The Beat Tower

Allison, Autumn, Zack

12/5/14

Summary

The beat tower is a single frequency audio spectrum volume meter.  It can isolate around a certain frequency and display it on a creative 8 segment LED bar graph. It is built around the **Arduino** Open Source Environment.

Table of Contents

1. Introduction
2. Uses
3. Code
4. Pictures
5. Conclusion
6. References
7. Appendices
8. Introduction

Our group juggled with the idea on what project to choose because we weren’t sure what we really wanted. Autumn saw an idea for the beat tower online and Zack liked the idea because music is awesome. Once we started, we broke up the work with Autumn taking over building the tower, Allison worked on the code, and Zack tackled the circuitry. We all pitched in with the speech assignment. When one needed help the others pitched in and over all we were a well-oiled machine.

1. Uses

The main use our product has is to be a party center piece. It lights up to the beat of the music translating the high pitch to the top lights and the bass on the lower lights. It is basically a very low-tech light bar. It will turn any boring party into a very fun one. If you are not planning on having parties any time soon, no problems. Whether you listen to music to pump you up, sooth you, or just for the entertainment, the beat tower enhances all of these experiences.

1. Code

#include <avr/pgmspace.h>

//#include "fix\_fft.h"

#include <WProgram.h>

/\* fix\_fft.c - Fixed-point in-place Fast Fourier Transform \*/

/\*

 All data are fixed-point short integers, in which -32768

 to +32768 represent -1.0 to +1.0 respectively. Integer

 arithmetic is used for speed, instead of the more natural

 floating-point.

 For the forward FFT (time -> freq), fixed scaling is

 performed to prevent arithmetic overflow, and to map a 0dB

 sine/cosine wave (i.e. amplitude = 32767) to two -6dB freq

 coefficients. The return value is always 0.

 For the inverse FFT (freq -> time), fixed scaling cannot be

 done, as two 0dB coefficients would sum to a peak amplitude

 of 64K, overflowing the 32k range of the fixed-point integers.

 Thus, the fix\_fft() routine performs variable scaling, and

 returns a value which is the number of bits LEFT by which

 the output must be shifted to get the actual amplitude

 (i.e. if fix\_fft() returns 3, each value of fr[] and fi[]

 must be multiplied by 8 (2\*\*3) for proper scaling.

 Clearly, this cannot be done within fixed-point short

 integers. In practice, if the result is to be used as a

 filter, the scale\_shift can usually be ignored, as the

 result will be approximately correctly normalized as is.

 \*/

#define N\_WAVE 256 /\* full length of Sinewave[] \*/

#define LOG2\_N\_WAVE 8 /\* log2(N\_WAVE) \*/

**int** led[] = {5,6,7,8,9,10,11,12};

**int** x = 0;

**char** im[128], data[128];

**char** data\_avgs[14];

**int** i=0,val;

#define AUDIOPIN 3

**void** setup()

{

 **for** (**int** i = 0; i <8; i++)

 {

 pinMode(led[i], OUTPUT);

 }

 Serial.begin(9600);

}

**void** loop()

{

 **for** (i=0; i < 128; i++){

 val = analogRead(AUDIOPIN);

 data[i] = val;

 im[i] = 0;

 }

 fix\_fft(data,im,7,0);

 **for** (i=0; i< 64;i++){

 data[i] = sqrt(data[i] \* data[i] + im[i] \* im[i]); // this gets the absolute value of the values in the

 //array, so we're only dealing with positive numbers

 };

 // average bars together

 **for** (i=0; i<14; i++) {

 data\_avgs[i] = data[i\*4] + data[i\*4 + 1] + data[i\*4 + 2] + data[i\*4 + 3]; // average together

 data\_avgs[i] = map(data\_avgs[i], 0, 30, 0, 9); // remap values for LoL

 }

 **int** value = data\_avgs[0];//0 for bass

 ledArray(value);

}

**void** ledArray(**int** input)

{

 //

 **if** (input > 8)

 {

 **for** (**int** i = 0; i <8; i++)

 {

 digitalWrite(led[i], HIGH);

 }

 }

 **else** **if** (input > 7)

 {

 **for** (**int** i = 0; i <7; i++)

 {

 digitalWrite(led[i], HIGH);

 }

 **for** (**int** i = 7; i <8; i++)

 {

 digitalWrite(led[i], LOW);

 }

 }

 **else** **if** (input > 6)

 {

 **for** (**int** i = 0; i <6; i++)

 {

 digitalWrite(led[i], HIGH);

 }

 **for** (**int** i = 6; i <8; i++)

 {

 digitalWrite(led[i], LOW);

 }

 }

 **else** **if** (input > 5)

 {

 **for** (**int** i = 0; i <5; i++)

 {

 digitalWrite(led[i], HIGH);

 }

 **for** (**int** i = 5; i <8; i++)

 {

 digitalWrite(led[i], LOW);

 }

 }

 **else** **if** (input > 4)

 {

 **for** (**int** i = 0; i <4; i++)

 {

 digitalWrite(led[i], HIGH);

 }

 **for** (**int** i = 4; i <8; i++)

 {

 digitalWrite(led[i], LOW);

 }

 }

 **else** **if** (input > 3)

 {

 **for** (**int** i = 0; i <3; i++)

 {

 digitalWrite(led[i], HIGH);

 }

 **for** (**int** i = 3; i <8; i++)

 {

 digitalWrite(led[i], LOW);

 }

 }

 **else** **if** (input > 2)

 {

 **for** (**int** i = 0; i <2; i++)

 {

 digitalWrite(led[i], HIGH);

 }

 **for** (**int** i = 2; i <8; i++)

 {

 digitalWrite(led[i], LOW);

 }

 }

 **else** **if** (input > 1)

 {

 **for** (**int** i = 0; i <1; i++)

 {

 digitalWrite(led[i], HIGH);

 }

 **for** (**int** i = 1; i <8; i++)

 {

 digitalWrite(led[i], LOW);

 }

 }

 **else**

 {

 **for** (**int** i = 0; i <8; i++)

 {

 digitalWrite(led[i], LOW);

 }

 }

}

#define N\_WAVE 256 /\* full length of Sinewave[] \*/

#define LOG2\_N\_WAVE 8 /\* log2(N\_WAVE) \*/

/\*

 Since we only use 3/4 of N\_WAVE, we define only

 this many samples, in order to conserve data space.

 \*/

**const** prog\_int8\_t Sinewave[N\_WAVE-N\_WAVE/4] PROGMEM = {

 0, 3, 6, 9, 12, 15, 18, 21,

 24, 28, 31, 34, 37, 40, 43, 46,

 48, 51, 54, 57, 60, 63, 65, 68,

 71, 73, 76, 78, 81, 83, 85, 88,

 90, 92, 94, 96, 98, 100, 102, 104,

 106, 108, 109, 111, 112, 114, 115, 117,

 118, 119, 120, 121, 122, 123, 124, 124,

 125, 126, 126, 127, 127, 127, 127, 127,

 127, 127, 127, 127, 127, 127, 126, 126,

 125, 124, 124, 123, 122, 121, 120, 119,

 118, 117, 115, 114, 112, 111, 109, 108,

 106, 104, 102, 100, 98, 96, 94, 92,

 90, 88, 85, 83, 81, 78, 76, 73,

 71, 68, 65, 63, 60, 57, 54, 51,

 48, 46, 43, 40, 37, 34, 31, 28,

 24, 21, 18, 15, 12, 9, 6, 3,

 0, -3, -6, -9, -12, -15, -18, -21,

 -24, -28, -31, -34, -37, -40, -43, -46,

 -48, -51, -54, -57, -60, -63, -65, -68,

 -71, -73, -76, -78, -81, -83, -85, -88,

 -90, -92, -94, -96, -98, -100, -102, -104,

 -106, -108, -109, -111, -112, -114, -115, -117,

 -118, -119, -120, -121, -122, -123, -124, -124,

 -125, -126, -126, -127, -127, -127, -127, -127,

 /\*-127, -127, -127, -127, -127, -127, -126, -126,

-125, -124, -124, -123, -122, -121, -120, -119,

-118, -117, -115, -114, -112, -111, -109, -108,

-106, -104, -102, -100, -98, -96, -94, -92,

-90, -88, -85, -83, -81, -78, -76, -73,

-71, -68, -65, -63, -60, -57, -54, -51,

-48, -46, -43, -40, -37, -34, -31, -28,

-24, -21, -18, -15, -12, -9, -6, -3, \*/

};

/\*

 FIX\_MPY() - fixed-point multiplication & scaling.

 Substitute inline assembly for hardware-specific

 optimization suited to a particluar DSP processor.

 Scaling ensures that result remains 16-bit.

 \*/

inline **char** FIX\_MPY(**char** a, **char** b)

{

 //Serial.println(a);

 //Serial.println(b);

 /\* shift right one less bit (i.e. 15-1) \*/

 **int** c = ((**int**)a \* (**int**)b) >> 6;

 /\* last bit shifted out = rounding-bit \*/

 b = c & 0x01;

 /\* last shift + rounding bit \*/

 a = (c >> 1) + b;

 /\*

 Serial.println(Sinewave[3]);

 Serial.println(c);

 Serial.println(a);

 while(1);\*/

 **return** a;

}

/\*

 fix\_fft() - perform forward/inverse fast Fourier transform.

 fr[n],fi[n] are real and imaginary arrays, both INPUT AND

 RESULT (in-place FFT), with 0 <= n < 2\*\*m; set inverse to

 0 for forward transform (FFT), or 1 for iFFT.

 \*/

**int** fix\_fft(**char** fr[], **char** fi[], **int** m, **int** inverse)

{

 **int** mr, nn, i, j, l, k, istep, n, scale, shift;

 **char** qr, qi, tr, ti, wr, wi;

 n = 1 << m;

 /\* max FFT size = N\_WAVE \*/

 **if** (n > N\_WAVE)

 **return** -1;

 mr = 0;

 nn = n - 1;

 scale = 0;

 /\* decimation in time - re-order data \*/

 **for** (m=1; m<=nn; ++m) {

 l = n;

 **do** {

 l >>= 1;

 } **while** (mr+l > nn);

 mr = (mr & (l-1)) + l;

 **if** (mr <= m)

 **continue**;

 tr = fr[m];

 fr[m] = fr[mr];

 fr[mr] = tr;

 ti = fi[m];

 fi[m] = fi[mr];

 fi[mr] = ti;

 }

 l = 1;

 k = LOG2\_N\_WAVE-1;

 **while** (l < n) {

 **if** (inverse) {

 /\* variable scaling, depending upon data \*/

 shift = 0;

 **for** (i=0; i<n; ++i) {

 j = fr[i];

 **if** (j < 0)

 j = -j;

 m = fi[i];

 **if** (m < 0)

 m = -m;

 **if** (j > 16383 || m > 16383) {

 shift = 1;

 **break**;

 }

 }

 **if** (shift)

 ++scale;

 } **else** {

 /\*

 fixed scaling, for proper normalization --

 there will be log2(n) passes, so this results

 in an overall factor of 1/n, distributed to

 maximize arithmetic accuracy.

 \*/

 shift = 1;

 }

 /\*

 it may not be obvious, but the shift will be

 performed on each data point exactly once,

 during this pass.

 \*/

 istep = l << 1;

 **for** (m=0; m<l; ++m) {

 j = m << k;

 /\* 0 <= j < N\_WAVE/2 \*/

 wr = pgm\_read\_word\_near(Sinewave + j+N\_WAVE/4);

 /\*Serial.println("asdfasdf");

Serial.println(wr);

Serial.println(j+N\_WAVE/4);

Serial.println(Sinewave[256]);

Serial.println("");\*/

 wi = -pgm\_read\_word\_near(Sinewave + j);

 **if** (inverse)

 wi = -wi;

 **if** (shift) {

 wr >>= 1;

 wi >>= 1;

 }

 **for** (i=m; i<n; i+=istep) {

 j = i + l;

 tr = FIX\_MPY(wr,fr[j]) - FIX\_MPY(wi,fi[j]);

 ti = FIX\_MPY(wr,fi[j]) + FIX\_MPY(wi,fr[j]);

 qr = fr[i];

 qi = fi[i];

 **if** (shift) {

 qr >>= 1;

 qi >>= 1;

 }

 fr[j] = qr - tr;

 fi[j] = qi - ti;

 fr[i] = qr + tr;

 fi[i] = qi + ti;

 }

 }

 --k;

 l = istep;

 }

 **return** scale;

}

/\*

 fix\_fftr() - forward/inverse FFT on array of real numbers.

 Real FFT/iFFT using half-size complex FFT by distributing

 even/odd samples into real/imaginary arrays respectively.

 In order to save data space (i.e. to avoid two arrays, one

 for real, one for imaginary samples), we proceed in the

 following two steps: a) samples are rearranged in the real

 array so that all even samples are in places 0-(N/2-1) and

 all imaginary samples in places (N/2)-(N-1), and b) fix\_fft

 is called with fr and fi pointing to index 0 and index N/2

 respectively in the original array. The above guarantees

 that fix\_fft "sees" consecutive real samples as alternating

 real and imaginary samples in the complex array.

 \*/

**int** fix\_fftr(**char** f[], **int** m, **int** inverse)

{

 **int** i, N = 1<<(m-1), scale = 0;

 **char** tt, \*fr=f, \*fi=&f[N];

 **if** (inverse)

 scale = fix\_fft(fi, fr, m-1, inverse);

 **for** (i=1; i<N; i+=2) {

 tt = f[N+i-1];

 f[N+i-1] = f[i];

 f[i] = tt;

 }

 **if** (! inverse)

 scale = fix\_fft(fi, fr, m-1, inverse);

 **return** scale;

}

1. Pictures





1. Conclusion

By far the biggest problem we had was the code. As Ms. Kanemoto can attest to, the code was not bad at first but soon became a monster. The problem was it used a function that our Arduino program did not recognize. Besides that the only other hump we had to get over was the connection with the wires. Our decision was to use electrical tape when in hind sight we probably should have soldered the wires together. In all, things went pretty smoothly and our project turned out to work very well.

1. References
* <http://www.instructables.com/file/FZWBXPZH5F2ZERD>
* [http://forum.arduino.cc/index.php/topic,38153.0.html](http://forum.arduino.cc/index.php/topic%2C38153.0.html)
* Ms. Kanemoto’s office hours
1. Appendices

1. Title Page ………………………………… 1
2. Introduction ……………………………. 2
3. Uses ………………………………………… 3
4. Code ………………………………………… 4-11
5. Pictures…………………………………….. 12
6. Conclusion………………………………… 13
7. References………………………………… 13
8. Appendices ………………………………. 14