when it is 1. Upon completion of the numerical conversion the flag is flipped to 1, and the serial transmission begins.

The code uses the same variable buffer locations as the audio task file by defining the values as 'extern' meaning for the complier to look elsewhere for the values. The code initializes the serial port for communication and sets up the baud rate (115200), as well as additional details for communication. The specific dissection of this code is not performed. The main purpose of the echo function is to send the minimized values of the filtered audio data to the computer. Figure 7, on the following page, showcases how this is performed.

```
if(transmitFlag == 1)
{
	int inx;
	len = 256u;

	for (inx = 0; inx <= 63 * 256; inx += 256)
	{
		status = GIO_submit(hUart_OUT,IOM_WRITE,&red_buffer[inx],&len,NULL);
			//timed_sleep(200);
		status = GIO_submit(hUart_OUT,IOM_WRITE,&green_buffer[inx],&len,NULL);
			//timed_sleep(200);
		status = GIO_submit(hUart_OUT,IOM_WRITE,&blue_buffer[inx],&len,NULL);
			//timed_sleep(200);
		status = GIO_submit(hUart_OUT,IOM_WRITE,&alpha_buffer[inx],&len,NULL);
			//timed_sleep(500); // maybe slow this down
		if (inx >= 63 * 256)
		{
			printf("64 Sent\n");
			timed_sleep(1000);
			transmitFlag = 2;
		}
}
```

Figure 8: Code from uart.c: code that cycles through the available data and sends it out serially.

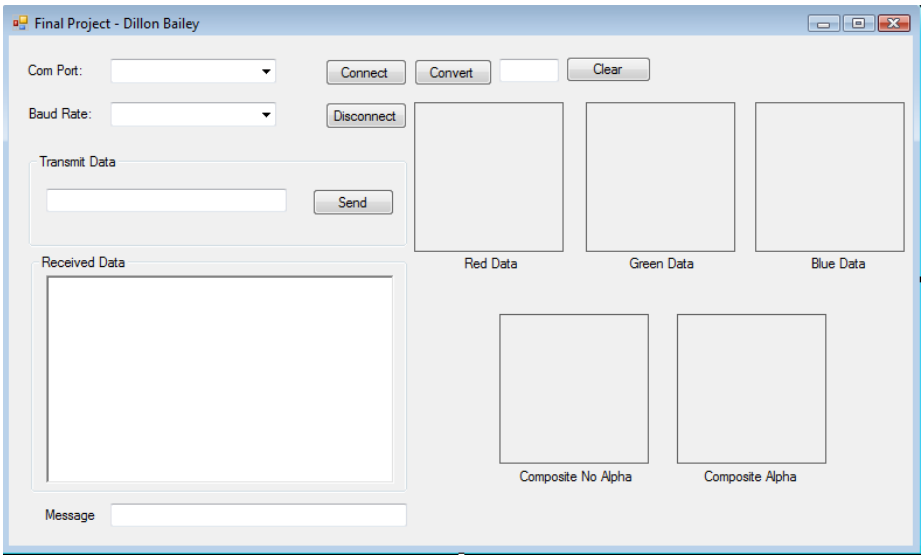
The code above is dependent upon the transmit flag being set to one. This occurs in the other file of operation. This is 'for' loop that is cycled over the information in each buffer. The four buffers to be transmitted house the red, green, blue, and alpha data. Each buffer is sent 256 bytes are a time, for a total of 1 kilobyte per loop. A timed_sleep function is provided to slow down the transmission of data. When the cycle has sent the required 64 times 1024 bytes of data then the process concludes. The transmit flag could be returned to zero, to allow the process to cycle. This is unnecessary, however, as the data needs to be interpreted on the computer side.

Visual Studio Project Code[1, 8]:

The code developed in Visual Basic (VB) is meant to retrieve data for the serial port and process it into image data for display.

Serial Data Retrieval:

The front panel for this project is shown in Screenshot 4. The display includes two drop down boxes, several textboxes and buttons, as well as 5 pictures boxes. The controls allow the user to select and connect to the serial port (COM1). The textboxes display information needed to see what data is being transmitted. The pictures boxes are used as labeled, to display the component data for each image processed. That is the red, green and blue information is displayed independently and the conglomerate image with and without the alpha effect are shown.



Screenshot 4: Front Panel design for the VB project. Includes controls and displays for retrieving data from the serial port and display the images

The serial port communication in VB entails adding a serial port control and altering the settings to match that of the transmitter, namely the baud rate (115200); as well as other information. The port is connected to by clicking Connect. This program waits until data is transmitted from the previous project and then stores the data sent. The program calls a data received function to perform the task of storing the data. This is shown in Figure 8.

```
Private Sub SerialPort1_DataReceived(ByVal sender As Object, ByVal e As
System.IO.Ports.SerialDataReceivedEventArgs) Handles SerialPort1.DataReceived
        Dim loc As Integer

        If (loc <= 64512) Then
            SerialPort1.Encoding = System.Text.Encoding.GetEncoding("windows-1252")
            For i As Integer = 0 To 1023 Step 1
                loc = count + i
                byteData(loc) = SerialPort1.ReadByte()
            Next
            count += 1024
        End If
    End Sub
```

Figure 9: Code from the VB project, this sub sets the encoding to receive data as unsigned char (bytes) and stores the bytes of data into an array.

The code is activated upon the serial port receiving a start bit. The information is in bytes, but the program must be told how to interpret those bytes. The encoding scheme sets the bytes to be received as whole single bytes of data. Each byte is read individually and stored into an array. The information being received totals to 65536 bytes of data. This corresponds to the 128 by 128 pixel image displays used. The purpose of the 'if' statement create a block at 64512 is to prevent over filling the "byteData" buffer.

After the code has been completely received the raw data can be converted into pixels and then into images. VB has a built-in function that takes arguments of either 3 or 4 integers and process then into a resulting color. The three values correspond to red, green and blue. In the event of the fourth then the first value is the alpha or opacity value. The function "pixelCR" in Figure 9 shows in detail how this process occurs.

The sub requires two inputs of an x and y coordinate. These coordinates tell the program where to place the pixel defined by the color determined. The fourth successive values taken from the 'byteData' buffer are converted to integers to be used in the "Color.FromArgb" function call. The individual pixel for each of the five images is defined as required to cause the necessary result. The red, green, or blue pictures only use the individual data for the color needed. The conglomerate images use all the data; one uses the alpha value and the other does not. The integers 'ci' and 'cj' maintain proper placement in the 'byteData' array.

```
Private Sub pixelCR(x As Integer, y As Integer)
        Dim red1 As Integer
        Dim green2 As Integer
        Dim blue3 As Integer
        Dim alphaA As Integer
        Dim pixel1 As Color
        red1 = Convert.ToInt32(byteData(cj + ci))
        green2 = Convert.ToInt32(byteData(cj + 256 + ci))
        blue3 = Convert.ToInt32(byteData(cj + 512 + ci))
        alphaA = Convert.ToInt32(byteData(cj + 768 + ci))

        ' red
        pixel1 = Color.FromArgb(red1, 0, 0)
        bmpR.SetPixel(x, y, pixel1)
        ' green
        pixel1 = Color.FromArgb(0, green2, 0)
        bmpG.SetPixel(x, y, pixel1)
        ' blue
        pixel1 = Color.FromArgb(0, 0, blue3)
        bmpB.SetPixel(x, y, pixel1)
        ' no alpha
        pixel1 = Color.FromArgb(red1, green2, blue3)
        bmpNA.SetPixel(x, y, pixel1)
        ' with alpha
        pixel1 = Color.FromArgb(alphaA, red1, green2, blue3)
        bmpWA.SetPixel(x, y, pixel1)


        cj += 1
        If (cj = 256) Then
            cj = 0
            ci += 1024
        End If
    End Sub
```

Figure 10: Code from the VB project, this sub creates individual pixel data for each image by getting the red, green, blue, and alpha data then using it where appropriate to construct the pixel and then store it.

The previous function is called repeatedly as the image is formed. As discussed earlier, the image is not constructed linearly as that does not appropriately showcase the information. The picture instead is constructed using a spiral pattern being in the center and moving counter-clockwise outward. The code for this development is shown in Figure 10.

```
        ' set to x/2 -1 same below for y; this is the start point as it cycles

        Dim x As Integer = 63
        Dim y As Integer = 63

        pixelCR(x, y)

        For a As Integer = 0 To 62
            'move down
            For b As Integer = 0 To (2 * a + 1) - 1 Step 1
                y += 1
                pixelCR(x, y)
            Next
            'move right
            For c As Integer = 0 To (2 * a + 1) - 1 Step 1
                x += 1
                pixelCR(x, y)
            Next
            'move up
            For d As Integer = 0 To (2 * (a + 1)) - 1 Step 1
                y -= 1
                pixelCR(x, y)
            Next
            'move left
            For f As Integer = 0 To (2 * (a + 1)) - 1 Step 1
                x -= 1
                pixelCR(x, y)
            Next
        Next
```

Figure 11: Code from the VB project, this code portion creates the necessary spiral pattern to display the sound.

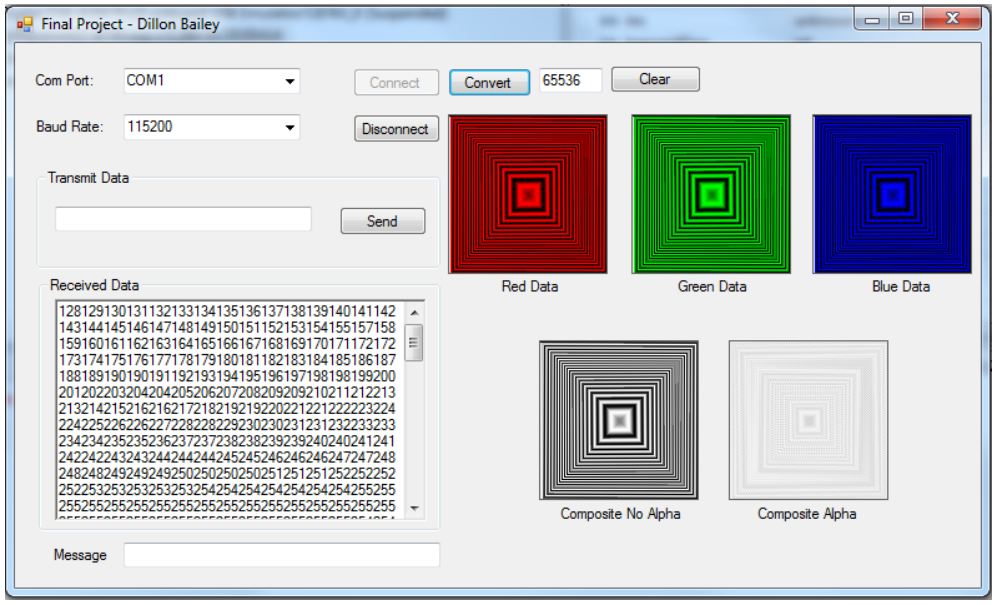
The process to create a spiral is a loop with 4 ‘for’ loops inside of it. First, the center pixel is determined and its value is set. The main ‘for’ loop controls the number of cycles and is one less than half of the size of the image. The cycle within this loop is to move down, right, up, left and repeat. The number of places to move each time controlled for each loop. The movements down and right are odd numbers starting at one. The movements up and left are even numbers starting at 2. Each step in each cycle places a pixel value by calling the previously defined “pixelCR” function on that x and y coordinate pair. This process continues until all the pixels are filled in the picture box.

This concludes the projects overall data acquisition, processing, transfer and implementation.

RESULTS:

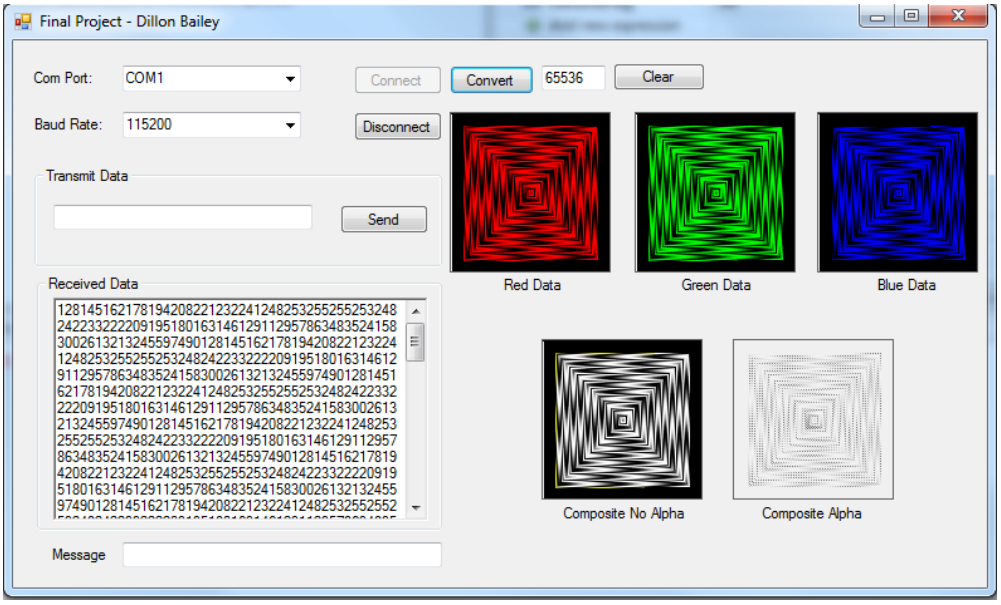
The project created cycles of images; several are included here for sampling.

Sine wave 60 Hz:



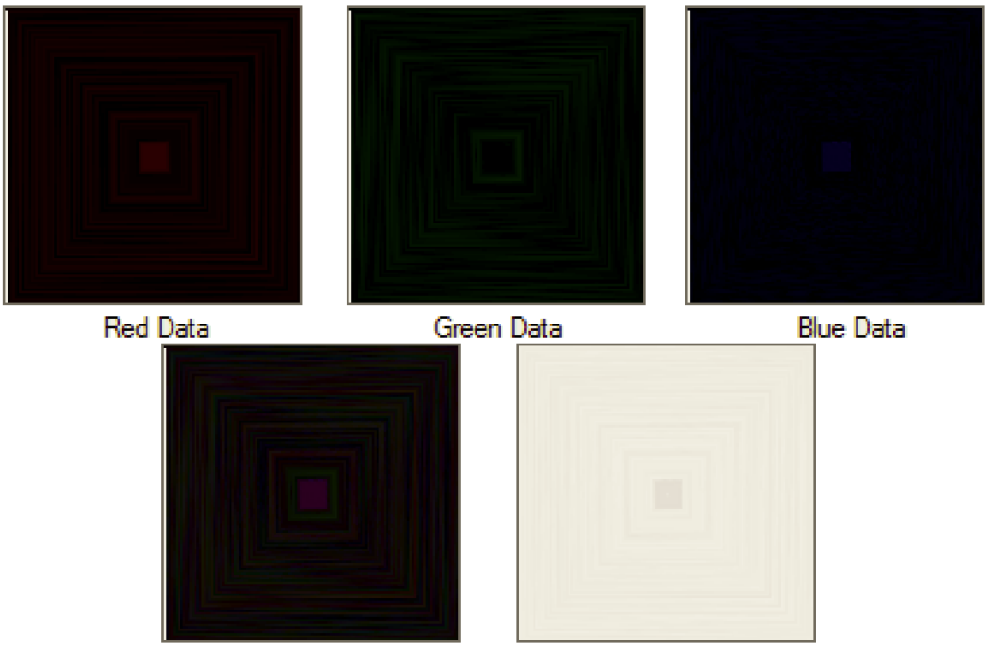
Screenshot 5: Results for image processing of a 60 Hz sine wave.

Sine Wave 1 kHz:



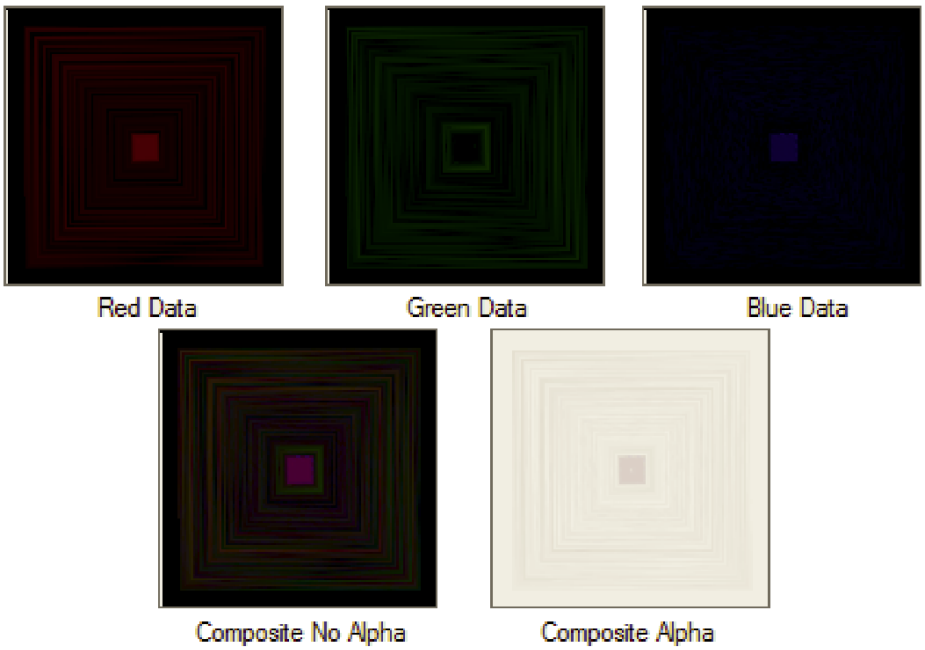
Screenshot 6: Results for image processing of a 1 kHz sine wave.

Data from a random sample (processed as absolute value divided by 256):



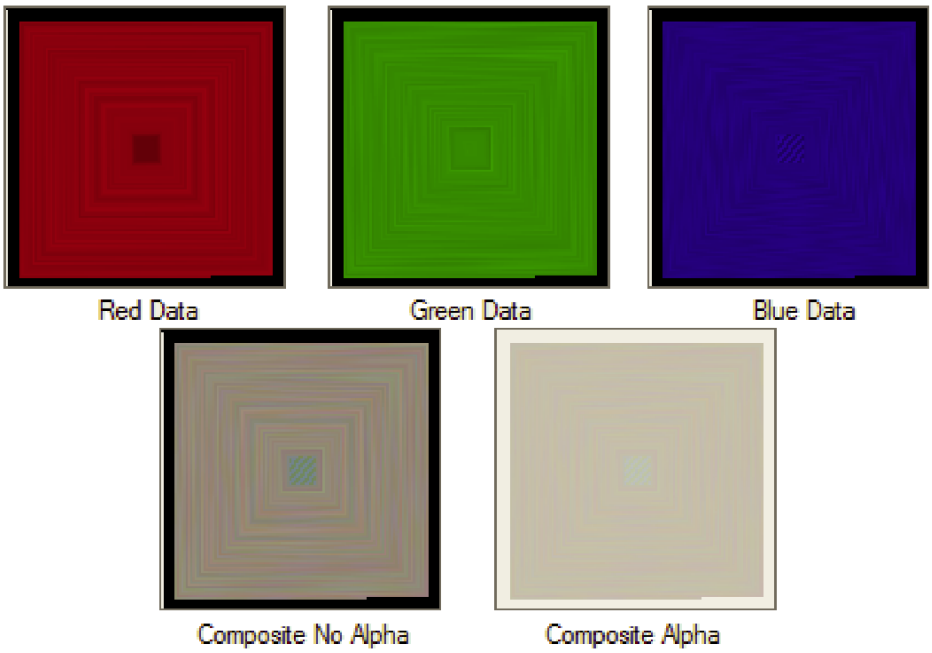
Screenshot 7: Results for random data processed by dividing the absolute value by 256, this is expanded to show the data better.

Data from a random sample (processed as absolute value divided by 128):



Screenshot 8: Results for random data processed by dividing the absolute value by 128, this is expanded to show the data better.

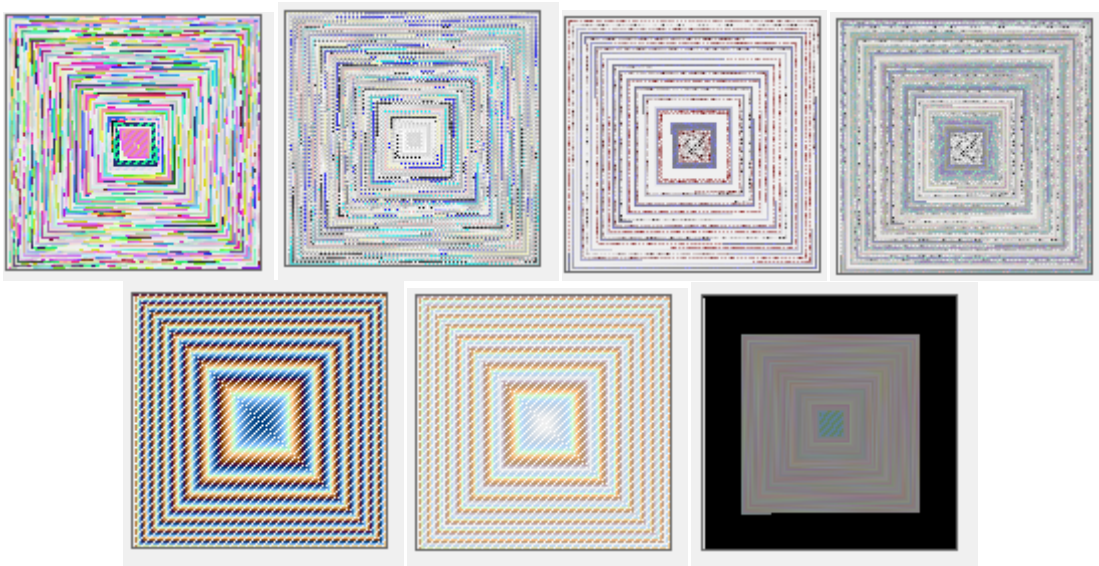
Data from a random sample (processed as adding 32768 and dividing by 256):



Screenshot 9: Results for random data processed by adding 32768 and dividing by 256, this is expanded to show the data better.

Other Samples:

The first row of samples was created with other means such as removing some samples from the data and processing. The first two images on the second row are sine waves with different frequencies between the colors. The final image is an incomplete image from a larger random sample; approximately 1.5 second sample.



Screenshots 10-16: Other images processed with a variety of techniques

DISSCUSSION:

The project's purpose is to allow for some research into best implementation of conversion of sound to picture. It is not meant to be an all conclusive way of performing this task. Instead it is meant to provide one means to showcase that the information need only undergo a transformation and still maintain the inherit information to that sound. Additionally, by creating a comparison between a sine wave and random sound characteristic of that sounds smoothness can be portrayed. Finally, and most importantly the processing of digital audio into an image can make for interesting pictures.

The idea of this project is more research and exploration than defining an exact method to how it must be accomplished. Several approaches were used to create an image from sound data. The approaches have all been discussed in the implementation section, but will be recounted for completeness here.

The main digression between the ways the images were developed lies in the conversion to an integer ranging from 0 to 25. The data was altered in three ways. To see how it was altered, it needs to be understood what the initial data was. The sound is stored in 'short' variable type, specifically signed. This allows the values to range from -32767 to 32767. Or if unsigned, the values range from 0 to 65535. This number of these values is 256 times greater than the values allotted for a color's integral value. Since a color cannot be negative the sound data must be transfer to only positive. This can be accomplished by taking the absolute value of the function, which was used in two processes. This seems to actually hide some data as the cycle does quite compare when all the values are represented as their magnitudes. However, this approach was used with two final implementations. These two styles are separated by then the lowering of the value by means of division. One process divided by 256, the other by 128. The reason for dividing by 256, although limiting the data, is more closely associated with what would be expected from a conversion from a range of 65536 values to 256. This was used in the first attempt. Noting as was said, that this creates value only up 127 the image seemed to lack information. Therefore the second style in this approach was to divide by 128. This scales the information back up to full range of values and gives an output range up to 255. This method provided more image depth. Finally, to maintain the proper positive to negative swing in the cycle of data, "32768" was added to each value. This prevented any value from being negative and allowed the full range of values required to create a good image. The additional benefit was the maintaining of all the cycle information. The image created from this approach was much richer than the other attempts.

Other attempts at creating images included removal of some of the sample data. The reason behind removing some of the data was because it was initially thought that there would not be enough detail in such a small grouping of information. This approach created more white space in the images and a much greater separation in color. However, in the end, this approach does not adequately represent the audio sample and therefore was not wholly utilized.

In dissection of the results obtained, the sine waves functioned much as was originally thought. The image produced is obviously cyclic and a smooth shaded transition from center to edge. As the frequency is increased the black space of the cycles no longer aligns as smoothly and therefore creates a drastic image. Combining colors made from the same frequency create a gray shaded image. Combining colors derived from different frequencies create a rainbow array of colors through a smooth image.

The images processed from random samples of audio by the different methods did not quite show what was initially expected. The data seemed to show a rather uneventful sample. However, by looking at the 3 methods and what they specifically divulge about the audio there is information to be gleaned from this process. The red, green, and blue samples showcase the individual ranges of frequencies and the content provided therein. The samples in Screenshots 7 and 8 show the high and low or treble and bass ranges better because they are zero referenced. The black space in these images is the zero reference point. Therefore the composite image without alpha presents that those values at the extremes of frequency better. In Screenshot 9 the composite image shows the mid tones better. The green and those values at the top end of the bass (red) and low end of the treble (blue) are better displayed. This is because the zero reference of the image is moved from zero to 32767, the center of the band. This can be seen that more image data is present. But specifically it is these middle range frequencies that are assisted.

The alpha images are valuable as well. Specifically in the sine images dual, revere spiral forms can be seen. This effect is best seen in Screenshot 5. This is a semi-compounded average across the length of the sample. It expresses how the sample moves across the entire sample as portions of the sample are darker the information present is more present at this value throughout the image. As the image become duller or less visible, data at this level is less present throughout the image, or sound sample.

Overall, this project was successful in accomplishing the goals set out from the start. But, there have been several points of interest where lessons were learn based on failure. Initially, a real-time processing was attempted, but was quickly determined to be impossible. The data recorded and filtered three times could not be done within a real time scale. It was then chosen to record and filter the samples. In terms of filtering, originally, the filters were set for performance over limitation of order. The first three filters developed were on the range of 2000 coefficients. The reason for implementing high order filters was to provide the best transitions between ranges of frequency. However, this was unnecessary, plenty of data is being removed in the transition and it still works. Also, by having filters of such a large order required a processing time of around 20 minutes for a 1.5 second sample.

The decision to limit the sound sample to such a small value was posed on the same premise as lowering the filter orders. The 0.75 second sample produces enough data to create a 128 by 128 image, as shown. The issue with creating a larger image and audio sample is that the processing time to do so increase exponentially. That is to say to increase the image to a 256x256 it would require 4 times as much image data and thereby over four times as long in processing time. Unfortunately, the turnaround time for producing the current images was on

an order of about 15 minutes apiece. The lengthy processing time is partially due to the filtering, which still takes 3 to 5 minutes with order of 160, 350, and 152 for bass, mid, and treble, respectively.

An additional problem and execution time slow down is the serial port communication. The serial port communication presented a few problems in project construction. First, solving the problem of how to communicate from the DSP to computer, serially, was fraught with difficulty. The main issue was getting the DSP and the VB project on the same communication idea. The data never seemed to be received correctly, at the start. The first trials used a string catch for the VB project. It was determined that this removed data from the transmission, because some values cannot be interpreted as strings and therefore are ignored. This was fixed with reading individual bytes of data. However, the data still was correct. It was explored that sending the negative values would not be able to be interpreted in the appropriate way. The solution was to make all the values positive and on the correct order, that is unsigned byte (8bits) data. The VB project had to store this information in the same format. Finally, the communication between the devices needed to happen at the same speed. Therefore the baud rate for both needed to be set. The rate was set to 115200 bits per second, or 14.4 kb/s. At this rate the transmission of 65.536 kb should take just over 5 seconds. However, the problems persisted.

It is still undetermined as yet, but the program crashes at some point when too much information is attempted to be transmitted serially. The program was implemented with a wait function to see if pausing in between each transmission would help. It did help some. To get a complete program cycle the wait function had to be over seconds between each sample (1024 byte) transmission. This equates to about a 7 minute transmission cycle. The alternative to this is to run a break point after the transmission, and control the sending of data manually. The issue with this is the removal of automation. As well as transmission failures continued to happen; if one clicks too quickly. However, the benefit of manual is to reduce the transmission time by half.

The only issues present on the VB program side, were as discussed with communication and a minor issue with math. The minor issue was created the spiraling pixel entry. The problem was overcome by working through a small example to derive the process required for any size spiral.

Additional implementation or fixes from this point would enhance the results of this process. Although the project is complete, here are some more ideas to improve it. The project would become more automated and better if the serial communication were either fixed or a different means of communicating with the computer were designed; such as USB. Assuming the communication issue could be improved the size of the audio sample and resulting image would easily be increased. This would result in a much better view on an audio sample present in an image. Another way to improve or create other images is to perform averages as the data for each color band. This would allow larger samples of audio to exist in smaller images. Finally, implementing a Fourier transform would allow conversion of frequency data in an image vice the amplitude information.

Overall, all goals set forth in this project were accomplished. Although this was not specifically on the scale originally thought. The project allows the user to record an audio sample. Then the program filters that sample for frequency content into three ranges; bass, mid and treble. These ranges are stored separately and converted values better fit for color conversion. The data serially transmitted from the OMAP L-137 development board to the computer. The data is received in a Visual Basic project that converts the raw byte data into colors and pixels. These pixels are then stored into an image. The pixels are added to the image initially at the center and spiraling out counter-clockwise from there. The images are displayed in the component color parts as well as in a composite view with and without an opacity value. The project also allows creating an image in the same way from a created sound sample; such a sine wave. The project was a success.

REFERENCES:

1. Website via Google search for “Serial Port Communication VB 2010” http://tiktakx.wordpress.com/2010/11/21/serial-port-interfacing-with-vb-net-2010/