Cathy Chen – cc464
Shane Pryor – sjp45

# A Pulse Oximeter on an ATmega644 Microcontroller

## *Introduction:*

In this project, we seek to monitor a patient's heart rate and blood-oxygen level using a pulse oximeter. The pulse oximeter is designed using infrared and visible (red) light detection from light that passes through a patient's finger from an emitter. The absorption will tell when blood is moving through the finger and how much of this is oxygen-rich. The output of this analog circuit will be fed into an Atmel ATmega644 microcontroller, which will compute the pulse and oxygen level from these numbers. The microcontroller will output an NTSC video signal to a black and white television so that the patient's signals can be monitored.

## *High Level Design / Background:*

Pulse oximeters have been used in medical settings for many years. In many cases, such as during an operation, in intensive care, the emergency room, even an unpressurized aircraft, a person's oxygen level may be unstable and needs monitoring. In addition, from these readings, the person's heart rate can also be determined. This project is an attempt to construct a working version of a pulse oximeter from a relatively cheap set of parts – including a microcontroller. An off-the-shelf microcontroller has enough processing power to perform the tasks required for this design; however, in any commercial application, specialized hardware will be designed that is specifically suited to the task.

The sampling portion of this design requires an infrared emitter (around 940 nm wavelength) and a red light emitter (around 660 nm wavelength). Absorption of oxyhemoglobin and the deoxygenated form differs significantly between these wavelengths. Therefore, using the ratio of the two absorption values gives the percentage of arterial hemoglobin for oxyhemoglobin.[3] The detectors do not give a very high voltage, so the output from the detector needs to be amplified using op amps before passing into the microcontroller for analysis. If not, the relative change will not be seen when the microcontroller makes the input a discreet value.

This attempt at a pulse oximeter is fairly crude and does not take into consideration some important facts if it were to be used in a serious situation. For instance, it does not take into account other gasses in the blood stream. If a person has been rescued from a burning building, they may have carbon monoxide poisoning. In order to distinguish the difference between CO and $O_2$, absorption at additional wavelengths must be performed. Another example is a person suffering from poor gas exchange in the lungs. Their blood may have a 100% oxygen level, but may still be suffering from too much carbon dioxide ($CO_2$) that cannot be exchanged and exhaled.[3]

The microcontroller is required to perform a discrete Fourier transform to determine the pulse. This transform will take a collection of data over time and extract the amplitude of each of the frequencies it contains. In the case of our data, there should be a pretty distinct pattern of when there is blood movement. Therefore, we should obtain one frequency where the amplitude is much higher than any other frequencies detected. This should correspond to the frequency of the pulse of the person using the device. In our application, only the forward transform is required as we want to go from data to frequencies, but not the inverse of this. We use a fast Fourier transform (FFT) to reduce calculation time.

Traditionally, what is seen on television is received from a public broadcast. However, this broadcast is in the National Television System Committee (NTSC) format[4], for which the specifications are public. Therefore, if the means are available, someone can generate their own NTSC signal for use with a television. In this project, this is what we do to output data. We generate a black and white NTSC signal that displays a history of samples to show how the heart beats, and then also compute the pulse and oxygenation of the blood for display on the screen.

In most design projects, there is a tradeoff to what should be done with hardware or with software. In our project, there is not much of a design comparison. The sampling and amplification must be done in hardware with analog values to obtain the correct results. For the calculations and the video generation, we need a device with enough processing power and features to perform meet all the timing requirements. In this case, the ATmega644 is a good fit at a low cost.

## *Hardware Design:*

The hardware for this project consisted of a light source and a photo detector. The light is shown through the tissue on the finger. As the blood passes through capillaries in the finger, the variation in blood volume causes a variation in the light detected by the phototransistor. The source and detector are mounted on either side of the finger to measure changes in transmitted light. In the project, we used an infrared LED and Red LED. The ratio of these two absorptions will give us a $Sp0_2$ reading. This ratio number corresponds to the oxidization level of the blood as per the website (5).

The signal from the photo detector needs to be amplified in order to be more accurately used by the Analog to Digital convertor on the SKD500. To do this, we used and LM358 package of dual operation amplifiers to band-pass and amplify the signal. The circuit used was as follows:

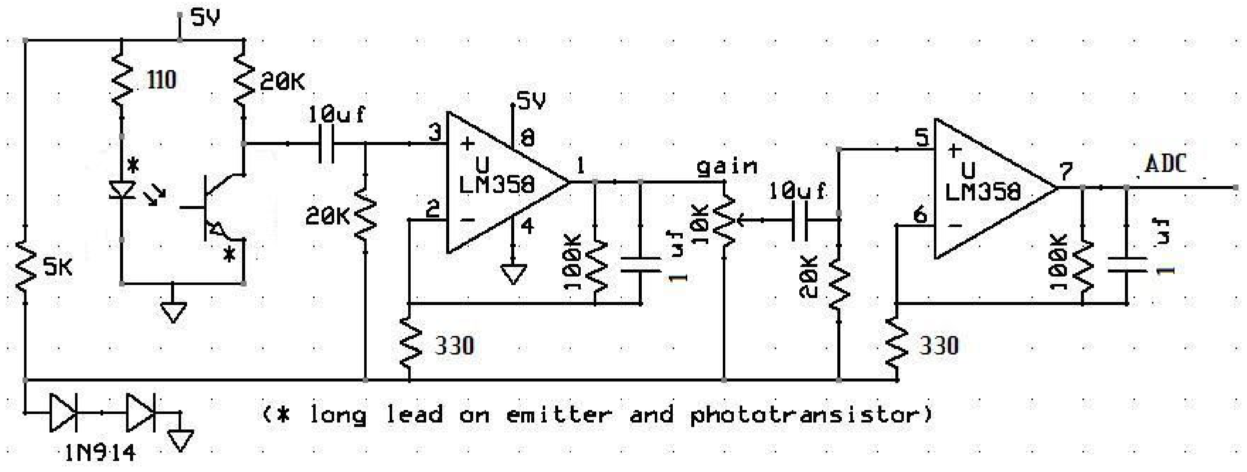


**Figure 1- Hardware Schematic**

On the low end, the signal is contained by the movement artifacts. (These are generated by the finger moving, causing the underlying tissue to distort). On the high end the signal is contained by mains-hum interference. The circuit consists of two identical band-pass filters, and each with a large gain. The original circuit consisted of a 1 K ohm resistor at ports 2 and 6, but this was changed to 330 to facilitate a larger gain. The potentiometer in between the first and second operational amplifier serves as another way to change the gain of the system. Using the potentiometer, clipping can be avoided from large signals. The 10 uf capacitors have the ability to stand some reverse bias. The signal, after it is amplified and filtered is tapped from pin 7 of the LM358 and sent to the ADC convertor. Two of these circuits needed to build using Infrared and Red spectrum LEDs and photo transistors. The Infrared LED used was a LTE-4208 160-1029-ND Emitter IR 5 MM 940 NM Clear. The phototransistor used was an LTR-4206E 160-1030-ND Phototran npn 3mm IR Dark. The Red phototransistor used was an All electronics PTR-1 REF 1140661 LN#4. We choose to build these on two separate boards since the phototransistor we had would have interference between the two circuits.

## *Program Design:*

The program design contains a few major parts. First, the samples must be read in from the circuit we built. This is done by using the built-in analog to digital converter (ADC) on the mega644. It performs a conversion from analog to a ten bit digital value based upon a reference voltage ($A_{ref}$). In our case, our circuit ran on a five volt $V_{cc}$, so our $A_{ref}$ was set to five volts. Since the program was running C, we had eight and 16 bit variable sizes available to us to store the data history. Since the least two significant bits of the ADC do not significantly affect our design, we ignored those and stored each value as eight bits. In addition, we would need twice as much memory to store off those values, which we could not afford. The topic of memory will be discussed later in this section. To read two values from the ADC when needed, one would be read, then the ADMUX register changed to the other input, and then the program waited until the second conversion was complete.

The past 64 samples are stored for both the infrared and red light circuits. After they are initially filled, on a new sample, each sample is moved down one position in the array and the new value inserted at the end. Samples are taken at a rate corresponding to the video output. A new sample is taken once every 14 frames. Since there are 60 frames per second, this corresponds to about four times per second. This was chosen because it would let us determine pulse in increments of about four or five beats per minute. This resolution was chosen based upon the number of samples we could store and the length of time we needed to sample over to perform the FFT.

Next, once the history of samples is collected, we perform an FFT on them to obtain our values. The FFT routine that we use performs the math in fixed point notation to decrease the number of cycles required to run it (over floating point). This routine, and also the fixed point multiply routine it uses, was provided by Professor Bruce Land.[1] The FFT takes an array of 16-bit fixed point numbers in 8:8 format for both the real and imaginary components of the input. Since our signal is a real signal, there is no imaginary component, so this part of the input is always set to zero. The output real and imaginary components are stored in these same two arrays in memory. Therefore, before computation, a copy of the samples needs to be made so they are not overwritten. Also, as stated above, the FFT takes 16-bit fixed point values, so the eight bit values needed to be shifted up to fill the top eight bits of the 16-bit inputs.

Once the data arrays are filled, the output needs to be determined. Each element of the arrays now contains a value corresponding to the amount the input signal contained that frequency. Since a real and imaginary portion are returned, we take these and square them, sum them, and then take the square root of the result for each array element. The square root routine was provided by Professor Land as well, as part of his fixed point arithmetic page.[1] The square root routine was modified by us to only perform the whole number portion of the procedure. This was done to eliminate some computation time, as it is now a small set of comparisons. Elements of the array start at frequency of zero and increase by (60/14) beats per minute for each element since this is our sampling rate. We first tested this FFT routine on sine wave inputs from a function generator. This took a while to be able to perform correctly as there were a few small bugs that were difficult to remove.

The calculations for oxygenation level were much simpler than the FFT required to extract the pulse. The oxygenation level can be represented as a ratio between the red light and infrared light samples. To get a good estimation of this ratio, we take an average of both sets of samples, then divide. On the screen we display one whole number place and two decimal places. For this ratio, a ratio of 0.5 corresponds to about a 100% blood oxygen saturation. A reading of 1.0 corresponds to about an 82% blood oxygen saturation. Anything near this percentage or below would be dangerous to the person being observed.[5] Generally during testing we found our value to hover in the 0.65 to 0.80 range, which corresponds to a value in the 90%-95% range, a very normal reading.

The final portion of the code is video generation to place the results on a black and white television set. The video is generated in NTSC video so that it will work on any television that supports this format (any made for use in the United States, Canada, Japan, and some other countries). The video generation is taken from a project completed in the fall semester of 2008.[2] The previous version of the code was changed to output a scrolling video of the current signal, and the past approximately four seconds of input data. The output is read from a buffer that stores the current output. When the buffer is updated, the screen will reflect the results on the next frame. The size of the buffer is based on the fact that the device has 4KB of on-chip RAM. After some testing, it was determined that with a 20MHz clock frequency, to be able to meet the NTSC timing specifications, about 30 bytes of data could be output per line. After memory was set aside for storing the raw input, this left us with 96 lines of output to the screen. This many vertical lines left the memory about 97% full so that there was a small amount of RAM left for run time.

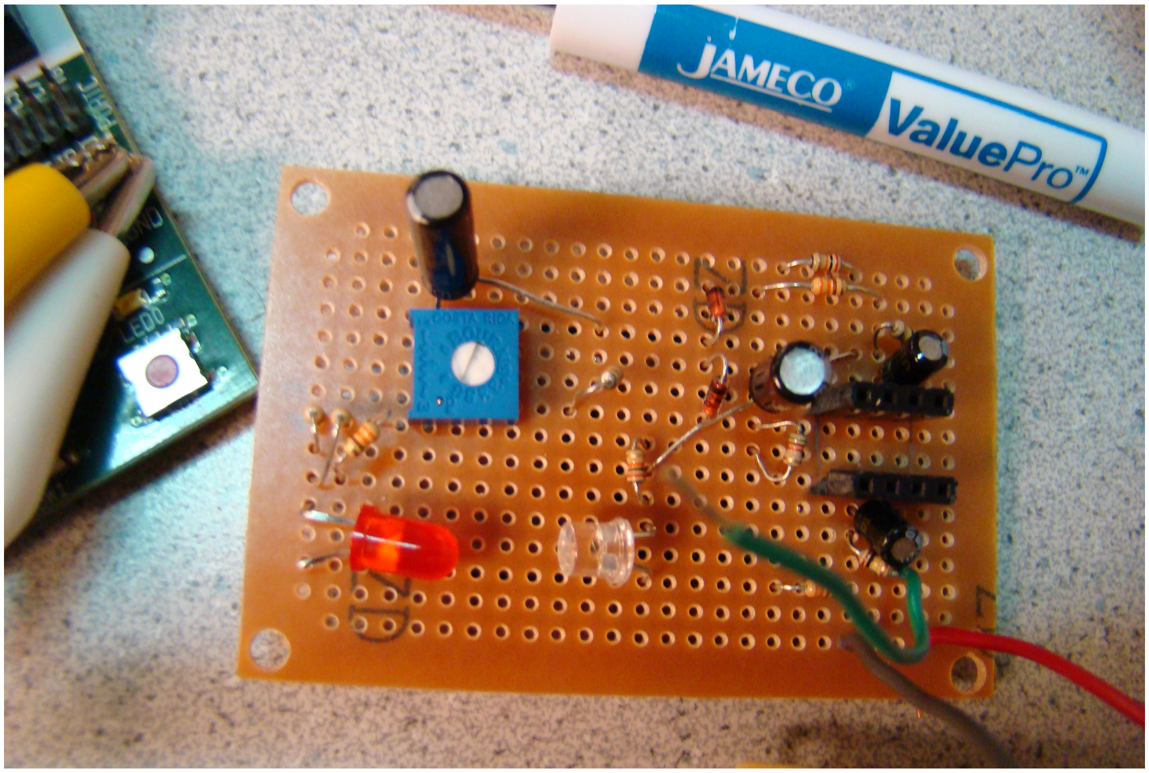
The output screen size was changed to be a wider and shorter output than the previous versions of the code. This widening allows for more past data to be stored on the screen. The scrolling effect of the screen was generated by having a pointer to the current byte on the screen to start output from, and the progressing once through the entire buffer per frame from whatever starting point was desired. New data would be written into the buffer starting at one bit position before whatever the current byte and bit were for the scrolling. In this way, the newest data would always appear at the right side of the screen as the last to be output. This produces a scrolling effect from right to left.

There is a small portion of the screen at the bottom that does not scroll that contains the extracted beats per minute and oxygen information. Therefore, the video generation was deterministically done based upon the current line number. This version of the video code had 96 lines, where the bottom eight lines did not scroll. The output is performed using the double-buffered USART, so as soon as one byte is done outputting, the next byte can start and another byte can be buffered from memory. This output is faster than reading the bytes from memory sequentially and outputting them through a port pin because the ports are not buffered like the USART.

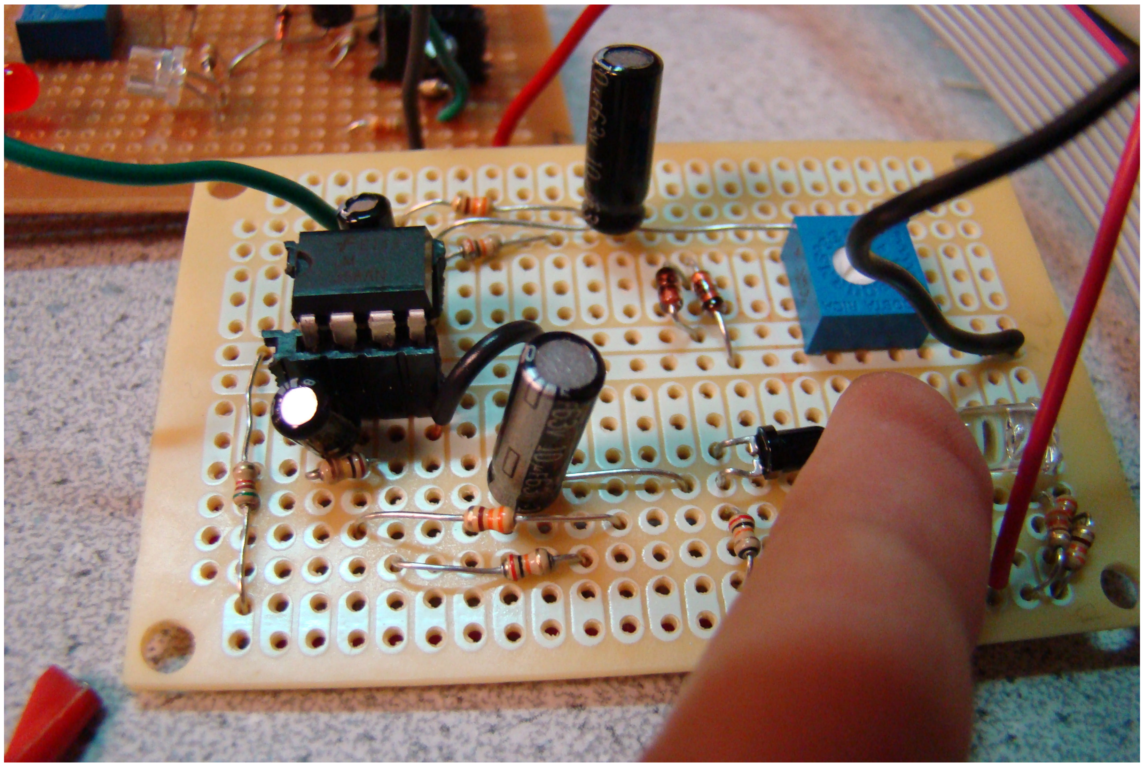
There is a library of characters that were kept from the original version of the code that contains the Arabic numerals, capital English letters, and some English punctuation. An entire ASCII character code set was removed from the code because this was not particularly useful for this project. Methods to draw a point, a line, and the small characters in the buffer are kept. In addition, the video driven interrupts are kept, but modified in the ways described above to produce the output we desire.

## *Results:*

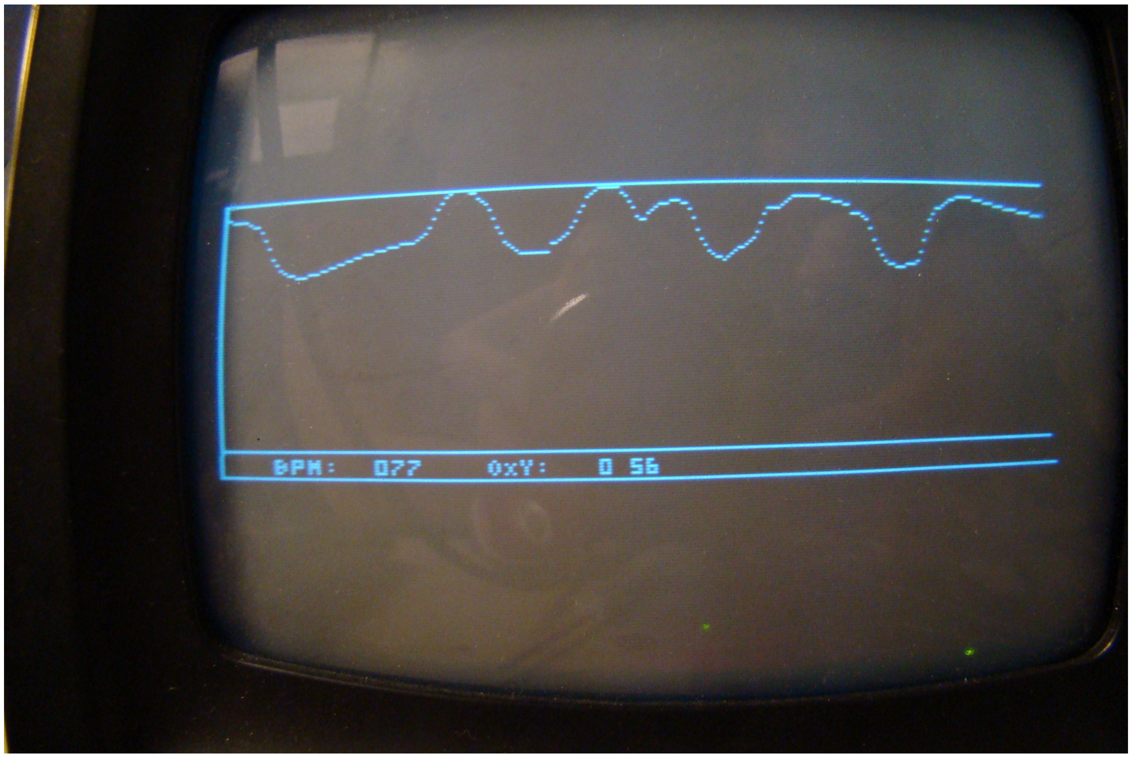
This project was fairly successful. After some experimentation, is appears to work the best with people with fairly thin skin, at the tip of the pointer finger. The following are pictures of our design and the output of the Beats Per Minute, Oxygen level, and graph of the heartbeat.



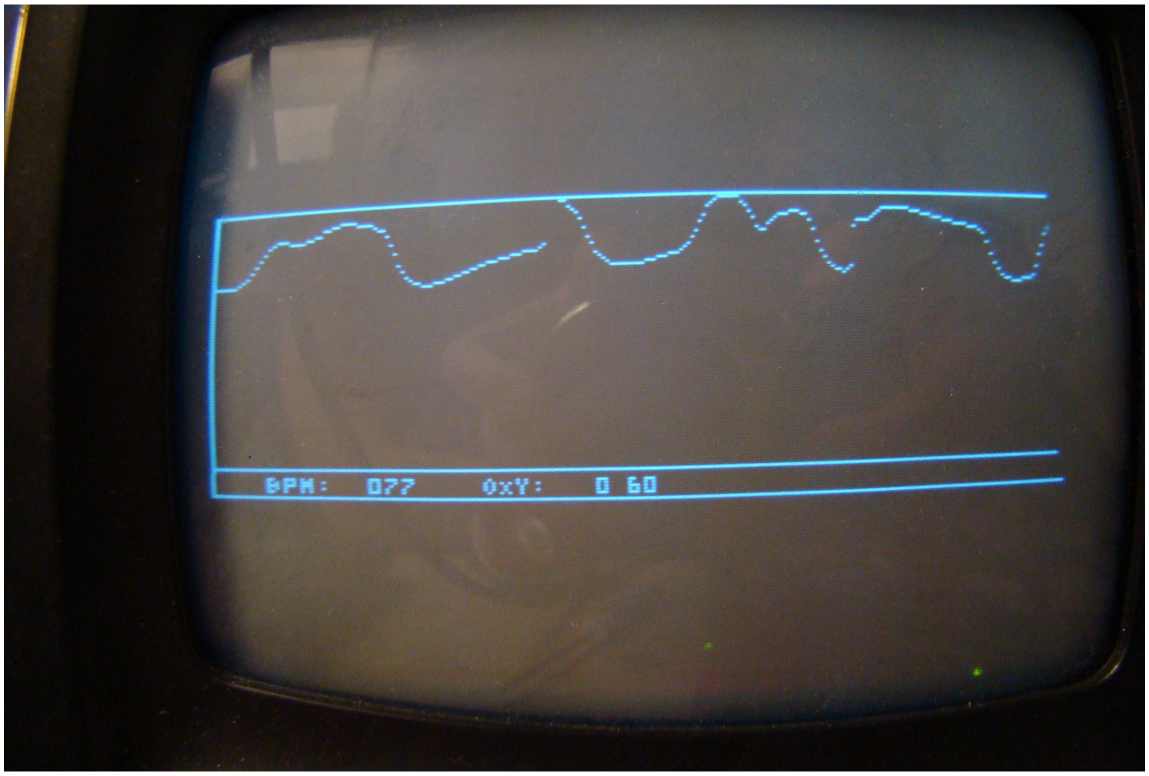
**Picture 1- Red LED amplifier**



**Picture 2 – IR Amplifier Circuit**



**Picture 3- Output of heart.c  on CRT BW TV**



**Picture 4 – Output of heart.c on CRT BW TV**

## *Future Work:*

The circuits are currently separated and require the use of two fingers. In future designs, we hope to change this to just be one finger. Additionally, the unit can be made portable with a Mega644 proto board and some batteries. Future designs of this should include a clip for the photo transistor and diode to clip onto a patients finger, and fabrication of the amplifier circuit on the same board. It is possible to make this a very portable device.

In addition, the device should be calibrated to an individual user to get an accurate ratio of oxygen level. In this design, we did not have a method to calibrate the levels appropriately. This can be done using an additional two LEDs at wavelengths of 590nm and 805nm. At these wavelengths, the absorption of oxygenated and deoxygenated blood should produce a 1:1 ratio. From this data, it can be determined what the user's actual absorption should be at the LEDs that we used to measure in this design.

## *Conclusions:*

This project was a relative success. For the most part, the functionality worked well. There were a few small bugs to work out, but we solved most of these problems through trial and error. The gain of both amplifiers can be calibrated accurately for the user, since many factors, including movement, skin thickness, and finger size all lead to different outputs for the amplifier. In our current configuration, the best way to calibrate the ratio of oxygen levels is to set the gain levels for a healthy individual to produce an oxygen level of above 95%. Using this calibration, others could check their oxygen levels as well, since we do not have the ability to calibrate the scope using the 590 and 805 nm method. Additionally, the reflection method could also be used to accommodate larger finger sizes. Both the LED and phototransistor can be bent upwards at about a 30 ° angle and the finger placed above the two. The reflections of the light waves off the finger will also measure pulse and oxygen level. Additionally, the software could be written to adapt to the input of a user. For instance, the screen only shows an 85 pixel change in the input when it scrolls. If the difference between the maximum and minimum input values is more than this, the values could be adaptively divided to fit more nicely on the screen. One additional change that might be proposed in software is a larger set of samples for the FFT.  While what we performed was adequate, especially given the memory limitations we worked with, more samples would give a better resolution than our approximate 4-5 beat increments.

## Appendix:

### *C Code:*

```
// Black and white NTSC video generation with fixed point animation
// D.6 is sync:330 ohm + diode to 75 ohm resistor
// D.5 is video:1000 ohm + diode to 75 ohm resistor
// Mega644 version by Shane Pryor
// mods by brl4@cornell.edu:
// UART SPI-mode video output from Morgan D. Jones

#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>
#include <stdlib.h>
#include <math.h>
#include <util/delay.h>
#include <avr/sleep.h>

//cycles = 63.625 * 20 Note NTSC is 63.55
//but this line duration makes each frame exactly 1/60 sec
//which is nice for keeping a realtime clock
// video timing
#define LINE_TIME 1272
#define SLEEP_TIME 1250

#define ScreenTop 80
#define ScreenBot 176


#define begin {
#define end }

#define N_WAVE          64    /* size of FFT */
#define LOG2_N_WAVE     6     /* log2(N_WAVE) */

#define FIRST_INDEX     1     /* 7 * 4.2 ~ 30 threshold bpm */


int Sinewave[N_WAVE]; // a table of sines for the FFT


extern int multfix(int a,int b);

//sync
char syncON, syncOFF;

//current line number in the current frame
```

```
volatile int LineCount;

// 96 vertical lines, 30 horizontal bytes (240 horizontal lines)
// screen buffer
char screen[30][96];

char position;

//One bit masks
char pos[8] = {0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};

//which horizontal position are we outputting first?
volatile char currentbyte;
volatile char currentbit;
///////////////

//store off current ADC value
unsigned char currentvalue;
unsigned char oxycurrentvalue;

//current BPM
unsigned char bpm;

//current OXY
unsigned char oxy;

unsigned char timetosample;
unsigned char samplenumber;

//last 128 heart samples
char heartsamples[N_WAVE];
int zeros[N_WAVE];
int heartsamplescopy[N_WAVE];

int maxsample;
unsigned char maxindex;

char oxysamples[N_WAVE];

unsigned int heartaverage;
unsigned int oxyaverage;


//=== fixed conversion macros
=========================================
#define int2fix(a)   (((int)(a))<<8)            //Convert char to fix.
a is a char
#define fix2int(a)   ((signed char)((a)>>8))    //Convert fix to int.
a is an int
```

```
#define float2fix(a) ((int)((a)*256.0))         //Convert float to
fix. a is a float
#define fix2float(a) (((float)(a))/256.0)       //Convert fix to
float. a is an int

//=================================
//3x5 font numbers, then letters
//packed two per definition for fast
//copy to the screen at x-position divisible by 4
prog_char smallbitmap[39][5] = {
      //0
    0b11101110,
      0b10101010,
      0b10101010,
      0b10101010,
      0b11101110,
      //1
      0b01000100,
      0b11001100,
      0b01000100,
      0b01000100,
      0b11101110,
      //2
      0b11101110,
      0b00100010,
      0b11101110,
      0b10001000,
      0b11101110,
      //3
      0b11101110,
      0b00100010,
      0b11101110,
      0b00100010,
      0b11101110,
      //4
      0b10101010,
      0b10101010,
      0b11101110,
      0b00100010,
      0b00100010,
      //5
      0b11101110,
      0b10001000,
      0b11101110,
      0b00100010,
      0b11101110,
      //6
      0b11001100,
      0b10001000,
      0b11101110,
      0b10101010,
      0b11101110,
      //7
      0b11101110,
      0b00100010,
      0b01000100,
      0b10001000,
      0b10001000,
      //8
      0b11101110,
      0b10101010,
      0b11101110,
      0b10101010,
      0b11101110,
      //9
      0b11101110,
      0b10101010,
      0b11101110,
      0b00100010,
      0b01100110,
      //:
      0b00000000,
      0b01000100,
      0b00000000,
      0b01000100,
      0b00000000,
      //=
      0b00000000,
      0b11101110,
      0b00000000,
      0b11101110,
      0b00000000,
      //blank
      0b00000000,
      0b00000000,
      0b00000000,
      0b00000000,
      0b00000000,
      //A
      0b11101110,
      0b10101010,
      0b11101110,
      0b10101010,
      0b10101010,
      //B
      0b11001100,
      0b10101010,
      0b11101110,
      0b10101010,
      0b11001100,
      //C
      0b11101110,
      0b10001000,
      0b10001000,
      0b10001000,
      0b11101110,
      //D
      0b11001100,
      0b10101010,
      0b10101010,
      0b10101010,
      0b11001100,
      //E
      0b11101110,
      0b10001000,
      0b11101110,
      0b10001000,
      0b11101110,
      //F
      0b11101110,
      0b10001000,
      0b11101110,
      0b10001000,
      0b10001000,
      //G
      0b11101110,
      0b10001000,
      0b10001000,
      0b10101010,
      0b11101110,
      //H
      0b10101010,
      0b10101010,
      0b11101110,
      0b10101010,
      0b10101010,
      //I
      0b11101110,
      0b01000100,
      0b01000100,
      0b01000100,
      0b11101110,
      //J
      0b00100010,
      0b00100010,
      0b00100010,
      0b10101010,
      0b11101110,
      //K
      0b10001000,
      0b10101010,
      0b11001100,
      0b11001100,
      0b10101010,
      //L
      0b10001000,
      0b10001000,
      0b10001000,
      0b10001000,
      0b11101110,
      //M
      0b10101010,
      0b11101110,
      0b11101110,
      0b10101010,
      0b10101010,
      //N
      0b00000000,
      0b11001100,
      0b10101010,
      0b10101010,
      0b10101010,
      //O
      0b01000100,
      0b10101010,
      0b10101010,
      0b10101010,
      0b01000100,
      //P
      0b11101110,
      0b10101010,
      0b11101110,
      0b10001000,
      0b10001000,
      //Q
      0b01000100,
      0b10101010,
      0b10101010,
      0b11101110,
      0b01100110,
      //R
      0b11101110,
      0b10101010,
      0b11001100,
      0b11101110,
      0b10101010,
      //S
      0b11101110,
      0b10001000,
      0b11101110,
        0b00100010,
        0b11101110,
        //T
        0b11101110,
        0b01000100,
        0b01000100,
        0b01000100,
        0b01000100,
        //U
        0b10101010,
        0b10101010,
        0b10101010,
        0b10101010,
        0b11101110,
        //V
        0b10101010,
        0b10101010,
        0b10101010,
        0b10101010,
        0b01000100,
        //W
        0b10101010,
        0b10101010,
        0b11101110,
        0b11101110,
        0b10101010,
        //X
        0b00000000,
        0b10101010,
        0b01000100,
        0b01000100,
        0b10101010,
        //Y
        0b10101010,
        0b10101010,
        0b01000100,
        0b01000100,
        0b01000100,
        //Z
        0b11101110,
        0b00100010,
        0b01000100,
        0b10001000,
        0b11101110
};


void FFTfix(int fr[], int fi[], int m)
//Adapted from code by:
//Tom Roberts 11/8/89 and Malcolm Slaney 12/15/94 malcolm@interval.com
```

```
//fr[n],fi[n] are real,imaginary arrays, INPUT AND RESULT.
//size of data = 2**m
// This routine does foward transform only
begin
    int mr,nn,i,j,L,k,istep, n;
    int qr,qi,tr,ti,wr,wi;

    mr = 0;
    n = 1<<m;   //number of points
    nn = n - 1;

    // decimation in time - re-order data
    for(m=1; m<=nn; ++m)
    begin
        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];   //for real inputs, don't need this
        //fi[m] = fi[mr];
        //fi[mr] = ti;
    end

    L = 1;
    k = LOG2_N_WAVE-1;
    while(L < n)
    begin
        istep = L << 1;
        for(m=0; m<L; ++m)
        begin
            j = m << k;
            wr =  Sinewave[j+N_WAVE/4];
            wi = -Sinewave[j];
            wr >>= 1;
            wi >>= 1;

            for(i=m; i<n; i+=istep)
            begin
                j = i + L;
                tr = multfix(wr,fr[j]) - multfix(wi,fi[j]);
                ti = multfix(wr,fi[j]) + multfix(wi,fr[j]);
                qr = fr[i] >> 1;
                qi = fi[i] >> 1;
                fr[j] = qr - tr;
                fi[j] = qi - ti;
                fr[i] = qr + tr;
                fi[i] = qi + ti;
            end
        end
        --k;
        L = istep;
    end
end


// WHOLE NUMBER ONLY VERSION
//==========================================================
int sqrtfix(int aa)
begin

    int a;
    char ahigh;
    a = aa;
    ahigh = a>>8 ;
    //
    // range sort to get integer part and to
    // check for weird bits near the top of the range
    if (ahigh >= 0x40)       //bigger than 64?
    begin
        if (a > 0x7e8f)          return 0x0b40;  // 11
        else if (ahigh >= 0x79) return 0x0b00;  // 11
        else if (ahigh >= 0x64) return 0x0a00;  // 10
        else if (ahigh >= 0x51) return 0x0900;  // 9
        else                    return 0x0800;  // 8
    end

    else if  (ahigh >= 0x10) //smaller than 64 and bigger then 16
    begin
        if (ahigh >= 0x31)      return 0x0700;  // 7
        else if (ahigh >= 0x24) return 0x0600;  // 6
        else if (ahigh >= 0x19) return 0x0500;  // 5
        else                    return 0x0400;  // 4
    end

    else                     //smaller than 16
    begin
        if (ahigh >= 0x09)      return 0x0300;  // 3
        else if (ahigh >= 0x04) return 0x0200;  // 2
        else if (ahigh >= 0x01) return 0x0100;  // 1
        else                    return 0;
    end
end


// put the MCU to sleep JUST before the CompA ISR goes off
ISR (TIMER1_COMPB_vect, ISR_NAKED)
{
      sei();
      sleep_cpu();
    reti();
}

//==================================
//This is the sync generator and raster generator. It MUST be entered
from
//sleep mode to get accurate timing of the sync pulses

ISR (TIMER1_COMPA_vect) {
      char i, screenStart, offset;

      //start the Horizontal sync pulse
      PORTD = syncON;

      //update the current scanline number
      LineCount++;

    if (LineCount == 250) {
        currentbit++;
        if (currentbit == 8) { currentbit = 0; currentbyte++; }
        if (currentbyte == 30) currentbyte = 0;
    }

      //begin inverted (Vertical) synch after line 247
      if (LineCount == 248) {
      syncON = 0b00000001;
      syncOFF = 0;
      }

      //back to regular sync after line 250
      if (LineCount == 251) {
            syncON = 0;
            syncOFF = 0b00000001;
      }

      //start new frame after line 262
      if (LineCount == 263)  LineCount = 1;

      //adjust to make 5 us pulses
      _delay_us(3);

      //end sync pulse
      PORTD = syncOFF;

      if (LineCount < (ScreenBot - 8) && LineCount >= ScreenTop) {
            //compute offset into screen array
            screenStart = LineCount - ScreenTop;
            //center image on screen
            _delay_us(4);

         // blast the data to the screen
            // We can load UDR twice because it is double-bufffered
            offset = currentbyte;
            UDR0 = screen[offset][screenStart];

         offset++;
         UDR0 = screen[offset < 30 ? offset : 0][screenStart];

            UCSR0B = _BV(TXEN0);

            for (i = 2; i < 30; i++) {
              //remove this line here because it causes video artifacts
because computing the offset
              //takes long enough that the register is ready without
the check
              //while (!(UCSR0A & _BV(UDRE0)));

           offset++;
           UDR0 = screen[offset < 30 ? offset : offset -
30][screenStart];
            }

            UCSR0B = 0;
    }

      else if (LineCount < ScreenBot && LineCount >= ScreenTop) {
         screenStart = LineCount - ScreenTop;

         //center image on screen
            _delay_us(5.53);


            // blast the data to the screen
            // We can load UDR twice because it is double-bufffered
            UDR0 = screen[0][screenStart];
         UDR0 = screen[1][screenStart];

            UCSR0B = _BV(TXEN0);

            for (i = 2; i < 30; i++) {
              while (!(UCSR0A & _BV(UDRE0)));

           UDR0 = screen[i][screenStart];
            }

            UCSR0B = 0;
      }
```

```
}

//==================================
//plot one point
//at x,y with color 1=white 0=black 2=invert
void video_pt(char x, char y, char c) {
      //each line has 30 bytes
      //calculate i based upon this and x,y
      // the byte with the pixel in it
      char i = (x >> 3);

      if (c==1)
        screen[i][y] = screen[i][y] | pos[x & 7];
    else if (c==0)
        screen[i][y] = screen[i][y] & ~pos[x & 7];
    else
        screen[i][y] = screen[i][y] ^ pos[x & 7];
}

//==================================
//plot a line
//at x1,y1 to x2,y2 with color 1=white 0=black 2=invert
//NOTE: this function requires signed chars
//Code is from David Rodgers,
//"Procedural Elements of Computer Graphics",1985

void video_line(char x1, char y1, char x2, char y2, char c) {
      int e;
      signed int dx,dy,j, temp;
      signed char s1,s2, xchange;
    signed int x,y;

      x = x1;
      y = y1;

      //take absolute value
      if (x2 < x1) {
            dx = x1 - x2;
            s1 = -1;
      }

      else if (x2 == x1) {
            dx = 0;
            s1 = 0;
      }

      else {
            dx = x2 - x1;
            s1 = 1;
      }
      if (y2 < y1) {
            dy = y1 - y2;
            s2 = -1;
      }

      else if (y2 == y1) {
            dy = 0;
            s2 = 0;
      }

      else {
            dy = y2 - y1;
            s2 = 1;
      }

      xchange = 0;

      if (dy>dx) {
            temp = dx;
            dx = dy;
            dy = temp;
            xchange = 1;
      }

      e = ((int)dy<<1) - dx;

      for (j=0; j<=dx; j++) {
            video_pt(x,y,c);

            if (e>=0) {
                  if (xchange==1) x = x + s1;
                  else y = y + s2;
                  e = e - ((int)dx<<1);
            }

            if (xchange==1) y = y + s2;
            else x = x + s1;

            e = e + ((int)dy<<1);
      }
}

//==================================
// put a small character on the screen
// x-coord must be on divisible by 4
// c is index into bitmap
void video_smallchar(char x, char y, char c) {
      char mask;
      char i = (x >> 3);
      if (x == (x & 0xf8)) mask = 0x0f;     //f8
      else mask = 0xf0;

      uint8_t k = pgm_read_byte(((uint32_t)(smallbitmap)) + c*5);
      screen[i][y]   = (screen[i][y] & mask) | (k & ~mask);

      k = pgm_read_byte(((uint32_t)(smallbitmap)) + c*5 + 1);
      screen[i][y+1] = (screen[i][y+1] & mask) | (k & ~mask);

      k = pgm_read_byte(((uint32_t)(smallbitmap)) + c*5 + 2);
      screen[i][y+2] = (screen[i][y+2] & mask) | (k & ~mask);

      k = pgm_read_byte(((uint32_t)(smallbitmap)) + c*5 + 3);
      screen[i][y+3] = (screen[i][y+3] & mask) | (k & ~mask);

      k = pgm_read_byte(((uint32_t)(smallbitmap)) + c*5 + 4);
      screen[i][y+4] = (screen[i][y+4] & mask) | (k & ~mask);
}

//==================================
// put a string of small characters on the screen
// x-cood must be on divisible by 4
void video_putsmalls(char x, char y, char *str) {
      char i;
      x = x & 0b11111100; //make it divisible by 4

      for (i = 0; str[i] != 0; i++) {
            if (str[i] >= 0x30 && str[i] <= 0x3a)
                  video_smallchar(x, y, str[i] - 0x30);

                   else video_smallchar(x, y, str[i]-0x40+12);

            x += 4;
      }
}

//==================================
// set up the ports, timers, ADC, USART.
// initialize variables
// enable interrupts

void initialize() {
  // init timer 1 to generate sync
  // TIMER 1: OC1* disconnected, CTC mode, fosc/1 (20MHz), OC1A and
OC1B
  // interrupts enabled
  char i;

  TCCR1B = _BV(WGM12) | _BV(CS10);
```

```
OCR1A = LINE_TIME;  // time for one NTSC line
OCR1B = SLEEP_TIME; // time to go to sleep
TIMSK1 = _BV(OCIE1B) | _BV(OCIE1A);

//init ports
DDRD = 0x03;        //video out

// USART in MSPIM mode, transmitter enabled, frequency fosc/4
UCSR0B = _BV(TXEN0);
UCSR0C = _BV(UMSEL01) | _BV(UMSEL00);
UBRR0 = 1;

//initialize synch constants
LineCount = 1;
syncON = 0b00000000;
syncOFF = 0b00000001;

//initialize display
video_line(0,0,239,0,1);
video_line(0,86,239,86,1);
video_line(0,95,225,95,1);
video_putsmalls(20, 89, "BPM:");
video_putsmalls(72, 89, "OXY:");

timetosample = 0;
samplenumber = 0;

maxsample = 0;
maxindex = 0;

currentvalue = 0;

//init the A to D converter
//channel seven / left adj / EXTERNAL Aref
ADMUX = (1<<ADLAR);


//enable ADC and set prescaler to 1/128*16MHz=125,000
//and clear interupt enable
//and start a conversion
ADCSRA = (((1<<ADEN) | (1<<ADSC)) + 7);

// one cycle sine table
//  required for FFT
for (i=0; i<N_WAVE; i++)
  Sinewave[i] = float2fix(sin(6.283*((float)i)/N_WAVE));

// Set up single video line timing
sei();
set_sleep_mode(SLEEP_MODE_IDLE);
```

```
  sleep_enable();
}

int main() {
  initialize();

  char i;

  for(;;) {

    // once per frame processing
    if (LineCount == 232) {
           //start another conversion
           ADCSRA |= (1<<ADSC);
        while ((ADCSRA & (1 << ADSC)));

           //get the current sample
        currentvalue = (ADCH >> 2);

           ADMUX = ((1<<ADLAR) | 0x01);
        ADCSRA |= (1<<ADSC);
        while (ADCSRA & (1 << ADSC));

        //get the current sample
        oxycurrentvalue = (ADCH >> 2);

           ADMUX = (1<<ADLAR);

        //calculate current position in buffer
        position = (currentbyte << 3) + currentbit;

        //erase the old samples at this position
        video_line(position, 1, position, 85, 0);

        //draw signal
        if (oxycurrentvalue >= 0 && oxycurrentvalue <= 86)
          video_pt(position, oxycurrentvalue, 1);

           else if (oxycurrentvalue < 0)
          video_pt(position, 0, 1);

           else
          video_pt(position, 86, 1);


        if (timetosample < 14)  timetosample++;

        else if (samplenumber < N_WAVE) {
          timetosample = 0;
          heartsamples[samplenumber] = currentvalue;
          oxysamples[samplenumber++] = oxycurrentvalue;
        }

        else {
          for (i = 0; i < N_WAVE-1; i++) {
            heartsamples[i] = heartsamples[i+1];
            heartsamplescopy[i] = int2fix(heartsamples[i]);

              oxysamples[i] = oxysamples[i+1];
          }

             heartsamples[N_WAVE-1] = currentvalue;
      heartsamplescopy[N_WAVE-1] = int2fix(heartsamples[N_WAVE-1]);

      oxysamples[N_WAVE-1] = oxycurrentvalue;

      timetosample = 0;
           maxsample = 0;
            maxindex = 0;

            for (i = 0; i < N_WAVE; i++)
             zeros[i] = 0;

           // compute BPM from samples
           FFTfix(heartsamplescopy, zeros, LOG2_N_WAVE);


           for (i = FIRST_INDEX; i < N_WAVE; i++) {
            heartsamplescopy[i] = sqrtfix(multfix(heartsamplescopy[i],
heartsamplescopy[i]) + multfix(zeros[i], zeros[i]));

            if (heartsamplescopy[i] > maxsample) {
              maxindex = i;
              maxsample = heartsamplescopy[i];
            }
          }

        bpm = fix2int(multfix(int2fix(maxindex),float2fix(60.0 /
14.0)));

           // compute oxygen percentage from samples
           heartaverage = 0;
           oxyaverage = 0;

        for (i = 0; i < N_WAVE; i++) {
              heartaverage += heartsamples[i];
              oxyaverage += oxysamples[i];
           }

           heartaverage /= N_WAVE;
           oxyaverage /= N_WAVE;

           oxy = (oxyaverage * 100) / (heartaverage);

        video_smallchar(44, 89, bpm / 100);
          video_smallchar(48, 89, bpm % 100 / 10);
          video_smallchar(52, 89, bpm % 100 % 10);

           video_smallchar(100, 89, oxy / 100);
           video_smallchar(108, 89, oxy % 100 / 10);
           video_smallchar(112, 89, oxy % 100 % 10);
       } // else
     } //if line
   }  //for
}  //main
```

**Figure 2- heart.c**

```
;*******************************************************************
*********
;*
;* FUNCTION
;*   muls16x16
;* DECRIPTION
;*   Signed multiply of two 16bits numbers with 16 bits result.
;* USAGE
;*   r25:r24 = r23:r22 * r25:r24
;*******************************************************************
*********
;int multfix(int a,int b)

.global multfix
multfix:
      ;input parameters are in r23:r22(hi:lo) and r25:r24

      ;b aready in right place -- 2nd parameter is in r22:23


    mov  r20,r24 ;load a -- first parameter is in r24:25
      mov  r21,r25


      muls r23, r21    ; (signed)ah * (signed)bh
      mov   r25, r0         ;r18, r0"
      mul   r22, r20        ; al * bl"
      mov  r24, r1      ;movw    r17:r16, r1:r0"

      mulsu r23, r20   ; (signed)ah * bl
      add   r24, r0         ;r17, r0"
      adc   r25, r1         ;r18, r1"

      mulsu r21, r22   ; (signed)bh * al
      add   r24, r0         ;r17, r0"
      adc   r25, r1         ;r18, r1"

      clr  r1               ; required by GCC


      ;return values are in 25:r24 (hi:lo)

      ret
```

**Figure 3 – multASM.S**

## *Hardware Schematics:*

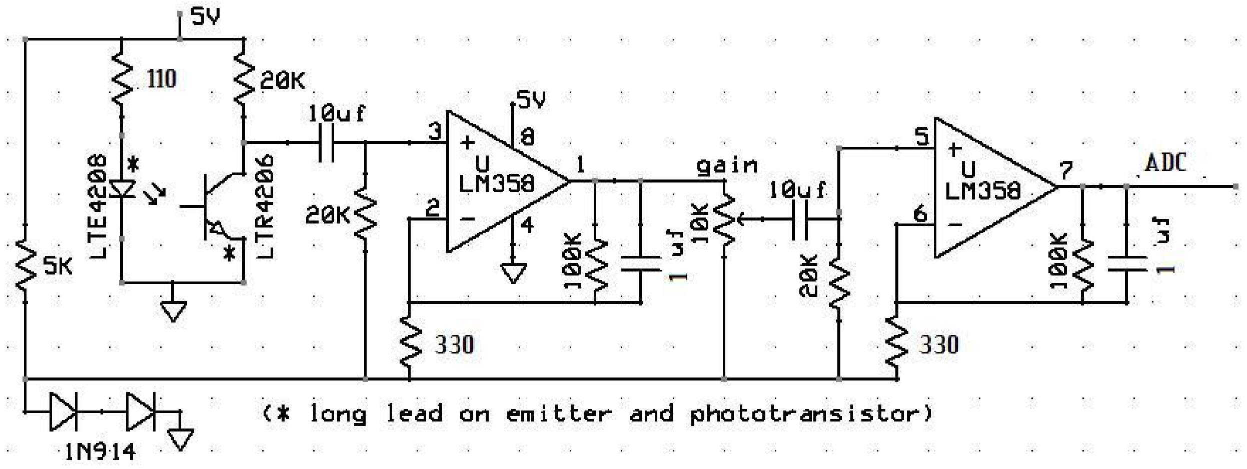


**Figure 4- Hardware Schematic**

## *References:*

1. http://instruct1.cit.cornell.edu/courses/ee476/Math/index.html
2. http://instruct1.cit.cornell.edu/courses/ee476/video/index.html
3. http://en.wikipedia.org/wiki/Pulse_oximeter
4. http://en.wikipedia.org/wiki/NTSC
5. http://www.oximetry.org/pulseox/principles.htm
6. http://www.picotech.com/experiments/calculating_heart_rate/