**Arduino-Based Heart Rate Monitor**

The Arduino-based heart rate monitor was developed by Jason Kahn, Marc Bucchieri, Chris Smith and Katrina Miaoulis as an affordable, reliable alternative to commercial or computer-based heart rate monitor systems. The circuit runs entirely from an Arduino powered by a battery pack or external power source, and can deliver heart rate values via Serial port or XBEE.

**Usage**

Pending a more compact form factor for the heart rate monitor, the essential components consist of an Arduino Uno, a circuit embedded on a PCB or breadboard, and a fabric cuff with a wire tether. The PCB should be plugged directly into the pins on the Arduino (in the same style as most Arduino shields). The user slides the cuff up his/ her forearm until it is tight but comfortable, and the cuff tether connects to the PCB by way of a four-pin plug. Once the components have been set up (and the Arduino is powered from an external source) the heart rate monitor is ready for use.

Currently the most convenient interface for reading heart rate values is via the Arduino’s Serial communication interface. The values are sent in the form of 2 digit numbers every second, and can easily be received by a computer’s Serial port or through a radio relay such as XBEE. Since the heart rate monitor was developed for use with Shaky Table and other RAGE-Control games, future systems involving this device will most likely be set up with XBEE to communicate between components in the game system.

**Circuit Design**

The concept of using a circuit such as this to measure heart rate has already been successfully employed and documented as part of various open source projects. Our circuit employs a green light photo emitter/ photodiode (P/N: MTRS5250D). The sensor shines a green LED into the user’s skin and reads the intensity of the reflected light, which varies according to the amount of oxygenated blood in the capillaries near the surface of the skin. The photodiode returns this reflected light as a current. In order to handle this signal in the various filters and amplifiers we first convert the current to a voltage by means of a basic current-voltage converter on the operational amplifier (see circuit diagram).

The resulting voltage then passes through a series of active high- and low-pass filters on the operational amplifier chip. These are, in order of application,

1. A passive high pass filter with cutoff frequency of 0.7 Hz (or 43 bpm)
2. An active low pass filter with cutoff frequency of 3.4 Hz (204 bpm) and a gain of 45
3. A second active low pass filter with cutoff frequency of 3.4 Hz and a gain of 100
4. A non-inverting voltage follower to lower the output impedance

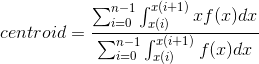
Altogether the op amp circuits act to amplify frequencies in the range of 40-200 beats per minute while filtering out all others. The output signal from the voltage follower is then sampled by the Arduino in conjunction with a pull down resistor.

**Signal Processing**

The truly original part of this design is the signal processing that takes place on the Arduino. The Arduino Uno platform has very limited programming capabilities, stemming from its low-level interface and the small amount of memory available. Arduino sketches are stored in the 32 Kb of flash memory while all of the non-constant program variables are stored in its 2 Kb of SRAM. This means that for the purpose of a project such as this only 2Kb, or about 3 arrays of 256 integers, are available for data processing in the sketch. For this reason we had to find a signal processing strategy that was not too memory- or computationally expensive and could take place at least partially in place.

The Arduino sketch uses a discrete wavelet transform to separate the signal into a set of frequency “bins”, and then computes a core frequency value based on the relative strengths of those bins. In theory the wavelet transform is similar to a Fast Fourier Transform, but localizes its results in time as well as frequency. For our purposes we basically ignored the time localization information for each window and instead sufficed to pull out a core frequency from each 4-second snapshot of the signal. (In case this wasn’t clear from above, the Arduino reads the signal in real time and calculates heart rate for a four second window every second. This also fits well computationally, as the sampling rate is 64 Hz and the input array holds 256 values). The sketch uses the Haar wavelet transform to perform a 7 level decomposition of the input signal. The Haar wavelet transform is one of the simplest discrete wavelet transforms, and acts by recursively taking the sum and difference between adjacent values from the signal. The differences are stored as one “level” and the procedure repeats on the sums until one final sum remains. Since our input array has 256 = 28 values, the wavelet transform results in 7 levels of differences (the smallest containing two values) and a final array of two sums, which we ignore.

After the discrete wavelet transform has taken place the procedure still has to extract a meaningful value from the results. First each level (or bin) of the wavelet transform is assigned a “power”, derived from the absolute mean value of the bin, and an average frequency. At this point the data could be represented as a line plot of power vs. frequency, or equivalently as a series of trapezoids with two corners on the x-axis and two corners formed by adjacent points on the plot. To find the core frequency we then calculate the geometric center of the shape by summing the area of each trapezoid times the x (frequency) position of its centroid times its width, and dividing by the sum of the areas times their respective widths. The relevant equations are given below:



This approach may at first seem to be needlessly complicated. However, it yields much more reliable results than taking a weighted average of the frequency spectrum since it automatically accounts for the varying widths of frequency bins. This method is made even more accurate by automatically setting to zero the powers of bins with completely unreasonable frequency values. The constraints for this are set in the sketch as constants. By default 0 Hz is taken as the low cutoff value and 8 Hz as the high cutoff value.

**Code Structure**

This procedure takes place entirely in a single Arduino sketch and accompanying helper class, and requires no external libraries (other than Arduino.h). The helper class WaveletArray handles a circular array for storing input values. This circular array is built specifically to be of the same size as the wavelet transform array (default: 256 integers) and the class defines functions for copying one “window” (256 samples or 4 seconds) worth of signal at a time, and shifting the focus over 64 samples (1 second) to the next window.

The Arduino sketch handles sampling and all computations on the input signal. In order to maintain a precise sampling rate, we set the Timer1 interrupt to execute every 1/rate seconds (default rate is 64 Hz). Every time the interrupt routine executes, it appends a new value to the input array inside WaveletArray. Meanwhile, every time the loop() function checks continuously whether the length of the current window in the input array is equal to the target window length. As soon as this is true, the sketch momentarily turns off interrupts, copies the last window worth of input values into a separate transform array, and then resumes sampling. In this manner we can avoid writing over the input array while doing (relatively) time-intensive calculations.

At this point, the sketch has an array of 256 raw input values from the circuit to turn into a heart rate. This is accomplished in 3 subroutines. The first performs a multilevel discrete wavelet decomposition based on the Haar wavelet, as described above. The second iterates over the bins created by the transform and assigns each a frequency and a power, which go in separate arrays. Finally, the third uses the formula for finding the weighted center of mass of a body made up of trapezoids. This is analogous to the problem of finding the central frequency over a range where all of the bins have different widths, and larger bins should be weighted more strongly because they contain more frequencies.

**Circuit Diagram**

