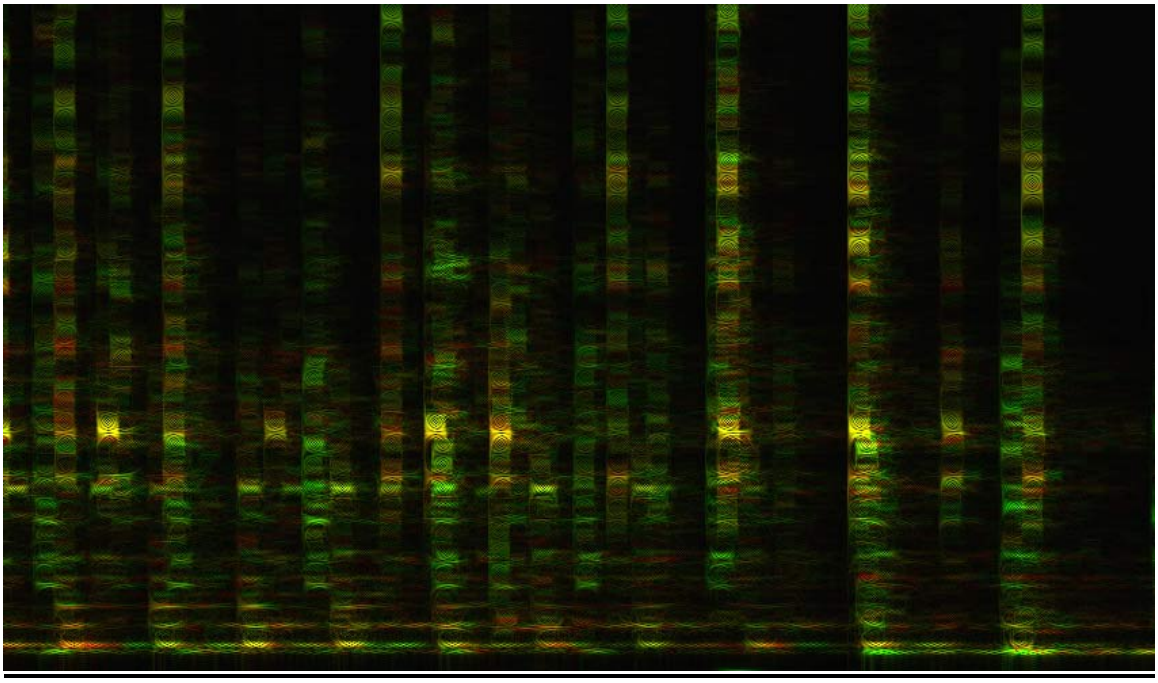
**18-551 Spring 2002 Group 10**

**Chris Graves & Stephen Tsou**

cgraves@andrew.cmu.edu & stsou@andrew.cmu.edu

# WAVE CHEZ-ZAM

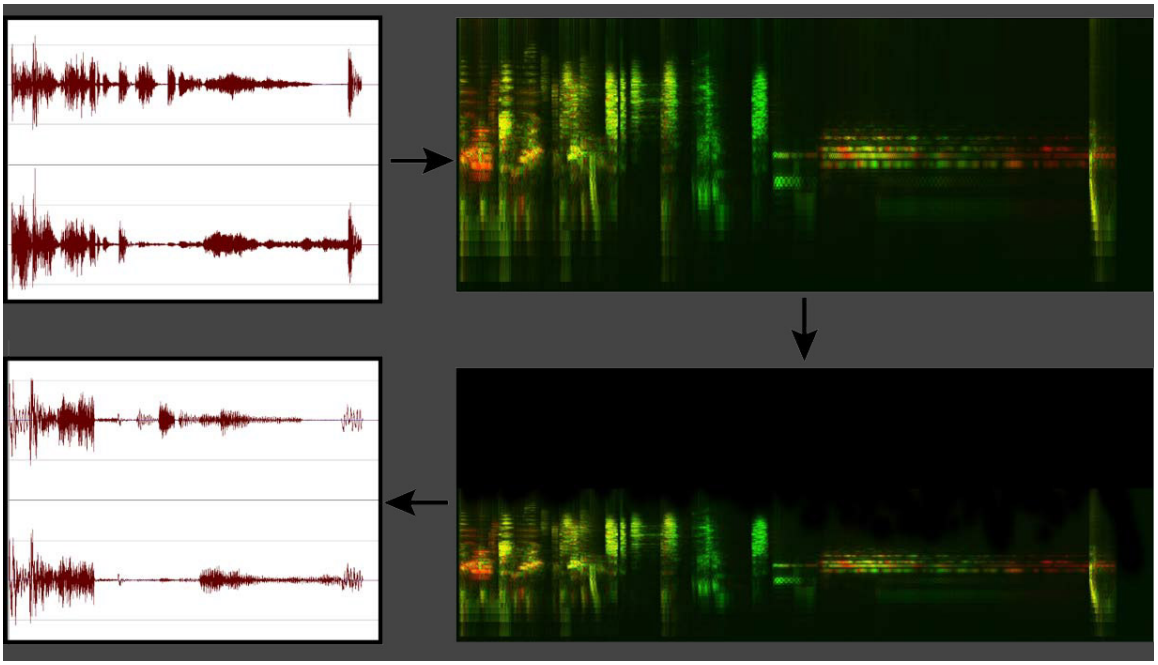


**What is the project.**

We are interested in being able to visually manipulate sound. Our project will create an unorthodox yet straightforward and intuitive way of working with sound. Electronic sound composing, manipulation, playing can be a very abstract (and intimidating to some people) process. One could transform a waveform sound file (amplitude over time) into its spectral representation (three-dimensional: frequency over time, with brightness being amplitude at that frequency and time). This graphical representation offers unique sound manipulations – all the usual waveform manipulations and more. One could take the image and manipulate it any way imaginable in an image editor (i.e. Photoshop, Paintbrush) and convert it back to audio with no loss of information in the transfers.

In the near future, this area of research may lead to the possibility of visual-sound relationships that will give an alternative sight to blind people and alternative hearing to deaf people.

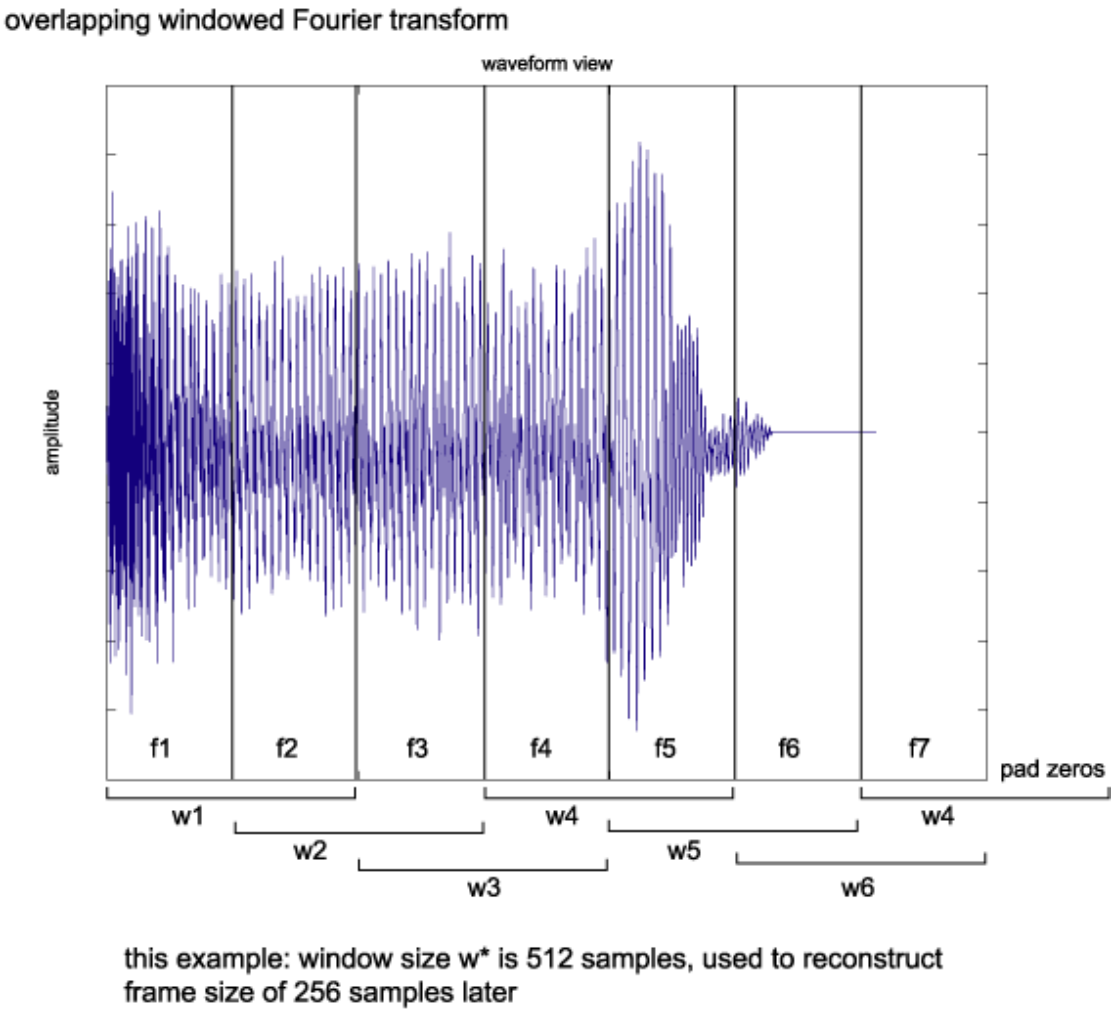
convert audio to sonogram image, edit (in this example pitch-shift while preserving duration), convert back to audio.



**SOLUTION**

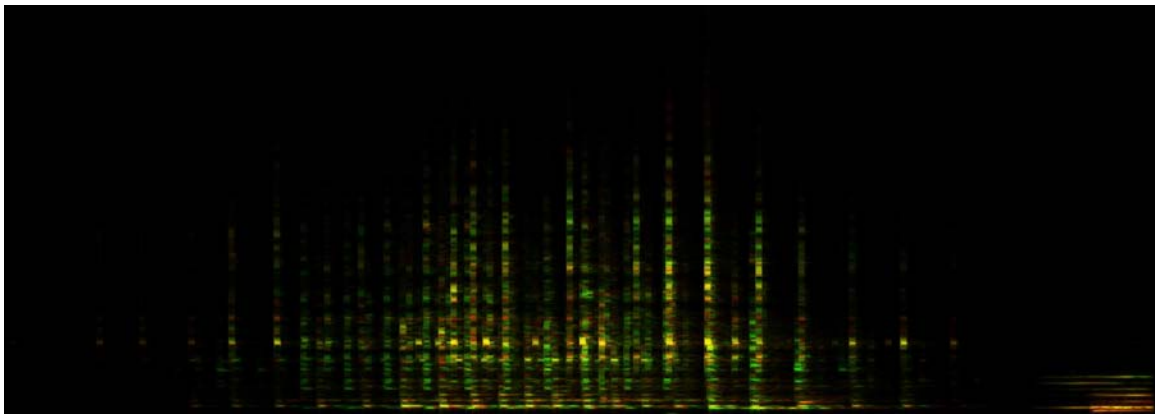
**Implement SOUND to IMAGE**

From sound to image, we used an overlapping windowed Discrete Fourier Transform (specifically a radix-2 Fast Fourier transform – see code section). With this algorithm we stored more information than is needed for reconstruction to allow for editing and drawing accuracy: we analyzed and stored per frame the FFT of that frame plus more, which overlapped with the next frame(s). The specifics are in the diagram below. Thus, we had the ability in reconstruction to build more of the waveform (framesize + overlap) than we actually built (framesize). We wanted the sampled region (framesize + overlap) to be high enough resolution to at least include all of the frequencies that correspond to the standard 12-note musical scale (see note-scale diagram), which in the lower octaves has less than 1-Hz frequency separation per semitone.



We achieved the desired high-resolution on frequency axis with the overlap amount. Without overlap, the resolution is not high:

See the vertical image resolution (frequency bins) of a sonogram with overlapping:



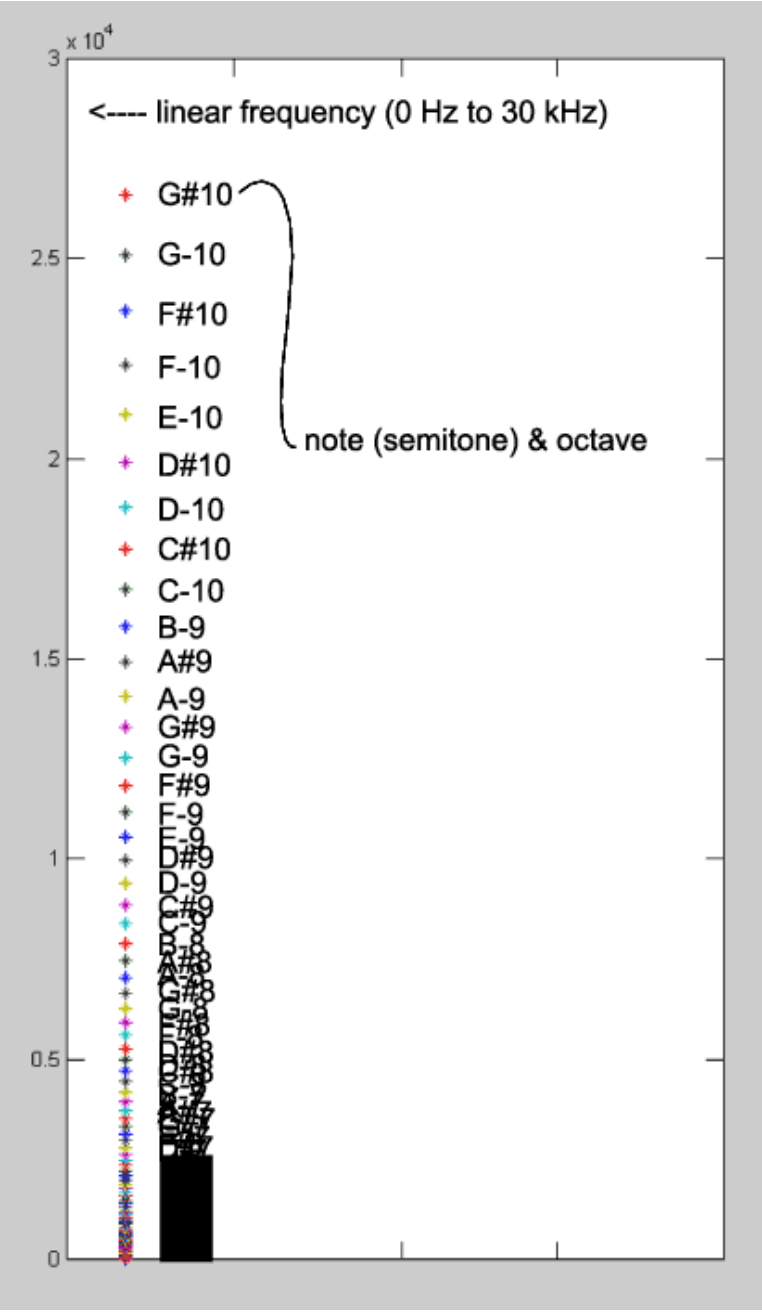
See the vertical image resolution (frequency bins) of same sonogram *without* overlapping:

It would still be possible to reconstruct the sound properly using the “low-resolution” sonogram. However, it is not at all useful for manipulation purposes: pitch-shifting it up by 1 pixel in y-axis would result in a greater than 1-octave pitch-shift. We desire much greater accuracy.

In this process, we converted .WAV waveform files to .BMP image files. This included learning how to encode and decode .BMP files. The red, green, and blue channels were all used to store our sonogram. The image is saved as a 24-bit BMP (8-bits per color R,G,B). The red channel is magnitude of freq (sample-rate / height of image * current pixel in y-dimension) in right channel of stereo audio file. The green channel is magnitude of freq in left channel. A mono audio file is converted to a stereo one on load (copy the one channel to both channels). The phase is stored in the blue channel with the lower four bits representing left and the higher four bits representing right. This may or may not actually aid in recreating an accurate sound in decoding.

For a stereo sound, we stored the sonogram as follows: R (magnitude of right channel), G (magnitude of left channel), B (phase of each, limited to 4 bits each).

The overlapping windowed FFT creates a vertically linear representation. Unfortunately, the human ear perceives frequency at a nonlinear logarithmic scale:



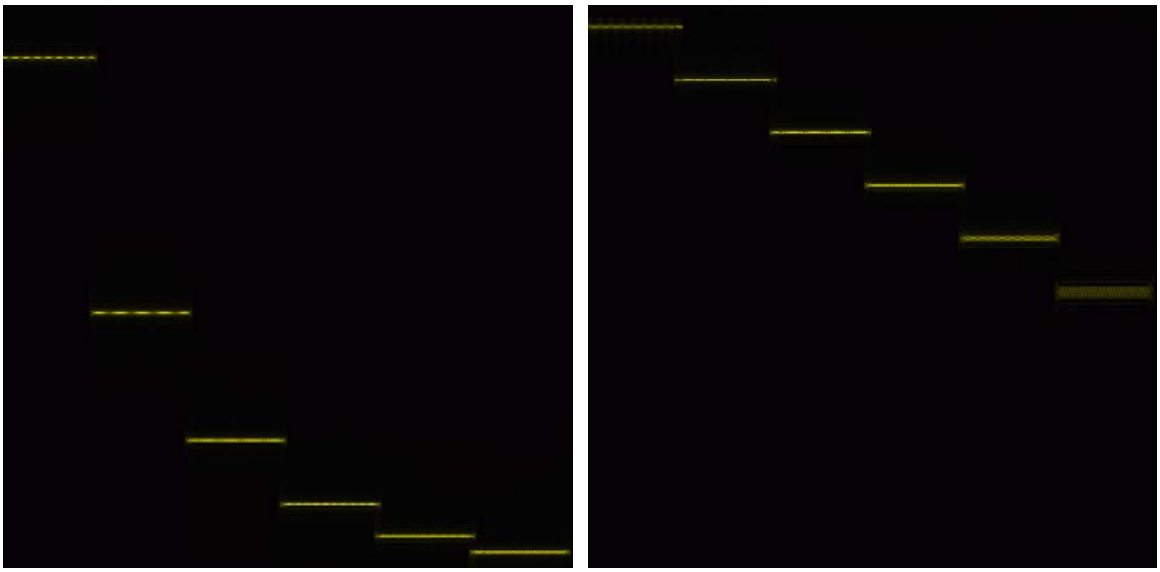
To implement a logarithmic spacing in the image we came up with the formula:

Float Logbin = co*2^(j/(12*8))/(fs/n)

where co = 16.35159783128745 Hz (note C in octave Zero), n is number of FFT bins, i is 0->n, and the reason for 12*8 -> we are assuming n=1024 for this formula right here, and 8 "notes" per semitone gave us approximately 1024 of the logarithmically-scaled bins.

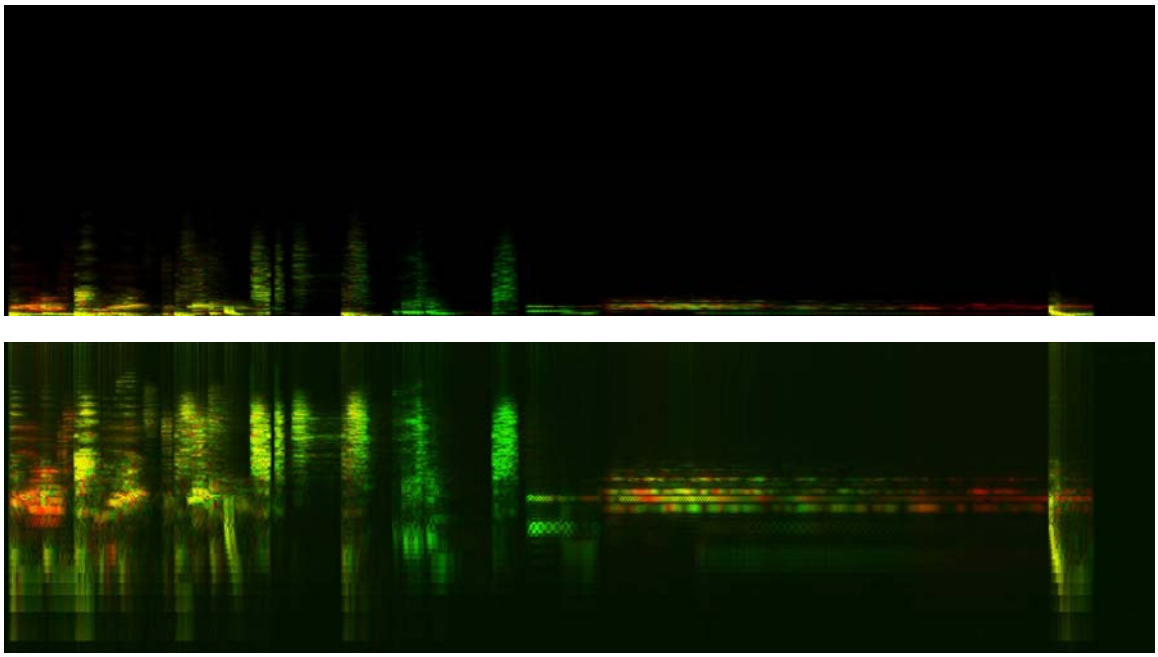
This will return the logbin you want depending on the frequency that you are given. This is a more efficient allocation of representing data since it is more attuned to the human ear, and therefore the spectrograms appear more “full” (most sound files that we deal with have a lower-frequency majority) and editing them makes more sense – every 8 pixels on the y-axis is 1 semitone. The formula is used in our code when accessing the real & imaginary FFT components (as we calculated magnitude & phase).

Sonograms with linear vs logarithmic frequency spacing:

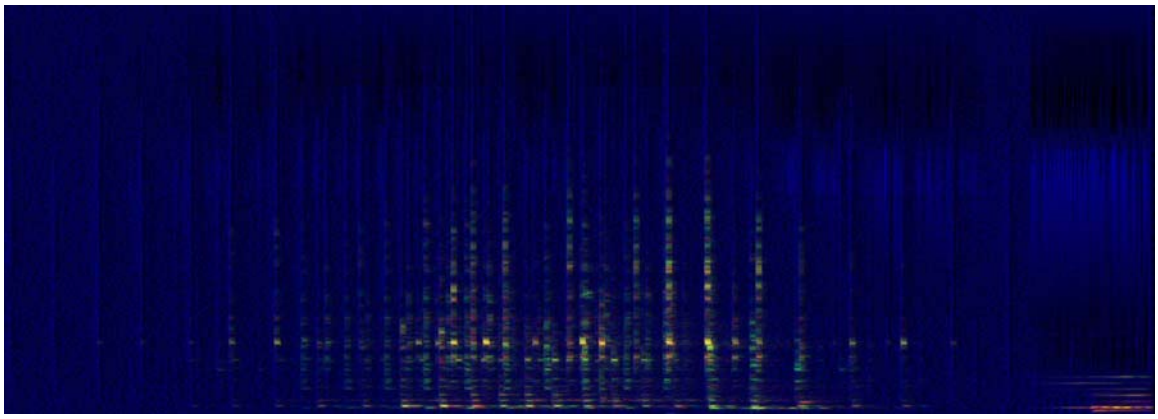


As you can see in the linearly-spaced sonogram (left) the notes (yellow bars) are logarithmically spaced, and in the logarithmically-spaced sonogram (right) the notes are linearly spaced.

Another example of linear vs logarithmic spacing:



We have not been encoding phase in the previous examples because graphically it is not useful to view nor edit. The phase value (magnitude of blue) cycles in most cases:



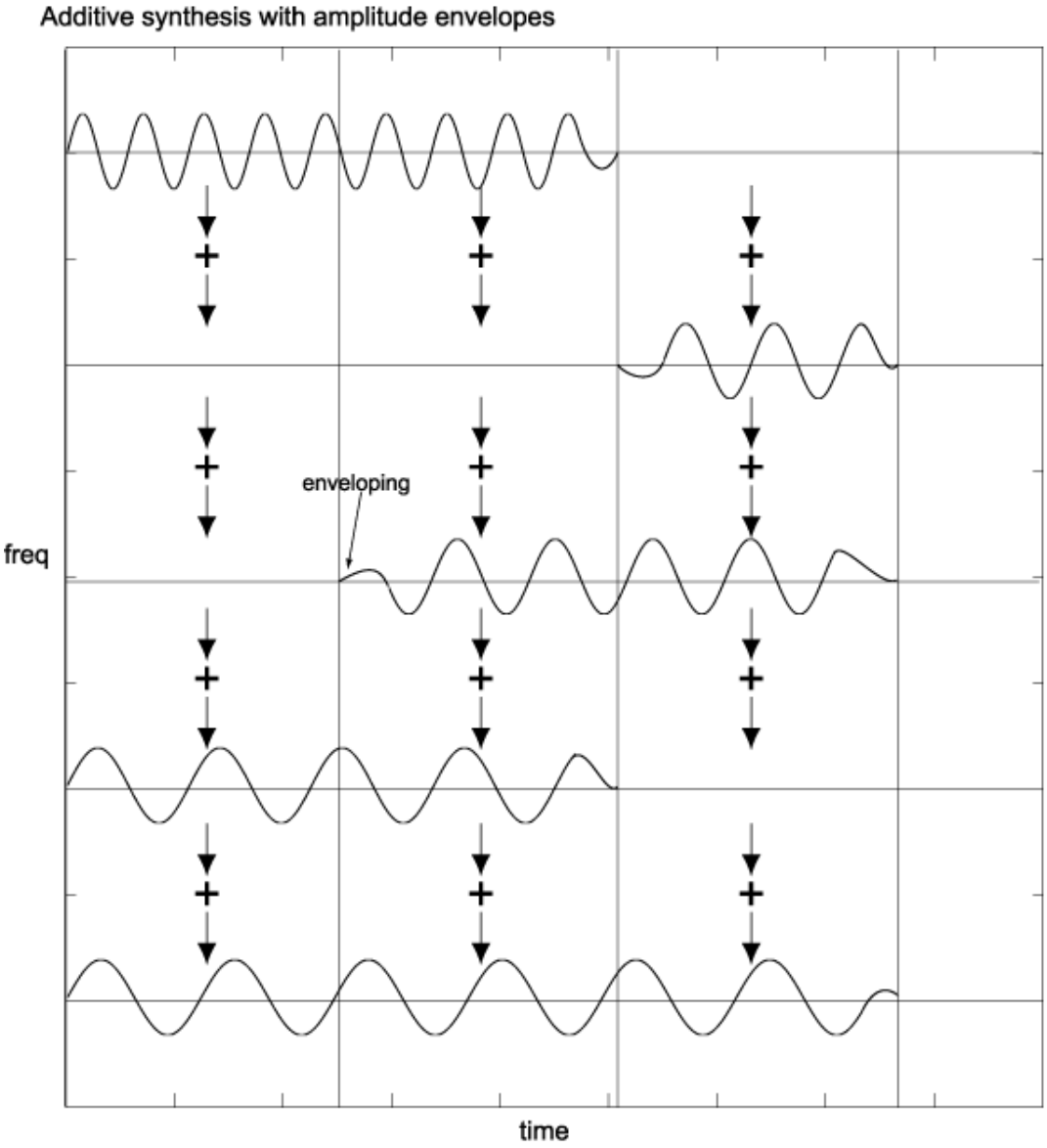
**Implement IMAGE to SOUND**

The second half of our program takes sonograms as input and outputs sound (at this stage it converts .BMP to .WAV). This is not as simple as doing an Inverse FFT. An IFFT would work if the sonogram was not edited in any way. However, our objective is to allow extensive editing. When the user edits the sonogram or draws from scratch, phase information will become erroneous. The misaligned phases from shifting the sonogram vertically, for example, or the lack of phase information when the user draws in the sonogram, will cause many unwanted clicks, and require patching up. We implemented additive sinusoid synthesis which optionally takes phase into account. Neglecting phase, it acts as a large sinusoid oscillator bank.

We attempt to mimic the IFFT by adding up a large bank of sine wave oscillators - one for each pixel in the y-dimension, for each frame in the x-dimension, with their amplitude envelopes interpolated between each frame. For each pixel in the y-dimension, it generates a sine wave at the corresponding frequency.

So if our image is 1024 pixels high, and our sample rate is 44.1kHz, then the sine wave for the $1024^{th}$ pixel (top row of image) should be generated at a frequency of 22.050kHz. With linear frequency spacing, in this example, each pixel would represent 22050Hz/1024 pixels = 21.533 Hz/pixel. We also added the option of logarithmic frequency spacing in this step, to correspond with the Sound to Image option (see code). The algorithm interpolates the magnitude (pixel intensity in red or green channel) of each successive column. So for the $1024^{th}$ pixel (top row) example, if the first pixel in that row is of value 255 in the red channel and the next pixel in that row is of value 0 in the red channel, the sinusoid will be interpolated to start at amplitude 1 and fade out to 0 within that time-frame. You can set the time-per-frame (actually samples-per-frame) when running the program – 64 samples per frame is a useful parameter to use. The ability to

set this parameters allow time-stretching of the entire image/sound-file without editing the image.  The same effect could be achieved by retaining the samples/frame value but just stretching the image out horizontally.




We ignore the phase for now, because after some testing, we learned that phase is mostly imperceptible to the human ear when resynthesizing. It affects only the timbre and not noticeably. When editing a sonogrammed waveform you cannot edit the phases properly anyway, so we would have to go through and 'fix' all of the phases, so it's not

worth the trouble. Maybe at some point in the future. We also had it use the erroneous phases after editing and the sound comes out sounding robotic since the phases are shifting around so much.

We did implement an algorithm that interpolates betweens successive phases, just like the magnitudes, in additive synthesis; however, erroneous phases cause erroneous frequencies and a consistently artificial “glaze” on the sound. Some references say you can mimic IFFT perfectly by *cubic* interpolation here – this would be useful if we did have valid phase information still. As you can see this is a complicated problem and deserves some in-depth research.

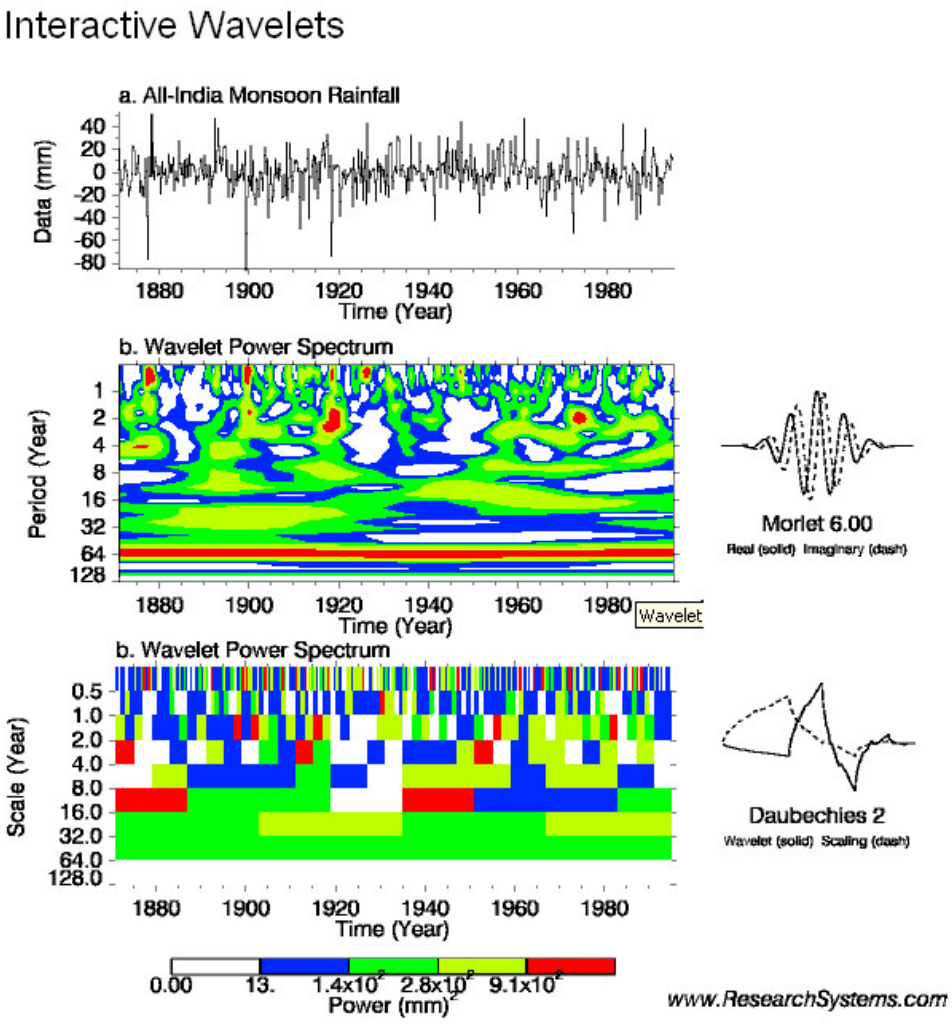
Since the phase of a channel is stored in 4 bits, there are sixteen points of the phase in a total of 2*pi or 360 degrees. Therefore, (pi/8) radians is our phase resolution. It snaps to a grid at 22.5 degrees (pi/8 radians) therefore giving a very rough estimation of phase.

If we had used a 4-channel image format like CMYK (Cyan-Magenta-Yellow-Black) colorspace we could have used a full 8 bits for each of the phases, but it is not the most common bitmap format and is difficult to edit - lack of color values is white, and the presence of black would be for one of the phase channels. The other option to have a full 8 bits for the phase is to only use mono audio files, but we didn't want to sacrifice sound so we stuck with stereo. Furthermore, the phase information effects mostly timbre and not the overall frequency so we are still flirting with if it is really necessary. We went with the RGB format and encoding both phases into the blue channel – humans will not be able to edit the phase properly anyway. Despite all of the inaccuracies, this is still usable.

**FUTURE PLANS**

SOUND TO IMAGE

1. Wavelet transform as per the ION Interactive Wavelets page – higher resolution, somehow, than the standard Wavelet transform. While this is a good idea, in essence we already have accomplished it in our nonlinear (log) overlapping windowed FFT. In fact it is very possible that our method works better due to user-specific options in the non-linear representation.



In the image above, taken from the ION web-page, the WT using Daubechies mother wavelet yields the usual WT results. However when using the Morlet wavelet their results look high resolution than usual and possibly useful for our spectral images.

2. CMYK phase storage. In the future, we may find the need to use 16-bit-per-channel RGB images or 8-bit-per-channel CMYK images if representing the phases with 4 bits each is too much of an approximation for accurate reconstruction later. There are a few options for this conversion algorithm, each with strengths and weakness and are interdependent with the reconstruction / image-to-sound algorithm.

3. Possibly user-specified spacing from a file – Hopefully, eventually the user could specify the range of limits and frequency scale that he/she wants represented in the image as well as FFT-size. Right now, there is only one real FFT size (1024) that the program prospers in and there is only a full range of frequencies represented in the.BMP.

4. We may try to write and use .PNG files instead of .BMP. .PNGs are more efficiently stored image files but the libraries that we attained to use .PNGs did not work for us. At the time, file sizes were not important because we thought that getting the algorithms working was a big priority.

IMAGE TO SOUND

1. Additive Synthesis - sinusoid doesn't have to be base for resynthesis – could be square wave, triangle, etc for interesting effect. Pitchbending between two disjunct frames : portamento. Speed optimizations: we could have a prewritten sine function look-up table so that the program does not have to calculate the values every time.

2. IFFT overlap-add method instead of additive synthesis - will be much faster but lose control of the options available in additive synthesis. Also problems and inconveniences: Fix phases if they're broken (difficult problem); Non-relevance of phase in human hearing; For logarithmic resynthesis we'd have to construct enormous FFT dataset to give to the IFFT because it needs linear-frequency spacing. To make it pitch-accurate we would have to expand the logarithmic bins into more than 22000 FFT bins, so this method seems out of the question. For example if we are using an image that is supposed to have logarithmic frequency spacing, we would need to build an oversized FFT and IFFT that, for each frame. The only benefit would be a great speed increase. We have read the IRCAM (ircam.fr) has patented an IFFT additive re-synthesis hybrid.

3. Inverse Wavelet Transform – This is of course reliant if we used the Wavelet Transform to generate images in the first step.

4. Parallelizing on EVM. 8 processors can be working in parallel but we only used one. This is because of our worries of correctness the entire time. In the future we could more fully utilize the capabilities of the EVM board to make it faster.

GRAPHICAL USER INTERFACE

1. At least able to display image with a 'cursor' / vertical line tracing the playback point in the spectral image.

2. A program that resides in memory and captures the screen of the operating system, on a timer. This would allow us to use other program's extensive GUI's to our advantage, and allow us to synthesize video into sound, as well as

anything the user can get on the GUI of their OS. The program will have realtime audio output and synthesis parameters, as well as specifying a window on the screen to capture from. This GUI will also have a cursor to track the moment of the image when it is being played. Most likely it will be a vertically thin window akin to those programs that sit in a horizontal banner window and place advertisements.

**PROCESS SUMMARY**

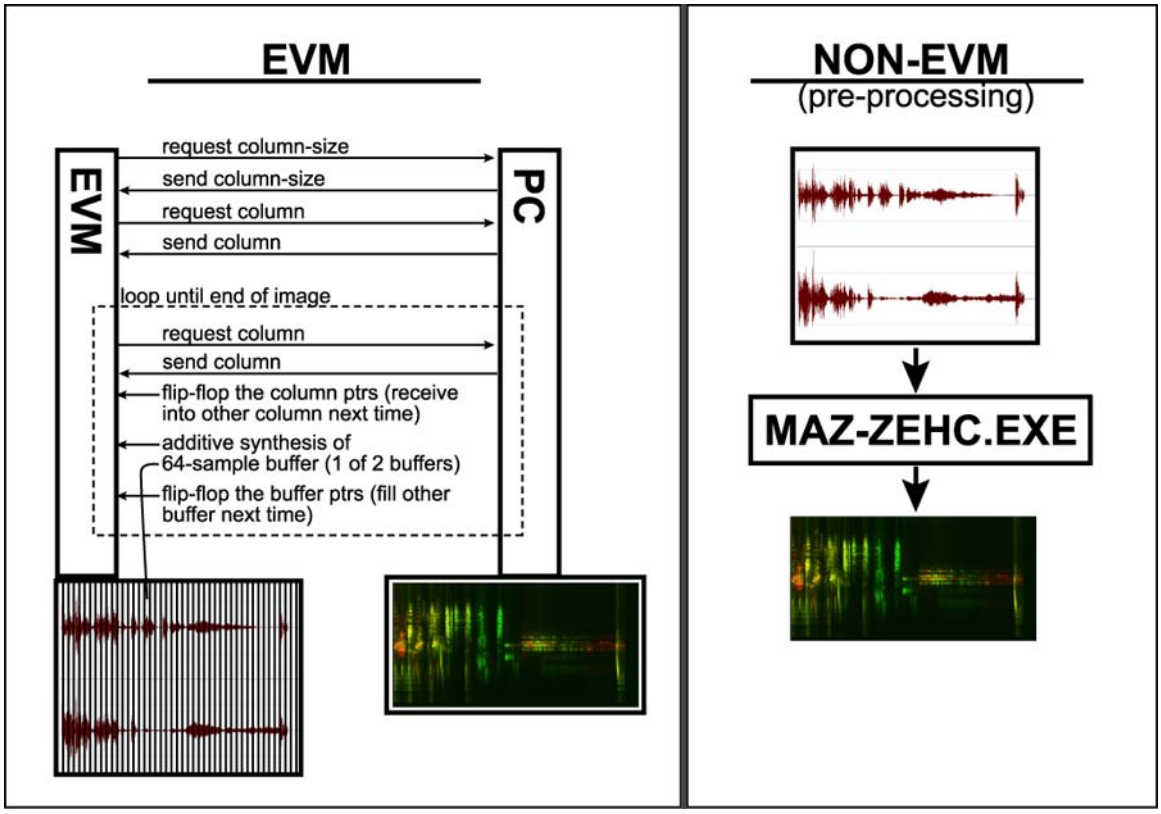
In the beginning there were many paths and algorithms that we could've used to solve the problem.

In sound->image we were able to eliminate a basic windowed discrete function just because it wouldn't give us a high enough frequency resolution to create the accuracy we were looking for. In addition we dismissed the idea of a Wavelet Transform, somewhat by necessity. We were unable implement it at a high enough frequency resolution either but we had hope early on. There was a website and organization that we saw that had an interface that would execute very fast and high-resolution wavelet transforms. Unfortunately, we couldn't figure out how it was done.

We read extensively on the subject of phase vocoders and methods of additive synthesis, stochastic versus deterministic sounds and how they could be generated separately, harmonic-tracking in resynthesis, and decided on a method that would give us the most freedom.

First we were able to secure a satisfactory algorithm in Matlab to go in both directions. We successfully converted sound to image using overlapping windowed FFT. We then chose to use additive synthesis to go back to sound from image.

**ROLE OF EVM**




In the end, the EVM was working in the above described process. It would not output in real-time however, and we do not believe that any amount of optimizations on our part could make this happen. We could implement sine lookup tables, parallelization, and so on, but additive sinusoid synthesis is just processor intensive. In the future we will try to speed it up. However, it needs a nearly 1000x speed boost to become realtime.

As with all groups we had many small problems using the EVM board. There were some quirks using the board that just took time and experience to straighten out. For instance somehow data kept on getting written to a channel that we specifically coded to stay empty (we set that part of the buffer to zero after setting the other half).

At some point we hit a wall in our use of the EVM that we could not hurdle. The .wav files created using additive synthesis constantly fed back erroneous data. Even with critical apparatus turned on, we still could not locate why there were random spikes in frequency in the sound file. Basically our progress halted to a stop.

We then came at a crossroads as to whether we wanted to spend the rest of the time in lab hacking away at this problem with no apparent progress anywhere in sight, or dedicate the time making the algorithm work outside of the EVM board.

For the most part this was a wise choice. We were able not only to get something working but we were able to brainstorm on possible ways in which the program failed using the EVM. With spare time, we moved our working, debugged code back onto the EVM and it worked properly.

**LAB DEMONSTRATION**

To a basic user, our program will allow the user to graphically manipulate a sound or create a sound using an intuitive and straightforward approach.

We will be able to feed in a .wav file. In a very short amount of time, it will create an image (.bmp) file that will be a visual representation (FFT) of the .wav file with vertical axis representing frequency and horizontal axis representing time. Since we were unable to create a GUI, we will open the image in Microsoft Paint and try to track the moment in the image being played with a mouse.

An option that our program allows is that we do not necessarily have to start with an sound file to sample. One could simply open up and create an image in an image creator/editor and the program should be able to create an accurate sound representation of the image. This was also part of our lab demonstration.

**CODE REFERENCES**

Most everything here was self-written. Much of the reason why we used mostly our own code was because much of the code we found was not helpful to us.

rfft() at http://www.jjj.de//fxt/fxtpage.html by Bill Simpson, based on a Fortran method by the well-known Sorenson. It computes the Radix-2 FFT from real values.

On the EVM side, we used a lot of the EVM side code from Lab 2 and Lab 3.

All our other code (over 1000 lines) was written from scratch:
BMP loader and saver, WAV loader and saver, Sound->image transform using rfft(), Image->sound transform using additive sinusoid synthesis, Logarithmic frequency spacing in both transforms, etc.  See code at end of report.

**NEW TO 551 / PRIOR IN 551**

NEW:

Most of what we have done is rather unexplored territory, not just in 551. The parts of the process - overlapping windowed FFTs, additive sinusoid synthesis, are not new but our application is, and the addition of the logarithmic sonogram representation effectively achieves a wavelet-style representation but with greater resolution than the sample-rate-halving wavelet transform that we know of - but it is slower to compute and uses redundant information to achieve it's high resolution.

Prior 551 work that relates: Possibly some in the fields of synthesis (additive) and pitch-shifting in the spectral domain.

**REFERENCES**

"Computer Modelling of Sound for Transformation and Synthesis of Musical Signals" by Paul Masri. 229-page PhD thesis discussing in detail computer modelling of musical signals. http://www.een.bris.ac.uk/Research/CCR/dmr/pub/96-Thesis-PM.html

Deterministic Synthesis - discussion & formulas. Additive synthesis vs IFFT synthesis pros and cons. Phase.
http://www.iua.upf.es/~xserra/articles/msm/deterministic_synthesis.html

"From Fourier Transform to Wavelet Transform: Basic Concepts". Short-time FT vs Wavelet Transform. http://www.xilinx.com/products/logicore/dsp/fft_to_wavelet.pdf

"Perceptually Smooth Timbral Guides by State-Space Analysis of Phase-Vocoder Parameters." Nicky Bailey and David Cooper, Dept of EE & Dept of Music at University of Leeds. Computer Music Journal, 24:1, pp. 32-42, Spring 2000.

"The Phase Vocoder: A Tutorial." Mark Dolson. Computer Music Journal, Vol 10 No 4, Winter 1986.

"Multirate Additive Synthesis". Desmond K Phillips. Computer Music Journal 23:1, pp. 28-40. Spring 1999.

"Fractal Additive Synthesis via Harmonic-Band Wavelets". Polotti and Evangelista. Computer Music Journal 25:3, pp.22-37, Fall 2001.

"High resolution seismic data analysis by Wavelet Transform and Matching PursuitDecomposition". Wavelet discussion.
http://www.vsl.com/downloads/abstracts/99cn011.pdf

"ANALYSIS OF SPEECH SEGMENTS USING VARIABLE SPECTRAL/TEMPORAL RESOLUTION". Discusses various aspects relevant to our project such as "over-sampled FFT", FFT spectrum resampling / frequency "warping" functions, in order to approximate the frequency resolution properties of human hearing, FFT values being

summed, Wavelets. http://www.asel.udel.edu/icslp/cdrom/vol2/643/a643.pdf

"THE ENGINEER'S ULTIMATE GUIDE TO WAVELET ANALYSIS: THE WAVELET TUTORIAL" by ROBI POLIKAR.
http://www.public.iastate.edu/~rpolikar/WAVELETS/waveletindex.html

"Music and Computers". Online book. Specifically chapter 3 about FFT/IFFT problems and alternatives, and chapter 4 about synthesis.
http://music.dartmouth.edu/~book/MATCpages/tableofcontents.html

"Sonogram" signal-analysis program for Macintosh.
http://www.dfki.de/~clauer/sonogram/

Many Wavelet articles and links at http://cm.bell-labs.com/who/wim/papers/index.html. Such as: "An Overview of Wavelet Based Multiresolution Analysis". http://cm.bell-labs.com/who/wim/papers/papers.html#overview

Fast Fourier Transform FAQ. http://www.dspguru.com/info/faqs/fftfaq.htm

BMP file format & WAV file format - several different places online detailed the header structures.

Some wavelet code. Wavelet-0.4f library, Waili library, ImageLib-0.0.3, ...

Tom Sullivan

## CONCLUSION

Our project delved into research in sound modeling. We were able to use an overlapping windowed Fast Fourier Transform with logarithmic frequency mapping for the sound to image conversion. In image to sound conversion, we wished to find an effective alternative to the Inverse Fast Fourier Transform (because while editing the spectral image the user cannot properly edit the phase information, which the IFFT needs accurate values for in order to not produce clicks), but with more options and flexibility. Additive sinusoid synthesis sacrificed computational speed for compensating for erroneous phase and added the ability to control more parameters during re-synthesis. However, additive sinusoid synthesis which accurately mimics the IFFT is not an extremely well-explored field and so here too we came up with our own algorithms, which still need work in the accuracy department.

Sound files used, code and .exe files of our programs Mazzehc and Chezzam, and high resolution versions of all images shown in this report can be found at:
http://www.contrib.andrew.cmu.edu/~cgraves/551/final/

```
// Chris Graves, Steven Tsou, Spring 2002 18-551 Group 10
// Mazzehc.cpp - opposite of Chezzam; transforms sound.wav to image.bmp
// compiled with Borland C++ 32-bit free compiler v5.5
// currently only works properly with parameters of
//   64 for windowsize, 1024 for getsize, 1024 for n

#include <iostream.h>
#include <stdio.h>
#include <io.h>       //for filelength()
#include <math.h>
#include <vector>
using std::vector;

/* stuff for logarithmic scaling */
// TOPNOTE = C0*2^(1024/(12*8)) = 26.5795 kHz
// so this is the formula we use to get to next semi-semitone (we'll have 8
// pixels per semitone). obviously it's going to spill over because 22050 is
// our max frequency, so just fill the last part of the image with zeros
#define co 16.35159783128745  // note C in octave zero


void rfft(float X[],int N);


/* holds waveform data - header, some key info, and the left & right
   channels (16-bits per) */
struct  waveform {
        char header[44];
        short* right;
        short* left;
        int fs;
        int numchans;
        int bitdepth;
        int datasize;
};

/* holds BMP data - 2 headers; the red, green, & blue channels
   (8-bits per), storage is used to temporarily store each channel
   with more accuracy as it is made from waveform in mazzehc() */
struct  bmp24 {
        char header1[14];
        char header2[40];
        unsigned char* red;
        unsigned char* green;
        unsigned char* blue;
        float* storage;
        int width;
        int height;
        int arrsize; //=width*height
};


/* byterectifier, just makes negatives bytes be 256-value ints.
   not really useful if you used unsigned chars but we used
   signed chars at first and got by with this roundabout method */
int br(char c) {
        if(c<0) return (c+256);
        else return c;
}


/* loads wave file 'infile' into waveform 'wf' */
bool load_wav(char* infile, waveform* wf)
{
        FILE *fi;
        int i=0;
        int fsize=0;
        char temp;

        fi = fopen(infile,"rb");
        if (fi == NULL) {
                printf("ERROR: Can not open %s\n",infile);
                return false;
        }

        fsize = filelength(fileno(fi));              //get total file size
        i = fread(wf->header, 1, 44, fi);

        if (ferror(fi) || feof(fi)) {
                cout << "File read error or wave file too small to have a header even" << endl;
                return false;
```

```
        }
        if(!(wf->header[0] == 'R' && wf->header[1] == 'I' && wf->header[2] == 'F' && wf->header[3] == 'F')) {
                cout << "Not a valid WAV file" << endl;
                return false;
        }
        wf->numchans = br(wf->header[22]) + br(wf->header[23]*256);
        cout << wf->numchans << " channels, ";
        wf->fs = br(wf->header[24]) + br(wf->header[25])*256 + br(wf->header[26])*256*2 + br(wf->header[27])*256*3;
        cout << wf->fs << " samples/s, ";
        wf->bitdepth = br(wf->header[34]) + br(wf->header[35]*256);
        cout << wf->bitdepth << "-bit." << endl;

        unsigned long datasize;
        fseek(fi, 40, SEEK_SET);
        i = fread(&datasize, 4, 1, fi);
        cout << "bytes: " << fsize << " filesize - " << datasize << " data size =\n       "
             << fsize - datasize << " extra bytes of info at end of file." << endl;

        if(fseek(fi, 44, SEEK_SET)) {
                cout << "Only header was readable" << endl;
                return false;
        }

        int multor = 1;
        int reador = 2;
        if(wf->bitdepth == 16) ;
        else if(wf->bitdepth == 8) {  //8-bit
                multor = 256;
        }
        else {
                cout << "Please load only 8- or 16-bit wav files" << endl;
                return false;
        }

        int j = 0;
        int itemp;
        wf->datasize = datasize/4;    //because as we read it in we read in 2 shorts per..
        wf->right = (short *)malloc(wf->datasize*sizeof(short));
        wf->left = (short *)malloc(wf->datasize*sizeof(short));

        while(j < wf->datasize)
        {
                itemp = fread(&i, reador, 1, fi) * multor;
                wf->left[j] = i;
                if(wf->numchans == 2)
                        itemp = fread(&i, reador, 1, fi) * multor;
                wf->right[j] = i;
                j++;
        }

        if(ferror(fi)) cout << "ERROR"<<endl;
        if(feof(fi)) cout << "END OF FILE:" << ftell(fi) << endl;
        fclose(fi);

/* // debugging: uncomment this block to immediately write wav back out to z.wav
        char*   out = "z.wav";
        FILE *fo;
        fo = fopen(out,"wb");
        fwrite(wf->header, 1, 44, fo);
        for(i=0; i<wf->datasize; i++) {
                fwrite(&wf->left[i], 2, 1, fo);
                fwrite(&wf->right[i], 2, 1, fo);
        }
        fclose(fo);
*/

        return true;
}


/* builds a window of size n that is sort of like a hamming
   window but does not use the hamming formula. is just similarly
   shaped and functional */
void do_hammy(float hammy[], int n)
{
        int i;
        int m = n/32;
        for(i=1; i<m+1; i++)
                hammy[i] = (float)((2*i+m)/3)/m;
        for(i=m+1; i<n/2; i++)
                hammy[i] = 1;
```

```
	for(i=n/2; i<=n; i++)
		hammy[i] = hammy[n/2 - (i - n/2)];
//	for(i=0; i<=n; i++)		//uncomment to see #s & make sure you like the shape
//		cout << " " << hammy[i];
}


/* mazzehc() is the bulk of the program - it is the algorithm that
   translates a stereo waveform 'wf' into a 24-bit RGB bitmap 'img'.
   parameters: wf -> waveform data input, img -> bmp24 data output,
   windowsize -> number of samples that one frame/column of image represents,
   getsize -> number of samples to use in the process of building representation
   of window sized windowsize, m -> number of FFT points and height of image,
   phase_switch -> if 1 then store phase in blue channel, if 0 do not store phase,
   lin -> if 1 then store the image with linear frequency spacing (y-axis), if 0
   then store the image with logarithmic (semitonal, specifically) frequency spacing
   by mapping the linear FFT results to logarithmic semitone values. */
void mazzehc(waveform wf, bmp24* img, int windowsize, int getsize, int m, int phase_switch, int lin)
{
	int n = 2*m;
	img->width = m;
	float *windo = (float *)malloc((n+1)*sizeof(float));
	float *hammy = (float *)malloc((n+1)*sizeof(float));
	int i=0, j=0, arroffset=0; //arroffset is for the malloc'd arrays below
	int tmp=0;
	float tmpf, tmpph;
	float re, im;
	float logger;
	int loground;

	img->arrsize = wf.datasize/windowsize*m;
	while (++img->arrsize % getsize != 0);	//get to next multiple of getsize for the arraysize
	img->storage = (float *)malloc(img->arrsize * sizeof(float));
	float* storageph = (float *)malloc(img->arrsize * sizeof(float));

	do_hammy(hammy, n); //build pseudo-hamming window

	for(j=0; j<wf.datasize; j+=windowsize) {
		for(i=0; i<getsize; i++) {
			if(i+j < wf.datasize)
				tmp = wf.left[i+j];
			else tmp = 0;
			windo[i+1] = (float)tmp/32767;	//get a window of getsize into temp
		}
		if(getsize < n)
			for(i=getsize+1; i<=n; i++)
				windo[i] = 0;  //zero pad if necessary
		for(i=1; i<=n; i++)
			windo[i] *= hammy[i];
		rfft(windo, n);			//compute the fft

		for(i=0; i < n/2; i++)
		{
			if(lin == 1) {		//linear freq spacing
				re = windo[i];
				im = windo[n-i];
			}
			else {			//logarithmic freq spacing
				logger = co*(pow(2,((float)i/(12*8))))/((float)wf.fs/(float)n);
				loground = (int)logger;
				logger = logger -	loground;
				if(loground < n) {
					re = windo[loground]*logger + windo[loground+1]*(loground+1-logger);
					im = windo[n-loground]*logger + windo[n-(loground+1)]*(loground+1-logger);
				}
				else {
					re = 0;
					im = 0;
				}
			}

			if(i == 0 || i==n/2-1)
				tmpf = re;  //first value is already magnitude (no imag component)
			else {
				tmpf = sqrt(re*re + im*im);	//magnitude
				tmpph = atan(im/re);		//phase
			}
			img->storage[arroffset] = tmpf;
			storageph[arroffset++] = tmpph;
		}
	}
```

```
img->arrsize = arroffset;  //a double-checker to make sure we have the right arrsize
img->red = (unsigned char *)malloc(img->arrsize * sizeof(char));  //allocate rgb arrays
img->green = (unsigned char *)malloc(img->arrsize * sizeof(char));
img->blue = (unsigned char *)malloc(img->arrsize * sizeof(char));

float minz = 100, maxz = -100;
float minzph = 100, maxzph = -100;
for(j=0; j<img->arrsize; j++) {      //get min & max for scaling purposes
        if(img->storage[j] > maxz) maxz = img->storage[j];
        if(img->storage[j] < minz) minz = img->storage[j];
        if(storageph[j] > maxzph) maxzph = storageph[j];
        if(storageph[j] < minzph) minzph = storageph[j];
}

cout << "maxz: " << maxz << ", minz: " << minz << ". scaling..." << endl;

for(j=0; j<img->arrsize; j++) {      //scale values to 0-255 and store into green
        tmpf = (img->storage[j]-minz)/(maxz-minz) * 256;
        img->green[j] = (char)tmpf;
        if(phase_switch != 0) {              //then store phase
                tmpph = (storageph[j]-minzph)/(maxzph-minzph) * 15;  //lower 4 bits for green phase
                img->blue[j] = tmpph;
        }
}


arroffset=0; //for storage[]
for(j=0; j<wf.datasize; j+=windowsize) {
        for(i=0; i<getsize; i++) {
                if(i+j < wf.datasize)  //FIX
                        tmp = wf.right[i+j];
                else tmp=0;
                windo[i+1] = (float)tmp/32767;       //get a window of getsize into temp
        }
        if(getsize < n)
                for(i=getsize+1; i<=n; i++)
                        windo[i] = 0;  //zero pad if necessary
        for(i=1; i<=n; i++)
                windo[i] *= hammy[i];

        rfft(windo, n);                      //compute the fft

        for(i=0; i < n/2; i++)
        {
                if(lin == 1) {        //use linear freq spacing
                        re = windo[i];
                        im = windo[n-i];
                }
                else {                //use logarithmic freq spacing
                        logger = co*(pow(2,((float)i/(12*8))))/((float)wf.fs/(float)n);
                        loground = (int)logger;
                        logger = logger -      loground;
                        if(loground < n) {
                                re = windo[loground]*logger + windo[loground+1]*(loground+1-logger);
                                im = windo[n-loground]*logger + windo[n-(loground+1)]*(loground+1-logger);
                        }
                        else {
                                re = 0;
                                im = 0;
                        }
                }

                if(i == 0 || i==n/2-1)
                        tmpf = re;  //first value is already magnitude (no imag component)
                else {
                        tmpf = sqrt(re*re + im*im);   //magnitude
                        tmpph = atan(im/re);          //phase
                }
                img->storage[arroffset] = tmpf;
                storageph[arroffset++] = tmpph;
        }
}

minz = 100;
maxz = -100;
minzph = 100;
maxzph = -100;
for(j=0; j<img->arrsize; j++) {      //get min & max for scaling purposes
        if(img->storage[j] > maxz) maxz = img->storage[j];
```

```
                if(img->storage[j] < minz) minz = img->storage[j];
                if(storageph[j] > maxzph) maxzph = storageph[j];
                if(storageph[j] < minzph) minzph = storageph[j];
        }

        cout << "maxz: " << maxz << ", minz: " << minz << ". scaling..." << endl;

        for(j=0; j<img->arrsize; j++) {       //scale and store into red
                tmpf = (img->storage[j]-minz)/(maxz-minz) * 256;
                img->red[j] = (char)tmpf;
                if(phase_switch != 0) {              //and store phase if called for
                        tmpph = (storageph[j]-minzph)/(maxzph-minzph) * 15;  //upper 4 bits for red phase
                        img->blue[j] += ((char)tmpph)<<4;                           //shift it to upper 4 bits
                }
                else
                        img->blue[j] = 0;
        }

        img->height = img->arrsize / img->width;
}


/* saves a BMP file 'outfile' from struct bmp24 'img'.
   note: data in bmp24 must be already upside-down - use
   bmpupsidedown() to make this happen */
void savebmp(bmp24 img, char* outfile)
{
        FILE *fo;
        fo = fopen(outfile,"wb");
        if (fo == NULL) {
                printf("ERROR: Can not write to %s\n",outfile);
                exit(-1);
        }
        fwrite("BM", 1, 2, fo);
        unsigned int data[10];
        data[0]=0;                      //file size - fill in later
        data[1]=0;                      //reserved
        data[2]=54;                     //file offset to raster data
        fwrite(data, 4, 3, fo);

        data[0]=40;                     //size of infoheader
        data[1]=img.width;
        data[2]=img.height;
        fwrite(data, 4, 3, fo);

        char doto[4];
        doto[0]=1;                      //number of planes (always 1)
        doto[1]=0;
        doto[2]=24;                     //24bit
        doto[3]=0;
        fwrite(doto, 1, 4, fo);

        data[0]=0;                      //no compression is 0
        data[1]=0;                      //size of image (compressed) 0 if no compression
        data[2]=0;
        data[3]=0;                      //horiz + vert res in pixels/meter
        data[4]=16777216;               //number of colors actually used
        data[5]=0;                      //number of colors important (0 means all)
        fwrite(data, 4, 6, fo);

        data[0]=0;                      //used now to store image size

        for(int i=0; i<img.arrsize; i++) {
                doto[0] = img.blue[i];  //for some reason it is in this order.. bgr instead of rgb
                doto[1] = img.green[i];
                doto[2] = img.red[i];
                fwrite(doto, 1, 3, fo);
                data[0]+=3;
        }

        cout << "BMP raster data size (bytes): " << data[0] << endl;

        fseek(fo, 2, SEEK_SET);
        fwrite(data, 4, 1, fo);

        fclose(fo);
}


/* makes RGB data upside-down, how BMP likes it */
void bmpupsidedown(bmp24* img)
```

```
{
	int j;

	vector<char> tmpcolumn;
	for(int i=0; i<img->width; i++) {	//for each column
		for(j=0; j<img->height; j++) {  //for each row
			tmpcolumn.push_back( img->green[j*img->width + i] );
		}
		while(j%4 != 0) {
			tmpcolumn.push_back(0);
			img->arrsize++;
			j++;
		}
	}

	img->green = (unsigned char *)malloc(img->arrsize * sizeof(char));

	for(int i=0; i<img->arrsize; i++)
		img->green[i] = tmpcolumn[i];

	tmpcolumn.resize(0, 0);
	for(int i=0; i<img->width; i++) {	//for each column
		for(j=0; j<img->height; j++) {  //for each row
			tmpcolumn.push_back( img->red[j*img->width + i] );
		}
		while(j%4 != 0) {
			tmpcolumn.push_back(0);
			j++;
		}
	}

	img->red = (unsigned char *)malloc(img->arrsize * sizeof(char));

	for(int i=0; i<img->arrsize; i++)
		img->red[i] = tmpcolumn[i];

	tmpcolumn.resize(0, 0);
	for(int i=0; i<img->width; i++) {	//for each column
		for(j=0; j<img->height; j++) {  //for each row
			tmpcolumn.push_back( img->blue[j*img->width + i] );
		}
		while(j%4 != 0) {
			tmpcolumn.push_back(0);
			j++;
		}
	}

	img->blue = (unsigned char *)malloc(img->arrsize * sizeof(char));

	for(int i=0; i<img->arrsize; i++)
		img->blue[i] = tmpcolumn[i];

	int temp = j;
	img->height = img->width;
	img->width = temp;

}



/***** MAIN *****/

int main(int argc, char **argv)
{
	bmp24 img;
	waveform wf;
	char* inputwav = argv[1];
	char* outputbmp = argv[2];
	int windosize = 64;
	int getsize = 1024;
	int n = 1024;
	int phase_switch;
	int lin=1;

	if(argc < 5) {
		cout << endl << "cmd line usage: mazzehc.exe [infile] [outfile] [1|0 phase] [1|0 lin/log]" << endl;
		cout << "input wav filename: ";
		char str[80];
		scanf ("%s", str);
		inputwav = str;
		char str2[80];
```

```
                cout << "output bmp filename: ";
                scanf ("%s", str2);
                outputbmp = str2;
                cout << "phase saved in bmp on[1] or off[0]: ";
                scanf ("%d", &phase_switch);
                cout << "frequency spacing linear[1] or logarithmic[0]: ";
                scanf ("%d", &lin);
        }
        else {
                phase_switch = atoi(argv[3]);
                lin = atoi(argv[4]);
        }


        cout << "sonogram parameters - window size (#samples per frame) = " << windosize << endl
                << "   get size (#samples used for fft data per frame) = " << getsize << endl
                << "     (getsize - windowsize = overlap amount)" << endl
                << "   n (#fft points aka pixels in y-axis. if > getsize it is zeropadded) = " << n << endl;

        if(!load_wav(inputwav, &wf))
                exit(-1);

        mazzehc(wf, &img, windosize, getsize, n, phase_switch, lin);       //do_transformation and place in img
        bmpupsidedown(&img);                                  //bmp's are stored upside down so apply that to img

        savebmp(img, outputbmp);
        cout << inputwav << " sonogrammed into " << outputbmp << endl << endl;

        return 0;
}
```

```
/*************************************************************************
* rfft(float X[],int N)                                                  *
*     A real-valued, in-place, split-radix FFT program                   *
*     Decimation-in-time, cos/sin in second loop                         *
*     Input: float X[1]...X[N] (NB Fortran style: 1st pt X[1] not X[0]!)  *
*     Length is N=2**M (i.e. N must be power of 2--no error checking)    *
*     Output in X[1]...X[N], in order:                                   *
*           [Re(0), Re(1),..., Re(N/2), Im(N/2-1),..., Im(1)]            *
*                                                                        *
* Original Fortran code by Sorensen; published in H.V. Sorensen, D.L. Jones,*
* M.T. Heideman, C.S. Burrus (1987) Real-valued fast fourier transform   *
* algorithms.  IEEE Trans on Acoustics, Speech, & Signal Processing, 35, *
* 849-863.  Adapted to C by Bill Simpson, 1995  wsimpson@uwinnipeg.ca    *
*************************************************************************/

void rfft(float X[],int N)
{
int I,I0,I1,I2,I3,I4,I5,I6,I7,I8, IS,ID;
int J,K,M,N2,N4,N8;
float A,A3,CC1,SS1,CC3,SS3,E,R1,XT;
float T1,T2,T3,T4,T5,T6;

M=(int)(log(N)/log(2.0));              /* N=2^M */

/* ----Digit reverse counter----------------------------------------- */
J = 1;
for(I=1;I<N;I++)
        {
        if (I<J)
                {
                XT    = X[J];
                X[J]  = X[I];
                X[I]  = XT;
                }
        K = N/2;
        while(K<J)
                {
                J -= K;
                K /= 2;
                }
        J += K;
        }

/* ----Length two butterflies---------------------------------------- */
```

```
IS = 1;
ID = 4;
do
        {
        for(I0 = IS;I0<=N;I0+=ID)
                {
                I1    = I0 + 1;
                R1    = X[I0];
                X[I0] = R1 + X[I1];
                X[I1] = R1 - X[I1];
                }
        IS = 2 * ID - 1;
        ID = 4 * ID;
        }while(IS<N);
/* ----L shaped butterflies---------------------------------------------- */
N2 = 2;
for(K=2;K<=M;K++)
        {
        N2    = N2 * 2;
        N4    = N2/4;
        N8    = N2/8;
        E     = (float) 6.2831853071719586f/N2;
        IS    = 0;
        ID    = N2 * 2;
        do
                {
                for(I=IS;I<N;I+=ID)
                        {
                        I1 = I + 1;
                        I2 = I1 + N4;
                        I3 = I2 + N4;
                        I4 = I3 + N4;
                        T1 = X[I4] +X[I3];
                        X[I4] = X[I4] - X[I3];
                        X[I3] = X[I1] - T1;
                        X[I1] = X[I1] + T1;
                        if(N4!=1)
                                {
                                I1 += N8;
                                I2 += N8;
                                I3 += N8;
                                I4 += N8;
                                T1 = (X[I3] + X[I4])*.707106781186547524f;
                                T2 = (X[I3] - X[I4])*.707106781186547524f;
                                X[I4] = X[I2] - T1;
                                X[I3] = -X[I2] - T1;
                                X[I2] = X[I1] - T2;
                                X[I1] = X[I1] + T2;
                                }
                        }
                        IS = 2 * ID - N2;
                        ID = 4 * ID;
                }while(IS<N);
        A = E;
        for(J= 2;J<=N8;J++)
                {
                A3 = 3.0 * A;
                CC1   = cos(A);
                SS1   = sin(A);  /*typo A3--really A?*/
                CC3   = cos(A3); /*typo 3--really A3?*/
                SS3   = sin(A3);
                A = (float)J * E;
                IS = 0;
                ID = 2 * N2;
                do
                        {
                        for(I=IS;I<N;I+=ID)
                                {
                                I1 = I + J;
                                I2 = I1 + N4;
                                I3 = I2 + N4;
                                I4 = I3 + N4;
                                I5 = I + N4 - J + 2;
                                I6 = I5 + N4;
                                I7 = I6 + N4;
                                I8 = I7 + N4;
                                T1 = X[I3] * CC1 + X[I7] * SS1;
                                T2 = X[I7] * CC1 - X[I3] * SS1;
                                T3 = X[I4] * CC3 + X[I8] * SS3;
                                T4 = X[I8] * CC3 - X[I4] * SS3;
                                T5 = T1 + T3;
```

```
                                T6 = T2 + T4;
                                T3 = T1 - T3;
                                T4 = T2 - T4;
                                T2 = X[I6] + T6;
                                X[I3] = T6 - X[I6];
                                X[I8] = T2;
                                T2    = X[I2] - T3;
                                X[I7] = -X[I2] - T3;
                                X[I4] = T2;
                                T1    = X[I1] + T5;
                                X[I6] = X[I1] - T5;
                                X[I1] = T1;
                                T1    = X[I5] + T4;
                                X[I5] = X[I5] - T4;
                                X[I2] = T1;
                                }
                        IS = 2 * ID - N2;
                        ID = 4 * ID;
                        }while(IS<N);
                }
        }
return;
}
```

```
// Chris Graves, Steven Tsou, Spring 2002 18-551 Group 10
// Chezzam.cpp - opposite of Mazzehc; transforms image.bmp to sound.wav
// compiled with Borland C++ 32-bit free compiler v5.5
// currently requires 1024-pixel tall images - may break with others

#include <iostream.h>
#include <stdio.h>
#include <io.h>       //for filelength()
#include <math.h>

#define PI 3.14159265358979
#define co 16.35159783128745  // note C in octave zero

void rfft(float X[],int N);

struct  waveform {
        char header[44];
        short* right;
        short* left;
        int fs;
        int numchans;
        int bitdepth;
        int datasize;
};


struct  bmp24 {
        char header1[14];
        char header2[40];
        unsigned char* red;
        unsigned char* green;
        unsigned char* blue;
        unsigned char* storage;
        int width;
        int height;
        int arrsize; //=width*height
};


/* loads a BMP from file 'infile' into bmp24 'img' */
bool load_bmp(char* infile, bmp24* img) {
        int fsize;
        FILE *fi;
        fi=fopen(infile, "rb");
        if (fi == NULL) {
                printf("ERROR: Can not open %s\n",infile);
                return false;
        }
        fsize = filelength(fileno(fi));                     //get total file size

        unsigned char bmptest[4];

        fread(bmptest, 1, 2, fi);

        if(!(bmptest[0] == 'B' && bmptest[1] == 'M')) {
                cout << "not a valid BMP file." << endl;
                return false;
        }

        int data[10];

        fread(data, 4, 3, fi);

        int datasize = data[0];
        int dataoffset = data[2];

        fread(data, 4, 3, fi);

        //int sizeofheader2 = data[0];
        img->width = data[1];
        img->height = data[2];

        cout << "rasterdata size: " << datasize << endl
                 << "width: " << img->width << ", height: " << img->height << endl;

        fread(bmptest, 1, 4, fi);

        if(bmptest[2] != 24) {
                cout << "not a 24bit bmp: " << bmptest[2] << endl;
                return false;
        }
```

```
        else
                cout << "24-bit; good choice." << endl;

        cout << "loading " << infile << " into memory..." << endl;
        if(datasize > 2000000)
                cout << "you gave us a big file; now give us your patience..." << endl;

        img->red = (unsigned char *)malloc(img->width*img->height);
        img->green = (unsigned char *)malloc(img->width*img->height);
        img->blue = (unsigned char *)malloc(img->width*img->height);
        img->storage = (unsigned char *)malloc(img->width*img->height);

        if(datasize < 0) {
                cout << "Data size is negative! Using filesize instead. Check your BMPsaver." << endl;
                datasize = fsize - dataoffset;
        }

        img->arrsize = datasize/3;

        fseek(fi, dataoffset, SEEK_SET); //start of data


        for(int i=0; i<datasize/3; i++)
        {
                fread(bmptest, 1, 3, fi);
                img->blue[i] = bmptest[0];
                img->green[i] = bmptest[1];
                img->red[i] = bmptest[2];
        }

        fclose(fi);

        return true;
}


/* bmprearrange() rearranges columns to rows in 'img' -
   use before chezzam() because chezzam() needs frame 1
   to be row 1. done for simplicity within chezzam(). */
void bmprearrange( bmp24* img) {
        int i, j, k;

        k=0;
        for(j=0; j<img->arrsize; j++)
                img->storage[j] = img->red[j];       //copy to temporary storage

        for(i=0; i<img->width; i++) {
                for(j=0; j<img->height; j++) {       //rearrange columns to rows
                        img->red[k++] = img->storage[j*img->width + i];
                }
        }

        k=0;
        for(j=0; j<img->arrsize; j++)
                img->storage[j] = img->green[j];     //copy to temporary storage

        for(i=0; i<img->width; i++) {
                for(j=0; j<img->height; j++) {       //rearrange columns to rows
                        img->green[k++] = img->storage[j*img->width + i];
                }
        }

        k=0;
        for(j=0; j<img->arrsize; j++)
                img->storage[j] = img->blue[j];      //copy to temporary storage

        for(i=0; i<img->width; i++) {
                for(j=0; j<img->height; j++) {       //rearrange columns to rows
                        img->blue[k++] = img->storage[j*img->width + i];
                }
        }

}


void chezzam(bmp24 img, waveform *wf, int smpsperframe, int lin) {
        float tempmag, tempmagnext, tempscaler, freq;
        //used to determine if a sine restarts or continues (right channel, left channel)
        int *sincountersr = (int *)malloc(img.height*sizeof(int));
        int *sincountersl = (int *)malloc(img.height*sizeof(int));
        //used to temporarily store 1 frame of audio (left and right)
```

```
	short *audioframer = (short *)malloc(smpsperframe*sizeof(short));
	short *audioframel = (short *)malloc(smpsperframe*sizeof(short));
	// allocate memory for left & right channels of waveform - we already know how large it lwl be
	wf->right = (short *)malloc(img.width*smpsperframe*sizeof(short));
	wf->left = (short *)malloc(img.width*smpsperframe*sizeof(short));
	wf->datasize = img.width*smpsperframe*sizeof(short)*2;
				//width still is how many frames even though it is rotated
	wf->fs = 44100;
	wf->bitdepth = 16;
	wf->numchans = 2;

	for(int i=0; i<img.height; i++) {	//fill sincounters with zeros
		sincountersr[i] = 0;
		sincountersl[i] = 0;
	}

	int audiooffset = 0;
	int i;

	cout << "Chezzaming.";

	for(int j=0; j<img.width-1; j++) { //for each row
		for(int k=0; k<smpsperframe; k++) {
			audioframer[k] = 0;	//fill audioframe with zeros for adding in a minute
			audioframel[k] = 0;	//fill audioframe with zeros for adding in a minute
		}
		for(i=0; i<img.height; i++) { //for each column


			// right (red)
			tempmag = (float)(img.red[i+j*img.height]) / 256;
			tempmagnext = (float)(img.red[i+(j+1)*img.height]) / 256;
			if(lin == 1)					//linear frequency spacing
				freq = i*((wf->fs/2)/(img.height)); //  possibly /2 also for some reason
			else {						//logarithmic pitch then
				freq = (int)co*(pow(2,((float)i/(12*8))));
				if (freq>22050) freq=0;
			}
			//got those now interpolate
// this would speed up a lot when neighboring pixels are black - no calculating sines, but does not work properly
//			if(tempmag == 0 && tempmagnext == 0)
//				for(int k=0; k<smpsperframe; k++) { //if there are no pixels don't calc sine - make
silent (speedup)
//					audioframer[k] = 0; }
//			else
				for(int k=0; k<smpsperframe; k++) { //for audio frame size, generate audioframer
					tempscaler = (tempmag + k*(tempmagnext-tempmag)/smpsperframe) *
						sin(2.0*PI*sincountersr[i]/wf->fs*freq); //later +temppha goes inside
sine
					sincountersr[i]++;
					audioframer[k] += (int)(tempscaler*32768/12);
				}


			if( tempmagnext == 0 ) sincountersr[i] = 0; //reset counter for this freq if current sine
has stopped


			//left (green)
			tempmag = (float)(img.green[i+j*img.height]) / 256;
			tempmagnext = (float)(img.green[i+(j+1)*img.height]) / 256;
			if(lin == 1)					//linear frequency spacing
				freq = i*((wf->fs/2)/(img.height)); //  possibly /2 also for some reason
			else {						//logarithmic pitch then
				freq = (int)co*(pow(2,((float)i/(12*8))));
				if (freq>22050) freq=0;
			}
			//got those now interpolate
// this would speed up a lot when neighboring pixels are black - no calculating sines, but does not work properly
//			if(tempmag == 0 && tempmagnext == 0)
//				for(int k=0; k<smpsperframe; k++) { //if there are no pixels don't calc sine - make
silent (speedup)
//					audioframel[k] = 0; }
//			else
				for(int k=0; k<smpsperframe; k++) { //for audio frame size, generate...
					tempscaler = (tempmag + k*(tempmagnext-tempmag)/smpsperframe) *
						sin(2.0*PI*sincountersl[i]/wf->fs*freq); //later +temppha goes inside
sine
					sincountersl[i]++;
					audioframel[k] += (int)(tempscaler*32768/12);
				}
```

```
                        if( tempmagnext == 0 ) sincountersl[i] = 0; //reset counter for this freq if current sine
has stopped


                }
                for(int k=0; k<smpsperframe; k++) {
                        wf->right[audiooffset] = audioframer[k];
                        wf->left[audiooffset] = audioframel[k];
                        audiooffset++;
                }
                cout << ".";
        }
        cout << "done." << endl;
}


/* saves waveform 'wf' to wave file 'outputwav' */
void savewav(waveform wf, char* outputwav) {
        FILE* fo = fopen(outputwav,"wb");  /* open wav file for writing now */
        if (fo == NULL) {
                printf("ERROR: Can not write to %s\n",outputwav);
                exit(-1);
        }

        cout << "saving wav..." << endl;

        int totalbytes;
        int databytes;

        int data[2];
        fwrite("RIFF", 1, 4, fo);
        data[0]=0;     //filesize, fill in later (totalbytes-8)
        fwrite(data, 4, 1, fo);
        fwrite("WAVEfmt ", 1, 8, fo);
        data[0]=16; //16 for PCM
        fwrite(data, 4, 1, fo);
        data[0]=1;     //1 for PCM
        fwrite(data, 2, 1, fo);
        data[0]=2;     //# of channels
        fwrite(data, 2, 1, fo);
        data[0]=44100; //samplerate
        fwrite(data, 4, 1, fo);
        data[0]=44100*2*2;     //bytespersecond (samplerate*channels*bytespersample(bitspersample/8)
        fwrite(data, 4, 1, fo);
        data[0]=0;     //bytes per capture??
        fwrite(data, 2, 1, fo);
        data[0]=16;    //bits per sample
        fwrite(data, 2, 1, fo);
        fwrite("data", 1, 4, fo);
        data[0]=0;     //bytes in data
        fwrite(data, 4, 1, fo);
        totalbytes = 44;
        fseek(fo, 44, SEEK_SET); //totalbytes

        short doto[2];
        for(int i=0; i<wf.datasize/2/2; i++) {       // /2(stereo) /2(sizeof short)
                doto[0] = wf.left[i];
                doto[1] = wf.right[i];
                fwrite(doto, 2, 2, fo);              //write L (maybe swapped)
                totalbytes += 4;
                databytes += 4;
        }

        data[0]=totalbytes;
        fseek(fo, 4, SEEK_SET); //totalbytes
        fwrite(data, 4, 1, fo);

        data[0]=databytes;
        fseek(fo, 40, SEEK_SET); //databytes
        fwrite(data, 4, 1, fo);


}




/*** MAIN ***/

int main(int argc, char **argv)
```

```
{
	char* inputbmp = argv[1];
	char* outputwav = argv[2];
	int smpsperframe;
	bmp24 img;
	waveform wf;
	int lin=1;

	if(argc < 5) {
		cout << endl << "cmd line usage: chezzam.exe [infile] [outfile] [windosize(64ish)] [freq spacing
lin(1)|log (0)]" << endl;
		cout << "input bmp filename: ";
		char str[80];
		scanf ("%s", str);
		inputbmp = str;
		char str2[80];
		cout << "output wav filename: ";
		scanf ("%s", str2);
		outputwav = str2;
		cout << "windosize [64]: ";
		scanf ("%d", &smpsperframe);
		cout << "frequency spacing linear[1] or logarithmic[0]: ";
		scanf ("%d", &lin);
	}
	else {
		smpsperframe = atoi(argv[3]);
		lin = atoi(argv[4]);
	}

	if(!load_bmp(inputbmp, &img)) {
		cout << "can't load bmp." << endl;
		exit(-1);
	}

	bmprearrange(&img); //rows to columns for quicker sending
	chezzam(img, &wf, smpsperframe, lin);

	savewav(wf, outputwav);

	cout << inputbmp << " additively synthesized into " << outputwav << endl << endl;

	return 0;
}
```

```
// Chris Graves, Steven Tsou, Spring 2002 18-551 Group 10
// filterpc.c - based on lab 3 filterpc.c
// loads BMP and sends a columns of it to EVM when requested

/************************************************/
/* 18-551 Lab 3                                                 */
/* filterpc.c: Communication routines for filtering         */
/*             (PC side)                                        */
/* Author: <pwb@andrew.cmu.edu>                                 */
/*  Last Modified: '2/16/00 10:51:46 AM'                        */
/************************************************/


#include <iostream.h>
#include <stdio.h>
#include <windows.h>
#include <io.h>       //for filelength()
#include "evm6xdll.h"

/* Structure for holding transfer information */
struct transfer_s {
       void *buffer;
       unsigned long size;
       unsigned long command;
};

struct  bmp24 {
       char header1[14];
       char header2[40];
       unsigned char* red;
       unsigned char* green;
       unsigned char* blue;
       unsigned char* storage;
       int width;
       int height;
       int arrsize; //=width*height
};


/* Function prototypes */
int wait_request(struct transfer_s *ts);
int send_data(struct transfer_s *ts, void *local_buf);
int get_data(struct transfer_s *ts, void *local_buf);
void hpi_write_word(ULONG addr, ULONG data);
bool load_bmp(char* infile, bmp24* img);
void bmprearrange(bmp24* img);
void initializeEVMstuff();
int sendcolsize(bmp24 img);

/* Global vars */
unsigned char* frame;
//unsigned char frame1[F_SIZE];      /* Input */
//short int result[BUFFER_LEN];                             /* Output */


HANDLE hBd    = NULL;
LPVOID hHpi = NULL;
HANDLE h_event;



int main(int argc, char **argv) {
       int program_exit=0, i=0;
       char s_buffer[80];
       char temp[1024];
       struct transfer_s ts;
       char* inputbmp = argv[1];
       bmp24 img;


       if(!argv[1]) {
              printf("cmd line usage: filterpc.exe [infile.bmp]\ninput wav filename: ");
              char str[80];
              scanf ("%s", str);
              inputbmp = str;
       }


       if(!load_bmp(inputbmp, &img)) {
              cout << "can't load bmp." << endl;
              exit(-1);
```

```
	}

	bmprearrange(&img); //rows to columns for quicker sending

	/* Open the board */
	hBd = evm6x_open(0, 0);

	if ( hBd == INVALID_HANDLE_VALUE ) {
		fprintf(stderr, "Couldn't open board\n");
		exit(1);
	}
	fprintf(stderr, "Opened connection to board...\n");

	/* Also open connection to HPI */
    hHpi = evm6x_hpi_open(hBd);
    if ( hHpi == NULL ) exit(4);

	/* Set DMA AUX priority greater than CPU priority, so we
		don't lock the PCI bus */
    hpi_write_word(0x01840070 /*Addr*/, 0x00000010 /*Data*/);

	/* Reset the mailboxes and FIFO */
	hpi_write_word(0x0170003C, 0x0e000000);

	/* Setup a windows event so we don't have to poll for incoming
		messages */
	evm6x_clear_message_event(hBd);
	sprintf( s_buffer, "%s%d",EVM6X_GLOBAL_MESSAGE_EVENT_BASE_NAME, 0);
	h_event = OpenEvent( SYNCHRONIZE, FALSE, s_buffer );


	/***  Main loop... Waits for messages from the EVM and does a transfer
		depending on the value of the message received ***/
	int colsize = sendcolsize(img);
	frame = (unsigned char *)malloc(colsize);
	int k=0;

	while(!program_exit) {

		//prepare frame for sending
		for(i=0; i<colsize/3; i++)
			frame[i] = img.green[i+k];
		for(i=colsize/3; i<2*colsize/3; i++)
			frame[i] = img.green[i+k];
		for(i=2*colsize/3; i<colsize; i++)
			frame[i] = img.green[i+k];
		k += colsize/3; //k is offset into red/green/blue arrays so it incs by 1/3 of colsize

//		cout << "frame prepped. k=" << k << endl; //<< ", img.datasize=" << img.width*img.height << endl;

		wait_request(&ts);

//		fprintf(stderr, "Transfer request: CMD %x, SIZE %i, ADDRESS %x\n", ts.command, ts.size, ts.buffer);

		/* Now based on the command that was sent, do something */
		switch(ts.command) {
		case(0x01): /* Send frame */
			if(ts.size != colsize) fprintf(stderr, "Wrong size!!!\n");
			if(k > img.width*img.height) {
				fprintf(stderr, "Image done, sending last frame & exiting\n");
				ts.command = 4;
			}
			send_data(&ts, frame);
//			cout << "DATA SENT!" << endl;
			break;
		case(0x03): /* Print string */
			get_data(&ts, temp);
			printf("%s\n", temp);
			break;
		case(0x04): /* Exit program */
			program_exit = 1;
			break;
		default:
			fprintf(stderr, "Unknown command\n");
			break;
		}
	}

	/* Clean up and exit */
	if (!evm6x_hpi_close(hHpi)) exit(9);
	if (!evm6x_close(hBd)) exit(16);
```

```
	return(0);
}



/* Waits for windows event signalling incoming message.  Then reads
	address from mailbox 1, size from mailbox 2, and command from
	mailbox 3 */
int wait_request(struct transfer_s *ts)
{
	/* wait for event signaling a message from the DSP */
	WaitForSingleObject( h_event, INFINITE );

	if(!evm6x_retrieve_message(hBd, (unsigned long *)&ts->buffer))
		fprintf(stderr, "Error retrieving 1...\n");
	if(!evm6x_mailbox_read(hBd, 2, (unsigned long *)&ts->size))
		fprintf(stderr, "Error retrieving 2...\n");
	if(!evm6x_mailbox_read(hBd, 3, (unsigned long *)&ts->command))
		fprintf(stderr, "Error retrieving 3...\n");

	return(0);
}

/* Size and address are given in struct ts.  local_buf is the address of
	the PC buffer to send */
int send_data(struct transfer_s *ts, void *local_buf)
{
	unsigned long int ulLength = ts->size;

	/* Write the data to EVM memory*/
	if (!evm6x_hpi_write(hHpi, (PULONG)local_buf, (PULONG)&ulLength, (ULONG)ts->buffer)) {
		fprintf(stderr, "HPI write error\n");
		exit(1);
	}
	if (ulLength != ts->size) {
		fprintf(stderr, "HPI only wrote %i bytes\n");
		exit(1);
	}

	/* Use a message to signal the EVM that the transfer is done */
	if (!evm6x_send_message(hBd, (PULONG)&ts->command)) {
		fprintf(stderr, "Send message error!\n");
		exit(1);
	}
	return(0);
}


int sendcolsize(bmp24 img) {
	unsigned long colsize[1];
	colsize[0] = img.height*3;

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

	if(!evm6x_retrieve_message(hBd, &tempp)) {
		evm6x_close(hBd);
		fprintf(stderr, "Error retrieving message from EVM\n");
		getchar();
		exit(1);
	}
	fprintf(stderr, "Received sync message: %i.  Writing...\n", tempp);

	tempp = sizeof(long);

	if (!evm6x_write(hBd, colsize, &tempp)) {
		evm6x_close(hBd);
		fprintf(stderr, "Couldn't write to EVM\n");
		getchar();
		exit(1);
	}
	fprintf(stderr, "Sent colsize.\n");
	return (int)colsize[0];
}



/* loads BMP from file 'infile' into struct bmp24 'img' */
```

```
bool load_bmp(char* infile, bmp24* img) {
	int fsize;
	FILE *fi;
	fi=fopen(infile, "rb");
	if (fi == NULL) {
		printf("ERROR: Can not open %s\n",infile);
		return false;
	}
	fsize = filelength(fileno(fi));                     //get total file size

	unsigned char bmptest[4];

	fread(bmptest, 1, 2, fi);

	if(!(bmptest[0] == 'B' && bmptest[1] == 'M')) {
		cout << "not a valid BMP file." << endl;
		return false;
	}

	int data[10];

	fread(data, 4, 3, fi);

	int datasize = data[0];
	int dataoffset = data[2];

	fread(data, 4, 3, fi);

	//int sizeofheader2 = data[0];
	img->width = data[1];
	img->height = data[2];

	cout << "rasterdata size: " << datasize << endl
		 << "width: " << img->width << ", height: " << img->height << endl;

	fread(bmptest, 1, 4, fi);

	if(bmptest[2] != 24) {
		cout << "not a 24bit bmp: " << bmptest[2] << endl;
		return false;
	}
	else
		cout << "24-bit; good choice." << endl;

	cout << "loading " << infile << " into memory..." << endl;
	if(datasize > 2000000)
		cout << "you gave us a big file; now give us your patience..." << endl;

	img->red = (unsigned char *)malloc(img->width*img->height);
	img->green = (unsigned char *)malloc(img->width*img->height);
	img->blue = (unsigned char *)malloc(img->width*img->height);
	img->storage = (unsigned char *)malloc(img->width*img->height);

	if(datasize < 0) {
		cout << "Data size is negative! Using filesize instead. Check your BMPsaver." << endl;
		datasize = fsize - dataoffset;
	}

	img->arrsize = datasize/3;

	fseek(fi, dataoffset, SEEK_SET); //start of data


	for(int i=0; i<datasize/3; i++)
	{
		fread(bmptest, 1, 3, fi);
		img->blue[i] = bmptest[0];
		img->green[i] = bmptest[1];
		img->red[i] = bmptest[2];
	}

	fclose(fi);

	return true;
}



/* bmprearrange() rearranges columns to rows in 'img' -
   use before EVM side synthesis (chezzamming) because function that
   transfers columns to EVM needs frames (ex-columns) to be store in
```

```
   rows. done for simplicity. */
void bmprearrange( bmp24* img) {
	int i, j, k;

	k=0;
	for(j=0; j<img->arrsize; j++)
		img->storage[j] = img->red[j];		//copy to temporary storage

	for(i=0; i<img->width; i++) {
		for(j=0; j<img->height; j++) {		//rearrange columns to rows
			img->red[k++] = img->storage[j*img->width + i];
		}
	}

	k=0;
	for(j=0; j<img->arrsize; j++)
		img->storage[j] = img->green[j];	//copy to temporary storage

	for(i=0; i<img->width; i++) {
		for(j=0; j<img->height; j++) {		//rearrange columns to rows
			img->green[k++] = img->storage[j*img->width + i];
		}
	}

	k=0;
	for(j=0; j<img->arrsize; j++)
		img->storage[j] = img->blue[j];		//copy to temporary storage

	for(i=0; i<img->width; i++) {
		for(j=0; j<img->height; j++) {		//rearrange columns to rows
			img->blue[k++] = img->storage[j*img->width + i];
		}
	}

}
```

```
/* Same format as send_data */
int get_data(struct transfer_s *ts, void *local_buf)
{
	unsigned long int ulLength = ts->size;

	/* Read data from EVM memory */
    if (!evm6x_hpi_read(hHpi, (PULONG)local_buf, (PULONG)&ulLength, (ULONG)ts->buffer)) {
		fprintf(stderr, "HPI write error\n");
		exit(1);
	}
	if (ulLength != ts->size) {
		fprintf(stderr, "HPI only wrote %i bytes\n");
		exit(1);
	}

	/* Signal EVM that transfer is done */
	if (!evm6x_send_message(hBd, (PULONG)&ts->command)) {
		fprintf(stderr, "Send message error!\n");
		exit(1);
	}
	return(0);
}

/* Write a single 32-bit word to EVM memory */
void hpi_write_word(ULONG addr, ULONG data)
{
	ULONG ulLength = 4;

	if (!evm6x_hpi_write(hHpi, &data, &ulLength, addr)) {
		fprintf(stderr, "Error writing word via HPI\n");
		exit(1);
	}
	if (ulLength != 4 ) {
		fprintf(stderr, "Error writing word via HPI\n");
		exit(1);
	}
}
```

```
#ifdef LOAD_FILE
/* Load the .out file and run the program.  Code Composer must not
       be running */
int load_file(HANDLE hBd, LPVOID hHpi)
{
  if ( !evm6x_reset_board(hBd) ) exit(2);
  if ( !evm6x_reset_dsp(hBd,HPI_BOOT) ) exit(3);
  if ( !evm6x_init_emif(hBd, hHpi) ) exit(5);

  /* Load program */
  if (!evm6x_coff_load(hBd,hHpi,EVM_FILE,0,0,0)) exit(8);

  /* Set HPI priority and reset mailboxes */
  hpi_write_word(0x01840070 /*Addr*/, 0x00000010 /*Data*/ );
  hpi_write_word(0x0170003C, 0x0e000000);

  if (!evm6x_unreset_dsp(hBd)) exit(10);
  if (!evm6x_set_timeout(hBd,1000)) exit(11);
       return(0);
}
#endif
```

```
// Chris Graves, Steven Tsou, Spring 2002 18-551 Group 10
// filterevm.c - based on lab 3 filterevm.c
// EVM code composer code, receives columns/frames of image data
// from filterpc.c and transforms it into audio
// not fast enough to be realtime, but does place it in the output
// buffer (double-buffered actually), and writes a wav-file from
// that same buffer (pointed to by buffwrite*)

/*********************************************************/
/* 18-551 Lab 3                                                  */
/* filterevm.c: skeleton code for part II                  */
/*                                                               */
/* Author: <pwb@andrew.cmu.edu>                            */
/*********************************************************/

#include <stdio.h>
#include <stdlib.h>

#include <common.h>
#include <board.h>          /* EVM library */
#include <pci.h>            /* PCI communication library */
#include <dma.h>

#include <mcbsp.h>              /* mcbsp devlib */
#include <mcbspdrv.h>           /* mcbsp driver */
#include <codec.h>              /* codec library */
#include <mathf.h>              /* math library */
#include <intr.h>               /* interrupt library */
#include <linkage.h>

/////this is all related to sound output
#define SAMPLE_RATE 44100
#define BUFFER_LEN 64
#define AMP 10000

int xindex=2;                   /* Index for transmission ISR */
int *buffwrite;
int *buffer;       /* write ptr for buffer1 & buffer2*/
int buffer1[BUFFER_LEN];       /* Memory buffer for samples */
int buffer2[BUFFER_LEN];       /* Memory buffer for samples */
////////////////////////////////////

unsigned char goahead = 0;
int transfer_complete = 0;
int i, rev_i, devf;
unsigned char *frameA; //ptr to frame1 or frame2
unsigned char *frameB; //ptr to frame1 or frame2
unsigned char *frame1;
unsigned char *frame2;
int totalbytes=0;      /* kept track of for .wav header */
int databytes=0;       /* kept track of for .wav header */
int *counters;

/* Function prototypes */
int request_transfer(void *buf, int size, int command);
int wait_transfer();
interrupt void xmitISR(void);
void synthesize(unsigned char* frameA, unsigned char* frameB, int colsize); /* synthesizes from frameA to frameB
(aka frame1 to frame2 or frame2 to frame1) */
void addtowav(FILE *fo, int *buffwrite);
void startwavwrite(FILE *fo);
void finishwav(FILE *fo);

void callbacker(int status) {
       transfer_complete = 1;
}


int main(void)
{
  FILE *fo;
  char tmp[256];
  int colsize;
  int done=0;
  int i;

  Mcbsp_dev dev;                /* Serial port device */

  char *outfile = "zout.wav";
  fo = fopen(outfile,"wb");  /* open wav file for writing now */
  if (fo == NULL) {
```

```
      printf("ERROR: Can not write to %s\n",outfile);
      exit(-1);
  }


  evm_init();               /* Initialize the board */
  printf("initialized\n");
  pci_driver_init();        /* Call before using any PCI code */

  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 */

  /******************** configure 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(SAMPLE_RATE, 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,FALSE,LINE_SEL));
  EXIT_ERROR(codec_adc_control(RIGHT,0.0,FALSE,LINE_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));

  /***************** setup interrupt routines ******************/
  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);

  /* 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 */

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




  DMA_AUXCR = 0x00000010;     /* Set priority of HPI over CPU to avoid crashing */

  devf = pci_fifo_open();    /* Open FIFO */
  printf("fifo opened\n");

  pci_message_sync_send(1, 1);
  printf("sent sync\n");

  pci_fifo_sync_receive(devf, (unsigned int *)&colsize, sizeof(int));
  printf("colsize received %d\n",colsize);

  /* Allocate frame and result in external memory */
  frame1 = (unsigned char *)malloc(colsize);
  frame2 = (unsigned char *)malloc(colsize);
  counters = (int *)malloc(colsize/3);
  if(frame1==NULL || frame2==NULL) {
    printf("Couldn't allocate memory...\n");
    exit(1);
  }


  startwavwrite(fo);
  printf("wrote wav header");
```

```
  /***** MAIN LOOP *****/
  for(i=0; i<colsize/3; i++) {
       counters[i] = 0;
  }
  i=0;
  request_transfer(frame1, colsize, 0x01); /* do it once before loop to get frame1 filled */
  done = wait_transfer();   /* Wait until last transfer is done before starting a new one */

  frameA = frame2; /* pointer pts to frame2 - fill that next */
  buffer = buffer1; /* point playback buffer to buffer1 */
  while(done != 4) {
         request_transfer(frameA, colsize, 0x01); /* ask for frame2 from pc */
         done = wait_transfer();   /* Wait until last transfer is done before starting a new one */
         if(frameA == frame2) {    /* flip flop between each frame buffer */
                     frameA = frame1;
                     frameB = frame2;
         }
         else {
                     frameA = frame2;
                     frameB = frame1;
         }
         synthesize(frameA, frameB, colsize); /* synthesizes from frameA to frameB (aka frame1 to frame2 or frame2
to frame1) */
         addtowav(fo, buffwrite);
         i++;  /*not necessary but nice to see how many columns of image were processed*/
//       printf("%d\n", i);
  }
  /***** END MAIN LOOP *****/


  printf("done. # frames processed: %d\n", i);

  finishwav(fo);
  sprintf(tmp, "Exiting program...");
  pc_printf(tmp);
  /* A command of 0x04 means to exit the program */
  request_transfer(NULL, 0, 0x04);

  return(1);

  exit_err:
  return(ERROR);
}


/* write wave file header */
void startwavwrite(FILE *fo) {
  int data[2];
       fwrite("RIFF", 1, 4, fo);
       data[0]=0;     //filesize, fill in later (totalbytes-8)
       fwrite(data, 4, 1, fo);
       fwrite("WAVEfmt ", 1, 8, fo);
       data[0]=16; //16 for PCM
       fwrite(data, 4, 1, fo);
       data[0]=1;     //1 for PCM
       fwrite(data, 2, 1, fo);
       data[0]=2;     //# of channels
       fwrite(data, 2, 1, fo);
       data[0]=44100; //samplerate
       fwrite(data, 4, 1, fo);
       data[0]=44100*2*2;    //bytespersecond (samplerate*channels*bytespersample(bitspersample/8)
       fwrite(data, 4, 1, fo);
       data[0]=0;     //bytes per capture??
       fwrite(data, 2, 1, fo);
       data[0]=16;    //bits per sample
       fwrite(data, 2, 1, fo);
       fwrite("data", 1, 4, fo);
       data[0]=0;     //bytes in data
       fwrite(data, 4, 1, fo);
       totalbytes = 44;
       fseek(fo, 44, SEEK_SET); //totalbytes
//     fclose(fo); //for debugging if wav header was written correctly at least
}


/* add 2 shorts to interleaved waveform data - to wav file */
void addtowav(FILE *fo, int* buffwrite) {
       int i; //, data[2];
       short data[2];
       for(i=0; i<BUFFER_LEN; i++)
```

```
	{
		data[1] = (short)buffwrite[i];		//will write R
		data[0] = data[1];			//mono for now...
		fwrite(data, 2, 2, fo);
	}

	totalbytes += BUFFER_LEN*2;
	databytes += BUFFER_LEN*2;
}


/* close up wav file, writing rest of header first */
void finishwav(FILE *fo) {
	int data[1];
	data[0]=totalbytes;
	fseek(fo, 4, SEEK_SET); //totalbytes
	fwrite(data, 4, 1, fo);

	data[0]=databytes;
	fseek(fo, 40, SEEK_SET); //databytes
	fwrite(data, 4, 1, fo);

	fclose(fo);
}


 /* synthesizes from frameA to frameB (aka frame1 to frame2 or frame2 to frame1) */
 /* interpolating the values and placing them in the buffers and swapping the buffers */
void synthesize(unsigned char* frameA, unsigned char* frameB, int colsize) {
	int i, j;// tempbuffadd;
	float tempmag, tempmagnext, freq, tempscaler; //, maxforscale
	while(!goahead);
	goahead=0;
	if(buffer == buffer2) {
		buffwrite = buffer1;
	}
	else {
		buffwrite = buffer2; /*choose which buffer to write in (the one that's not being read from) */
	}

	for(i=0; i<BUFFER_LEN; i++)   //empty buffer for adding after down there
		buffwrite[i] = 0;

	for(i=0; i<colsize/3; i++)    //red comp only for now (/3) (actually blue was first so now we are looking
at red with the 1/3 to 2/3
	{
		tempmag = ((float)frameA[i])/256;
		tempmagnext = ((float)frameB[i])/256;
		freq = i*((SAMPLE_RATE/2)/(colsize/3))/2; // possibly /2 also for some reason
		for(j=0; j<BUFFER_LEN; j++) { //building out interpolation table
			tempscaler = (tempmag + j*(tempmagnext-tempmag)/BUFFER_LEN) *
sinf(2.0*PI*counters[i]/SAMPLE_RATE*freq); //later +temppha goes inside sine
			buffwrite[j] += (int)(tempscaler*32768/8);
				//the value 8 above is volume. so 8 is useful for scaling 8 full-amplitude
				//sine-waves to 100% amplitude now. if your image is full - not much black,
				//this value should be higher. ultimately you should be able to set it when
				//running the program, which would be easy. we cannot have it auto-scale, however
				//because it is happening in realtime and would have to scan the entire image to
				//find the maximum sum of simultaneous sine wave amplitudes (aka max sum of values
				// of pixels in y-dimension
			counters[i]++;

		}
		if( tempmagnext == 0 ) counters[i] = 0; //reset counter for this freq if sinusoid is starting to
take a break
	}
}


/* 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);
}

/* Use the request_transfer framework to send data to be output to
       the screen.  The PC uses command 0x03 to indicate data to be
       output */
int pc_printf(char *message) {
  int len;

  len = strlen(message);
  while(len%4) len++;  /* Must be multiple of 4 */

  request_transfer(message, len, 0x03);
  wait_transfer();
  return(0);
}



/*************************************************************
 * Name: xmitISR
 *  Inputs: none
 *  Output: none
 * Purpose: Interrupt vector to be called whenever the serial
 *  port is ready for a sample to be written.  A value is written
 *  from the buffer and the index is incremented.
 *************************************************************/
interrupt void xmitISR(void) {
  /* MCBSP0_DXR is the hardware register for outgoing samples */
  MCBSP0_DXR=buffer[xindex++];
  //printf ("%d\n", xindex);
  if (xindex>=BUFFER_LEN) {
       xindex=0;
       goahead=1;
       if(buffer == buffer1) buffer = buffer2;
       else buffer = buffer1;
  }
}




/*************************************************************
 *    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) {
    return(ERROR);
  }
  return(OK);
}
```