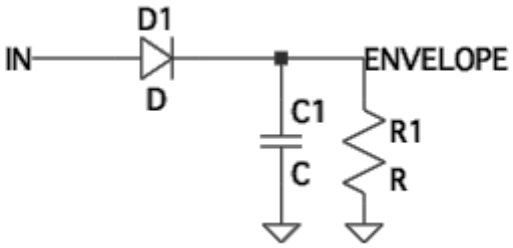
MARCH 2, 2015

Beat Detection On The Arduino

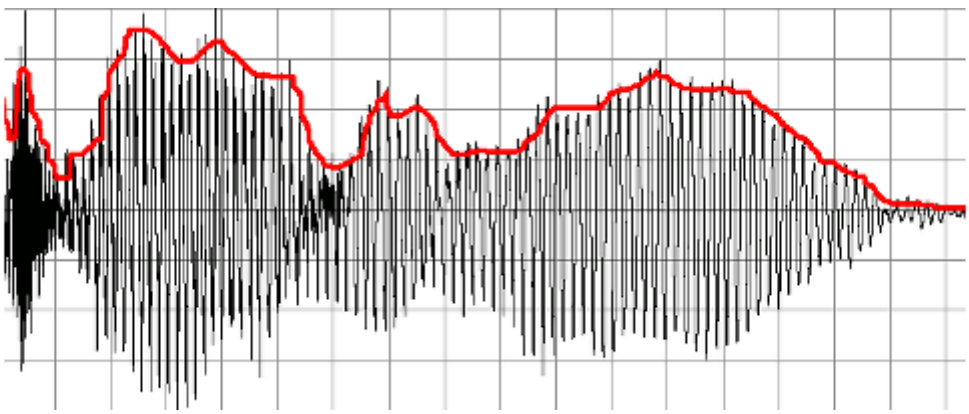
So it's an undeniable fact that the Arduino is a nifty little gadget but can it do Digital Signal Processing? For a recent music event, I set out to design a circuit that would flash and synchronise a set of LEDs to a music beat.

The choice of the Arduino was mainly out of simplicity, I had a matter of about a day to come up with my solution and I didn't feel like messing around with breadboards, it's a lot faster to do a recompile than it is to add another amplifier stage to a discrete circuit.

So how do you detect the beat of a music signal? I did a bunch of research and the common consensus came down to one of three major approaches. The simplest was an envelope detector, simply take the absolute magnitude of the audio signal and pass it through a low pass filter and then threshold against some fixed value. This can be achieved in analog form simply as a diode, capacitor, resistor and a comparator.



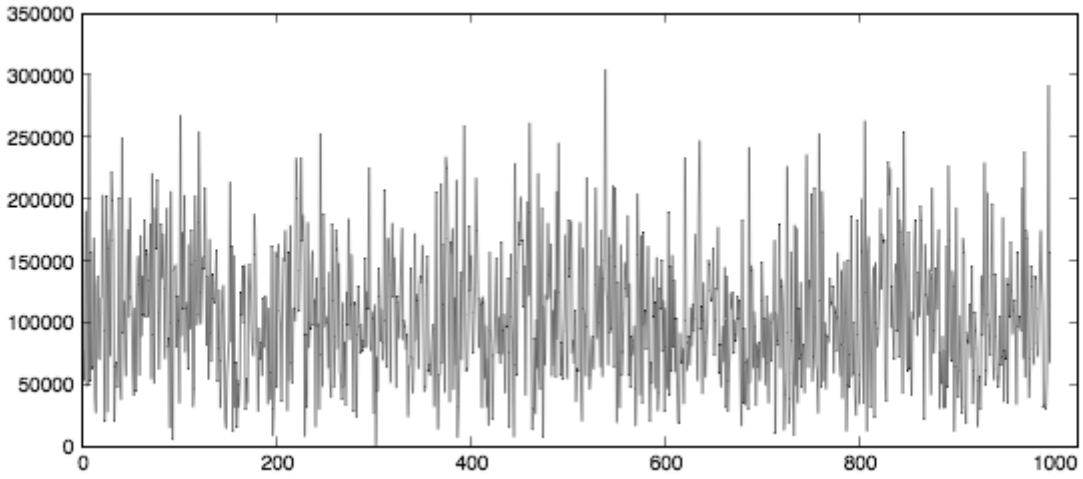
Envelope Detector Using Analog Electronics.



Envelope Detection Of a Signal.

The problem with envelope detection is that it frankly sucks, with a clean audio signal that hasn't undergone dynamic range compression it could be practical. However in my application there will be loud screaming (interference), and being dance music the dynamic range will be very compressed. These confounding factors make it very difficult to get a clean detection.

The second approach used Fourier analysis, to analyse the magnitude of the different frequency components of the signal. If you are able to break down the signal into frequency components you can separate out the components that make up the low frequency sounds of the bass instruments.



Example Fourier Transform Of A Noisy Audio Signal.

By thresholding the magnitude of the bass component against time and the average of the frequency spectrum you can fairly accurately detect the bass line of the track. This would work well for my scenario however Fourier analysis is mathematically complex.

There exists optimised libraries for the Arduino which use an approach know as the Fast Hartley Transform which is generalised for magnitude calculation only. I decided given I was using an Arduino Uno I didn't want to attempt a full Fourier transform on my signal, ultimately I would be discarding a lot of signal information, and on the Arduino clock cycles are a definite resource limit.

There exists another algorithm known as the Discrete Fourier Transform which essentially implements a single bin of the Fourier transform. I did experiments using the optimised Goertzel algorithm on the Arduino, this was very successful taking only 20us to process each sample. However the bandwidth of the DFT bins was too narrow to pick up the entire bass frequency range 20-200hz and once you start adding multiple bins it became far less efficient.

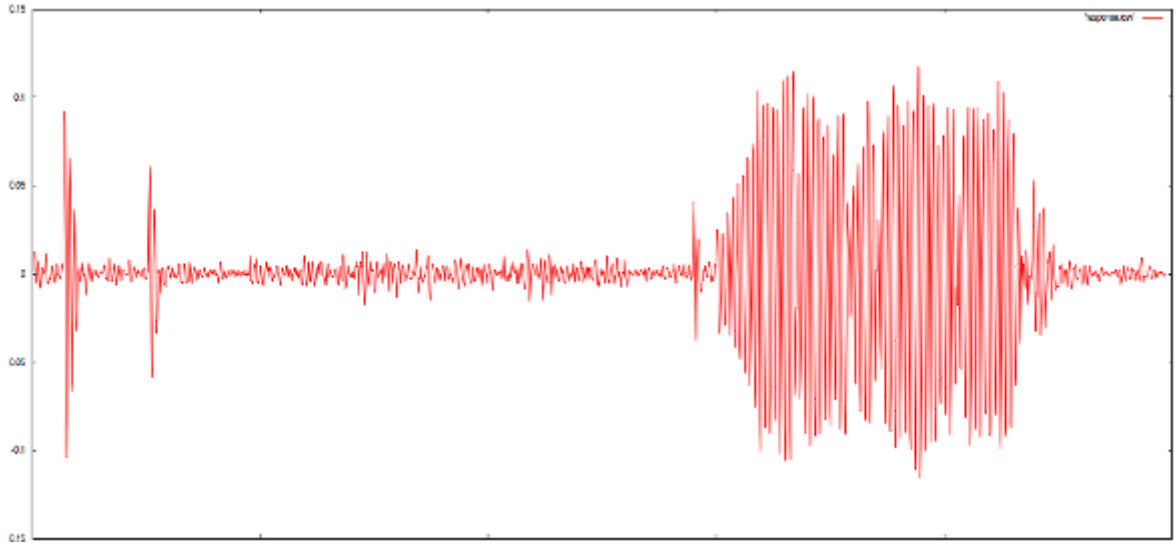
```
//Goertzel DFT By Damian Peckett 2015
#define FREQ 1000.f // Center Frequency
#define RATE 5000.f // Sample Rate
#define WINDOW 200 // Number Of Samples Per Window

void DFT(short *samples, int nSamples) {
    float coeff = 2.f*cos((2.f * 3.14159f * FREQ) / RATE);
    float Q0 = 0, Q1 = 0, Q2 = 0, power;

    for(int i = 0; i < nSamples; ++i) {
        Q0 = coeff * Q1 - Q2 + (float)(samples[i]-512)/1024.f;
        Q2 = Q1; Q1 = Q0;
        if(i && (i%WINDOW) == 0) {
            power = Q2*Q2 + Q1*Q1 - coeff*Q1*Q2;
            Serial.println(power);
            Q1 = 0; Q2 = 0;
        }
    }
}
```

Goertzel Algorithm For Arduino, 20us Per Sample.

The third approach, and ultimately the one I chose, was very similar to the first approach but instead the first step of the process was to pass the audio signal through a bandpass filter tuned to allow only low frequency bass components through.

I used a variant of this approach, firstly the signal was bandpass filtered to the range of 20-200hz and then I calculated the absolute magnitude of the signal and passed it through a low pass filter at 10hz. This detected envelope was then passed through a third bandpass filter tuned to 1.7-3.0hz.

This final filtering stage rejected constant low frequency components, leaving only the drum kicks, the range 1.7-3.0hz corresponds to 100-180bpm, common tempos for electronic music.



Darude Sandstorm Bass Drop.

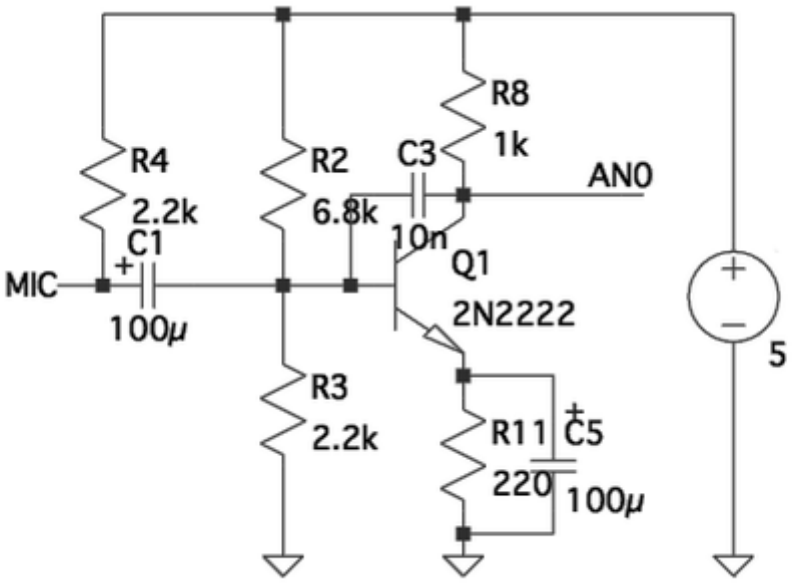
I prototyped the DSP algorithm on my pc. For testing I downloaded a recorded video of a music event and stripped the audio track and downsampled it to a 5khz sample rate. The choice of sample rate was driven by the frequency range I was interested in capturing, 5khz was high enough to capture the primary components of human speech but also low enough to give the Arduino a chance at processing it in realtime.

So for the first order of business, adding a microphone to the Arduino. The Arduino has a 10bit Analog-to-digital Converter capable of sampling at up to 200khz. The ADC was more than sufficient, the difficult part came down to interfacing with a microphone. With the default voltage reference the Arduino's ADC has a full scale input of 0-5 volts. The output of a common electret microphone can range from 5 - 50+ millivolts.

In order to maximise the dynamic range of the ADC we need to amplify our microphones output. Generally this is fairly trivial with a BJT amplifier or an Opamp. However we have an issue human speech usually occurs at an SPL (loudness) of 70dB or so, the SPL at an electronic music venue could be as high as 120dB. This is a dynamic range of over 50dB. With typical microphone sensitivities this would result in a microphone output of 0.5-100 millivolts RMS.

Our 10bit ADC only has an absolute dynamic range 60dB, however due to noise etc we won't be able to achieve the theoretical limit. Therefore it is not possible to build a fixed gain front-end amplifier that could handle the signal range.

One could implement an amplifier that had automatic gain control, however I was highly constrained for time so I designed a preamplifier based on the assumption that our music beat will be very loud as can be expected in a dance music venue. Due to our interest in the bass frequency range I also designed low pass filtering into the preamp.



Electret Preamplifier Circuit.

On the second ADC pin I added a 10k trimpot to provide an adjustable signal threshold. Ideally in a future revision I'd make the threshold levelling automatic based on tracking signal averages etc.

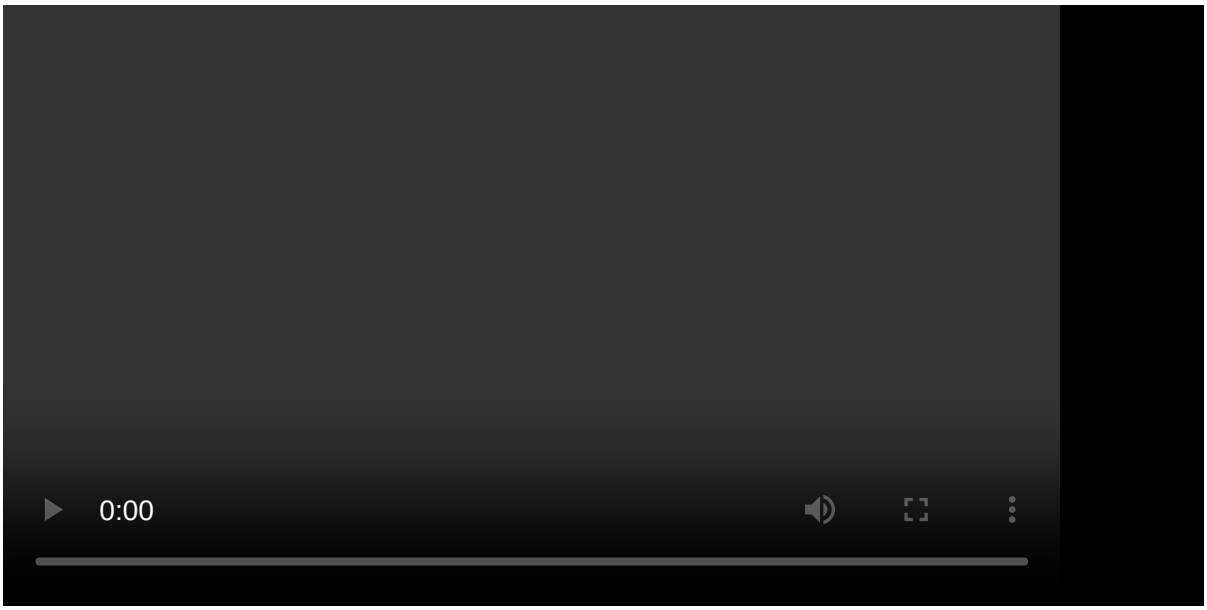
Anyway without further ado here is the source code to the beat detection routine, I haven't spent much time optimising it but it appears functional. Processing each sample takes 196us on an Arduino Uno @ 16Mhz.

```
// Arduino Beat Detector By Damian Peckett 2015
// License: Public Domain.

// Our Global Sample Rate, 5000hz
#define SAMPLEPERIODUS 200

// defines for setting and clearing register bits
#ifndef cbi
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))
#endif
#ifndef sbi
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#endif

void setup() {
    // Set ADC to 77khz, max for 10bit
```

```
    sbi(ADCSRA,ADPS2);
    cbi(ADCSRA,ADPS1);
    cbi(ADCSRA,ADPS0);

    //The pin with the LED
    pinMode(2, OUTPUT);
}

// 20 - 200hz Single Pole Bandpass IIR Filter
float bassFilter(float sample) {
    static float xv[3] = {0,0,0}, yv[3] = {0,0,0};
    xv[0] = xv[1]; xv[1] = xv[2];
    xv[2] = sample / 9.1f;
    yv[0] = yv[1]; yv[1] = yv[2];
    yv[2] = (xv[2] - xv[0])
        + (-0.7960060012f * yv[0]) + (1.7903124146f * yv[1]);
    return yv[2];
}

// 10hz Single Pole Lowpass IIR Filter
float envelopeFilter(float sample) { //10hz low pass
    static float xv[2] = {0,0}, yv[2] = {0,0};
    xv[0] = xv[1];
    xv[1] = sample / 160.f;
    yv[0] = yv[1];
    yv[1] = (xv[0] + xv[1]) + (0.9875119299f * yv[0]);
    return yv[1];
}

// 1.7 - 3.0hz Single Pole Bandpass IIR Filter
float beatFilter(float sample) {
    static float xv[3] = {0,0,0}, yv[3] = {0,0,0};
    xv[0] = xv[1]; xv[1] = xv[2];
    xv[2] = sample / 7.015f;
    yv[0] = yv[1]; yv[1] = yv[2];
    yv[2] = (xv[2] - xv[0])
        + (-0.7169861741f * yv[0]) + (1.4453653501f * yv[1]);
    return yv[2];
}

void loop() {
    unsigned long time = micros(); // Used to track rate
    float sample, value, envelope, beat, thresh;
    unsigned char i;

    for(i = 0;;++i){
        // Read ADC and center so +-512
        sample = (float)analogRead(0)-503.f;

        // Filter only bass component
        value = bassFilter(sample);

        // Take signal amplitude and filter
        if(value < 0)value=-value;
        envelope = envelopeFilter(value);

        // Every 200 samples (25hz) filter the envelope
        if(i == 200) {
```

```
                // Filter out repeating bass sounds 100 - 180bpm
                beat = beatFilter(envelope);

                // Threshold it based on potentiometer on AN1
                thresh = 0.02f * (float)analogRead(1);

                // If we are above threshold, light up LED
                if(beat > thresh) digitalWrite(2, HIGH);
                else digitalWrite(2, LOW);

                //Reset sample counter
                i = 0;
        }

        // Consume excess clock cycles, to keep at 5000 hz
        for(unsigned long up = time+SAMPLEPERIODUS; time > 20 && time <
up; time = micros());
    }
}
```

Arduino Beat Detector, Tested On Arduino Uno.



Testing The Detection Algorithm, Basic Test Using Subwoofer.

However despite the circuit working I decided against using it at the party, during testing I wired up the blue led strip I was going to use in my outfit and allowed it to sync to the music. The blue LEDs were incredibly bright and with the music synchronised strobing the effect was very intense.

I think it gave me some kind of mini-seizure, after switching it off I was left with a headache and I started seeing auras. I've never had any issues with migraines or epilepsy, I concluded that the device was just too dangerous to wear in public. In the end I took a bunch of the LEDs and made them into a belt.