ECE 476 Final Design Project: Polygraph

Jordan Crittenden (jsc59), Edwin Lai (ecl37)

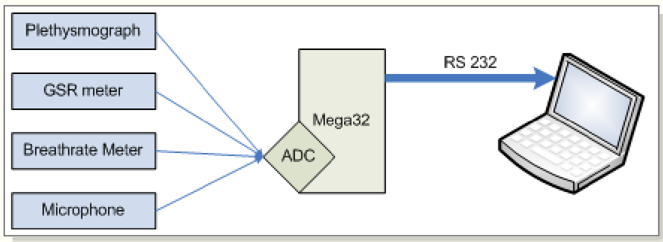
Introduction

A polygraph (often and incorrectly called a 'lie detector') is a machine which plots in real time several human biological signals such as pulse rate, galvanic skin resistance (GSR), blood pressure, and breathing rate. This machine, in conjunction with a certified examiner, is then used to analyze a subject's stress during interrogation with the intent of distinguishing truth from lying. These machines can cost anywhere from hundreds to thousands of dollars. We attempted to build one for $50.

For our design project we constructed a polygraph which measured pulse rate, GSR, and breathing rate, and performed a discrete cosine transform (DCT) on the subject's voice in the hope of measuring a deviation in the fundamental frequency (another indicator of stress). The measurements were sampled, analyzed, and transmitted to a computer for further analysis using an ATMEL AVR Mega32 microcontroller.

High Level Design

The basic design of the polygraph is shown in the diagram to the right. The four analog circuits: plethysmograph (pulse rate meter), GSR meter, breathing rate meter, and audio preamplifier, are attached to the first four pins of the Mega32 analog to digital converter (ADC). The polygraph can then be run in one of two modes: pulse, GSR, and breath sampling or audio sampling. This is simply because the microcontroller is the limiting factor in processing power. In the first case, the microcontroller samples the three signals in a polling loop. The samples are taken at a fast enough rate (100Hz) that it appeared to be sampling each signal in real time. In the second case, the microcontroller exclusively samples audio at 8kHz.

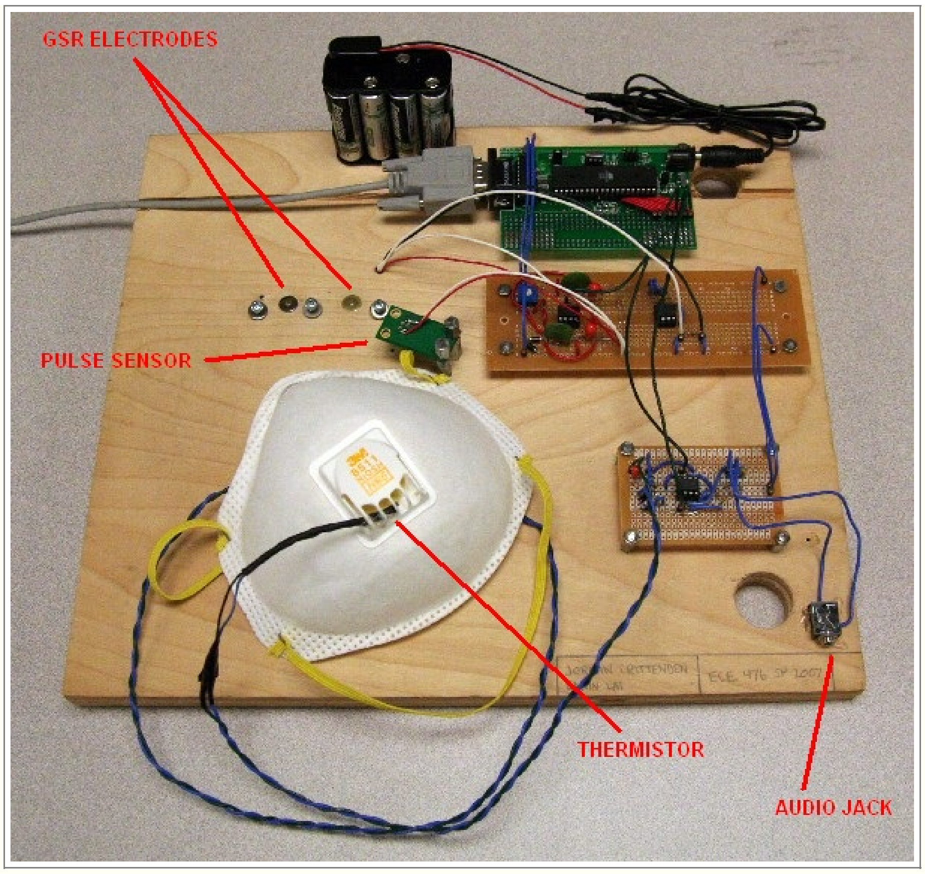


In PGB (pulse, GSR, breath) mode, the microcontroller waits for a signal from a concurrent MATLAB program (running on a laptop) indicating a data request. The data is sent to the laptop over an RS-232 link via a serial to USB adapter and it is plotted in real time. The MATLAB program performs the signal processing, such as computing pulse and breathing rates as well as skin resistance.

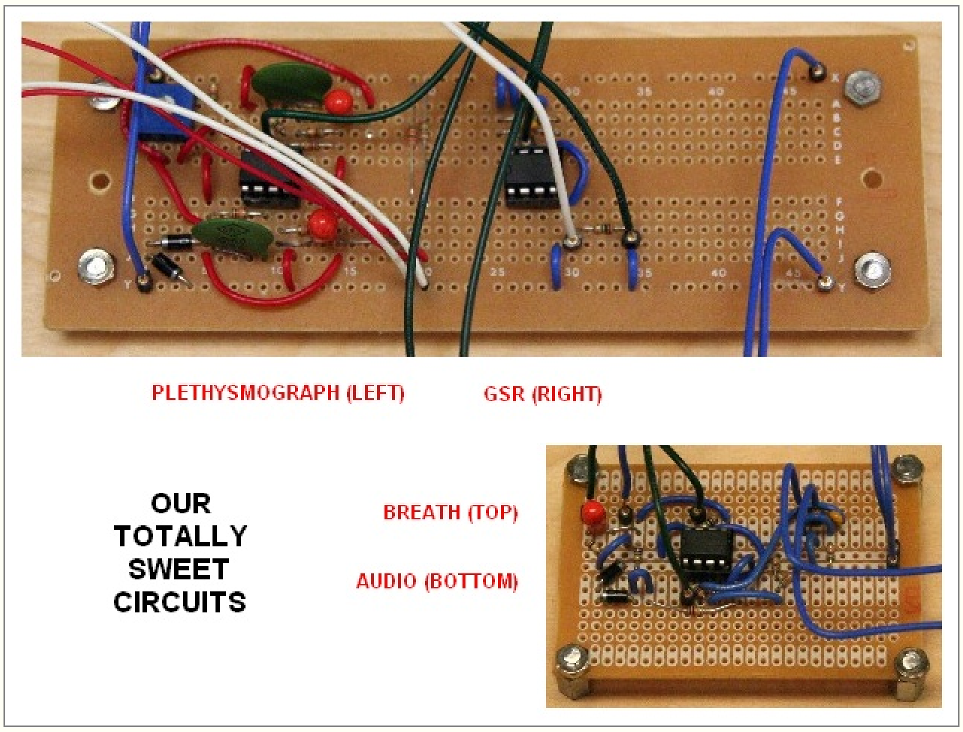
In audio sampling mode, once a set of 32 sequential samples is acquired, the microcontoller performs a 32 point DCT on the audio signal to extract the frequency decomposition of the subject's voice. At 8khz, 32 samples cover 4ms of speech and are enough to capture the primary features of human speech. The signal power is computed and the average DCT is updated with the new data. A concurrent MATLAB program periodically polls the microcontroller for the current DCT coefficients. These values are then plotted in a time-based heatmap, along with the signal power and average DCT.

Hardware Design

This design project was heavily analog. Each measurement component required amplification of a signal generated by some sensor. This presented many difficulties. For example, the pulse signal depends strongly on the particular subject, the finger being used, ambient lighting, etc. To combat these parasitic effects, we constructed an examination board, shown below and required that the subject being tested wear a specialized sensor facemask, also shown below. The board restricts movement of the subjects hand, which improves both the pulse and GSR signals. Similarly, the facemask, on which we mounted the breath sensor and microphone, has the dual purpose of maintaining the positions of the sensors relative to the subjects face and improving both the breath and audio signals by reducing noise such as outside audio stimulus.



Closeups of the circuits we built are shown below. Note that we made a special effort to lay the circuits out in such a way as to take up little physical space and to avoid any obstruction.

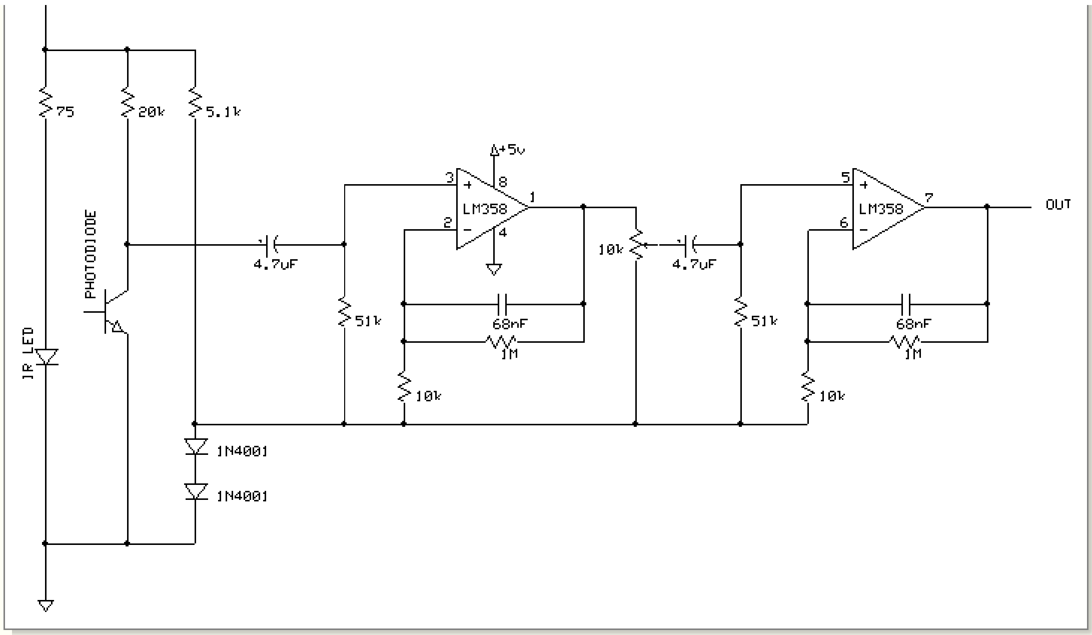


Plethysmograph

The plethysmograph that we built works on the principle that the amount of transmitted infrared light through a subject's finger is a function of the blood flow in the finger. That is, when the heart pumps there is an influx of blood into the tip of the finger. This leads to a decrease in the light received by the infrared sensor, which can be measured. Heart rate is a key component in the polygraph since under certain stressful conditions, the subject's heart rate will increase.

The circuit shown amplifies this signal by approximately 10000x in two 100x steps (the gain, in fact, can be optimally tweaked using the 10k trimpot) and filters out unwante

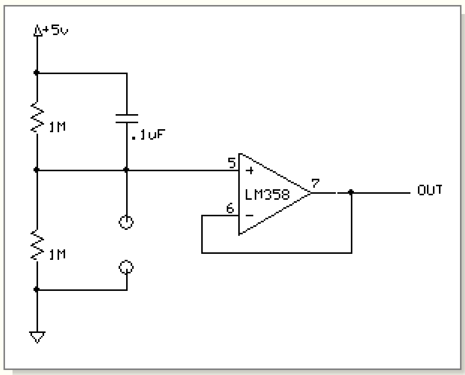
and capacitor values so that we could use lab components. Note that the capacitors are tantalum so that they can withstand some reverse bias.



The IR LED and photodiode sensor were mounted opposing each other on a the polygraph board. A hole in the polygraph board was drilled for the IR emitter to eliminate any interference. A measurement is acquired by having the subject places his finger between the elements and remain reasonably still.

GSR Meter

Perhaps the most common polygraph instrument, a GSR meter measure the subject's skin resistance. Under stress a person will begin to perspire, reducing the resistance measured between two electrodes.



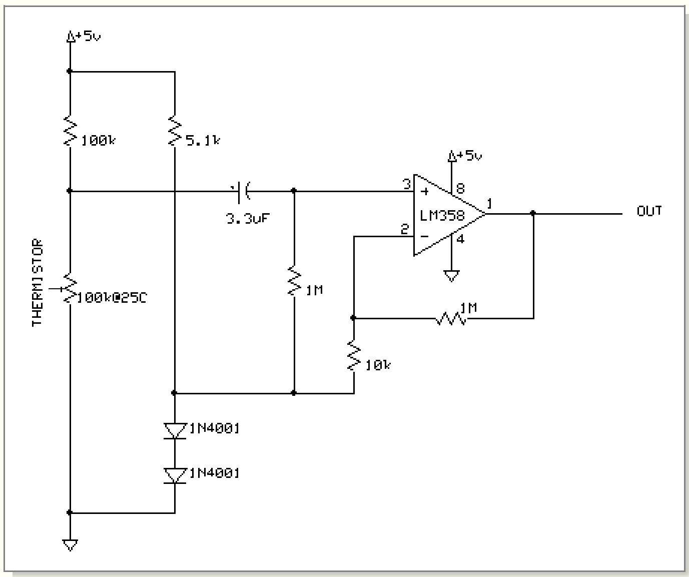
The circuit, shown to the right, is a simple voltage divider followed by a voltage follower. The resistors were chosen to be within the same magnitude as the subject's resistance so that there will significant variation in the output signal. We also set the equilibrium voltage at the intermediate node to 2.5V by pairing the resistors. Additionally, since this is the only part of the polygraph that sends current through the subject, we took careful care to chose a moderate upper voltage of 5V.

The subject touches the two electrodes (circles in the schematic) with the tips of the middle and ring fingers. When the resistance across the electrodes falls, the voltage at the intermediate node also falls because the parallel resistance in the bottom half of the divider is reduced.

Special thanks to Professor Bruce Land for donating professional medical electrodes and electrode gel for our testing.

Breathing Rate Meter

The breathing rate meter has a similar design to that of the plethymograph, except that in this case, we use a thermistor as our sensor (thanks to Professor John Belina for this idea). The thermistor is placed near the subject's mouth using a facemask such that upon exhaling the temperature increases (after a brief amount of time, or upon inhaling, the temperature decreases to it's original value).

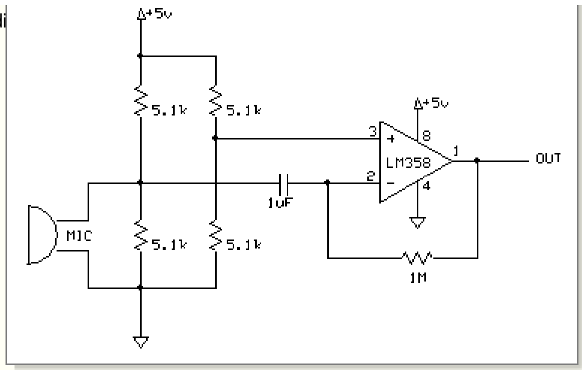


The circuit shown amplifies this signal 100x and biases it up to 1.4V using two 1N4001 diodes. In our tests, we noticed some clipping on the upper rail. Given more time, we might have modified the design. As it is, we could tell when a breath occurred, so we did not pursue this problem.

Audio Preamplifier

The audio preamplifier (credit to Rishi Sinha and William Chung) produces a non-di of the circuit is to amplify small deviations in the differential inputs by perturbing the lower half of the voltage divider attached to the inverting input.

The Microphone acts like a voltage source and it biased by 2.5V. Any variation from the 2.5V base line is then amplified by the LM358 by a factor of 200x.



Power Supply

A rechargeable battery pack was used to supply the power for our circuits. It consisted of 8 AA NiMH Rechargeable batteries and standard connector. This was firstly due to electrical isolation concerns. Secondly, 8 AA batteries in series had far less internal impedance than a single 9V battery as well as 5 times the life. Thirdly, it was rechargeable which drastically reduced money out of our pockets.

Software Design

The software design consists of four programs: two C programs for the two operating modes (these reside on the microcontroller) and two MATLAB programs (which run on the laptop). The function of the microcontroller programs is primarily to sample the various signals and pass them on to MATLAB for plotting. The MATLAB program then takes the signals, smoothes them through low-pass filters, plots them, and performs some analysis, such as computing pulse and breathing rates and their long term averages and standard deviations.

Mega32 Code

The C programs running on the Mega32 take advantage of the versatility of the ADC. In both modes, we chose to set the A/D sampling prescaler to 1/128 of the 16MHz clock, or 125kHz. At this speed, we could easily make the 8kHz sampling deadline in the audio case (the PGB case is trivial, even in the event of a change in the sampling channel which takes 26 ADC cycles).

In PGB mode, a samples is recorded every 10ms in the Timer0 ISR. We then use a polling loop to choose which channel to sample from next. The values are overwritten on each sampling pass, so most samples are lost. Since these signals are very low frequency, this is not a problem.

In audio mode, a vector of 32 samples is collected, at which point a flag is raised and audio processing begins. The DCT code written by Professor Land combines lightweight lower-order DCT-II transforms into a fast 32 point transform. Once the coefficients are calculated, we compute signal power as the sum of the absolute values of the AC coefficients and update a running average spectrum.

MATLAB Code

High Level

Our MATLAB scripts interface with the MEGA32 via an RS-232 serial connection that is established in the start of the script. In both scripts, the basic loop then performs data capture, analysis, and plotting. In each iteration of the loop, the script requests data from the MEGA32, which responds by sending a vector of data (in PGB mode, this vector containing the pulse, GSR, and breath signals; in audio mode, the vector holds the 32 DCT coefficients). An interesting property of the serial connection that we establish is that the time between transfers is not required to be (and is not in practice) uniform. Rather, it is dependent on the time spent by MATLAB drawing the plots and updating computations. The consequences of this behavior are that the horizontal axes in the various figures do not correlate directly with real time. In our rate computations, this problem is mitigated by using the tic and toc commands, which reintroduce wall time. But for general viewing, time axes cannot be completely trusted. We found, however, that the deviations are negligible for the most part, and do not significantly affect the accuracy of the figures.

Low Level - PGB Mode

After capture, the pulse and GSR signals are filtered by an 8 point Gaussian filter and the breath signal by a 10 point uniform filter. The signals are then plotted in a MATLAB figure. The plotting is done using plot handles instead of using the conventional plot function. This is due to the overhead involved in the full plot command, such as axis scaling and redrawing. We chose this implementation because regenerating all three graphs is too costly to achieve in real-time. In fact, we tried to implement a scrolling feature, which scrolls the signals across the screen as they were being read in. This turned out to be far too CPU intensive and severely hampered the program.

Before analyzing the data, we allow some arbitrary time for the system to warm up. For example, it can take up to 5 seconds for the pulse signal transient to stabilize before useful data can be captured. Once the warm up time elapses we then perform analysis on the data, such as computing instantaneous breath and pulse rates. The instantaneous breath rate is calculated based on a moving window of the breath signal. This signal corresponds to when the subject breathes. That is, when the subject breathes in, the voltage is high, when the subject breathes out, the voltage is low. So when there is a peak in voltage, it indicates that the subject took a breath. We implemented a scheme to capture the peak values of the signal by using an open source peak detection function. By then taking the difference in times at which those peaks occur, we calculate the period between peaks � or the rate. To convert this rate to wall time, we used MATLAB�s tic and toc functions to clock the time frame of the moving window. We then plot the peaks of the signal in our figure by again using handles. The average breath rate and standard deviation are both calculated with each sample on all the sample points captured thus far. The pulse rate is calculated in a similar fashion, however in this case, the moving window is narrower since a person's pulse rate is faster than a person's breath rate. In this case, a high voltage indicates a heart beat. The average pulse rate and standard deviation are also calculated in real time with every sample.

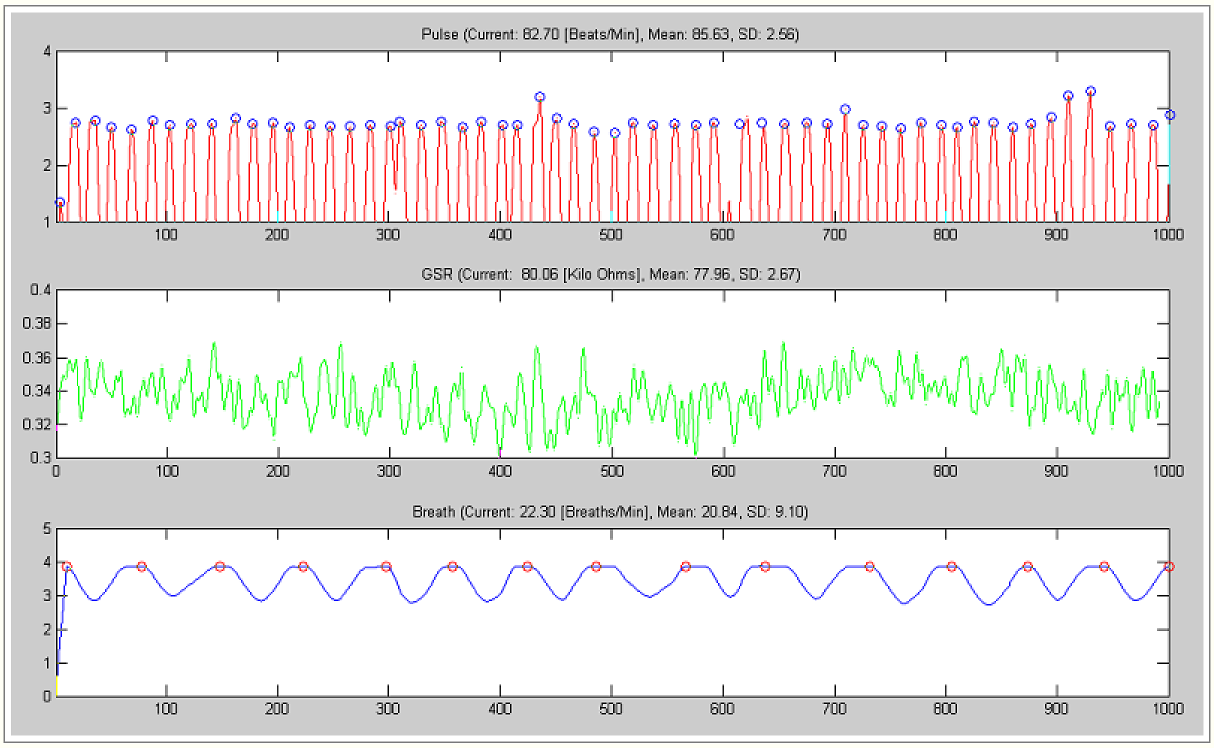
The GSR is determined more easily than the rates since it is not periodic. Using the voltage divider and parallel resistance laws, we determine the skin resistance. The mean GSR and standard deviation of the GSR are also calculated for a set of values equal to the span of the plot range.

Low Level - Audio Mode

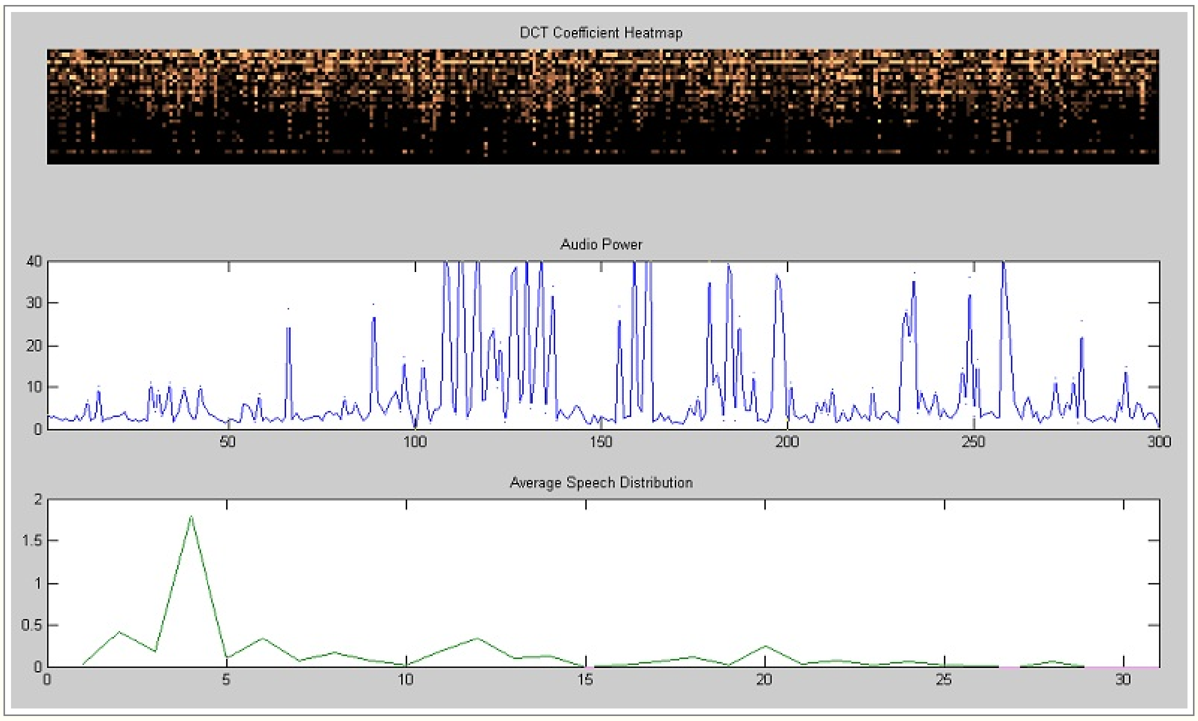
Once the vector of coefficients is captured, a time-base heat map is updated. This image shows the relative magnitudes of the DCT coefficients changing through times. The horizontal axis is time and the vertical axis is the index of the coefficient (lower frequencies at the top, higher frequencies at the bottom). In addition, the power that was computed in the Mega32 code is also plotted. Every 100 time steps, the average DCT spectrum (also calculated in the Mega32 code) is transmitted instead of the current DCT) and is plotted in the lower figure.

Results

Screen captures of the PGB and audio mode MATLAB programs are shown below. These show a typical (albeit set-up) run of the programs for a calm person. We are very pleased with the accuracy of these plots insofar as we can extract pulse and breathing rates and measure skin resistance.



Screen capture of PGB mode in action



Screen capture of Audio mode in action

Speed

The polygraph is capable of executing in real-time, which is vital to its usefulness as a lie detector. The signal sampling, although sequential, is quick enough to approximate real-time operation. Similarly, the RS-232 link, although serial, is fast enough that no delay between a heartbeat and a pulse signal on the computer is noticeable. The only significant delays occur in the MATLAB processing. In particular, MATLAB had to filter the signals and draw the plot. We pass the pulse and GSR signals through a 8 point filter and the breath signal through a 10 point filter. These filters necessarily introduce a very small delay between a peak in the original signal and a peak in filtered signal. However, since the delay is smaller than the pulse period, it does not impair the polygraph performance. Additionally, there is also a small amount of flicker in the MATLAB plots, which is due to MATLAB's plotting routines. We cannot improve the quality beyond what was discussed in the Software Design section.

Accuracy

Much of the work spent on the final design was aimed at improving the accuracy of the biosignals. Over the course of the development of the project we made steady progress to this end. Whereas at first, for example, the pulse signal represented a fraction of the total signal from the plethysmograph, by the end of development we were able to capture a very clean, very steady heart rate. Much of this can be credited to the design of the final polygraph board and to the signal processing performed in MATLAB. The stability of the breath signal from the thermistor also improved dramatically when we introduced the facemask.

Nevertheless, the final machine is still not completely robust. In particular, if the subject moves his hand during testing, the pulse and GSR signals can be temporarily corrupted. In addition, the breathing sensor is much more accurate when the subject is not speaking. Although unfortunate, it is a tradeoff with having a more complicated chest-stretch-sensor setup. The microphone is also highly susceptible to outside noise. For these reasons, an accurate test would need to be conducted under well-monitored conditions.

Safety

DO NOT POWER THIS MACHINE USING WALL CURRENT!

Safety was a very large concern in our project. Since epidermal contact is necessary for measuring GSR, we took care to make the project completely electrically separate from any wall current. This was accomplished by using batteries to power the entire system, including the laptop running the MATLAB script. Additionally, we chose a reasonable voltage of 5V for the top rail of the voltage divider. Furthermore, the contact that is made with the body is strictly between two fingers of the same hand to reduce the chance of any current passing through the heart or head.

If used in a live system, we would also require that the facemask be cleaned or disposed of between uses to promote health and prevent the spread of infection.

Interference

Our project has no RF component so interference with other projects is not an issue.

Usability

There are two aspects of usability with respect to our project: usability by the subject and usability by the examiner.

The polygraph is very usable by the subject. That is, the hand cradle and face mask fit most adults. As previously mentioned, if the subject has a weak pulse or shallow breath, the corresponding signals may not be measured correctly, but these are isolated cases.

The usability by the examiner is a more complicated issue. Whereas the polygraph can measure our signals with reasonable accuracy, its effectiveness at unearthing lies is primarily the responsibility of the examiner. We provide, for instance, pulse and breathing rate averages and deviations, but it is up to the examiner to analyze these numbers. Although drastic deviations from the mean of the signals may indicate lies or simply normal variation, we can say with certainty that we do not have the training to make this distinction.

Conclusions

Success

We consider our design a partial success. Our initial goals, as stated in our design proposal were to build a machine which could measure four signals: audio, pulse, GSR, and breathing, and use these signals to distinguish between truth and lying. We were completely successful in measuring three signals (pulse, GSR, and breathing) and partially successful in measuring audio (the preamplifier and DCT work but the analysis is shaky). But as far as detecting lies, we got no further than computing averages and deviations of the various signals.

In retrospect, we should have set our initial goals a little lower. It is difficult enough to get clean, stable signals from a human. To try to make a machine detect lies is nearly impossible. Even government organizations who have professional equipment do not rely solely on a machine. Instead, trained operators ('certified polygraph examiners') analyze the information.

Another goal at which we hinted in our proposal was analysis of speech to detect deviations in the fundamental frequency. We recognize now that this aspect of the project could have been a separate project in itself. Furthermore, it is not the position of the America Polygraph Association that voice analysis is even a valid indicator of stress. Thus, we do not view its absence as a terrible downside.

If we were to do this project again, we would have built the polygraph board much earlier in the semester. It turned out to provide not only an attractive layout, but also a valuable debugging and testing platform.

Standards

There are no applicable standards to our project that we are aware of.

Intellectual Property

Many of the components in our design are reused. We have noted these in this report. Specifically, the plethysmograph and audio preamplifier circuits are borrowed, as is the DCT code and MATLAB peakdet function.

Ethical Considerations

For the duration this project, we tried to uphold the IEEE Code of Ethics. We made special effort, as mentioned before, to isolate the user from all possible contact with power grid ground and even to avoid small shocks from batteries. We disclosed all borrowed designs, both hardware and software, even when modified (these are listed in the Borrowed Designs section as well as when mentioned above). We claimed no more success than was achieved and did not attempt to conceal the fact that we did not meet some of our initial goals, specifically automatically detecting lies and measuring deviation in the audio fundamental frequency. We were not bribed in any way nor did we attempt to bribe any other party for gain of any kind. We attempted to further our own knowledge concerning aspects of electrical engineering and biology, specifically measurement of biological signals, signal processing, and embedded programming, and to explain these things to those who inquired. In no way did we seek to discriminate against anyone (except, perhaps, those who attempt to deceive for personal gain).

There is, of course, the issue that, by design, polygraphs and the process of lie detection is psychologically demanding. The entire reason that these machines exist is because people attempt to subvert governments. For our project, these issues did not arise, primarily because this is an academic pursuit and because we did not, for the most part, attempt to interpret the signals that we measured.

Legal Considerations

Although we could find no laws pertaining to voltage / current limits on a human, we took care to prevent electric shock by avoiding and path to power grid ground.

Appendix A: Program Listing

The code is divided into two pairs of components, one for the pulse, GSR, and breathing components and another for the audio. Each pair consists of a C program running of a Mega32 and a MATLAB program running on a computer.

Pulse, GSR, Breath

C

```
/*** INCLUDES ***/
#include <mega32.h>
#include <stdio.h>
#include <delay.h>
#include <math.h>

/*** DEFINES ***/

/*** PROTOTYPES ***/
void initialize();

/*** GLOBALS ***/
char val;

// current measurement state
enum {PULSE, GSR, BREATH} measure, nextmeasure;

// current signal voltages
char pulseV = 0;
char gsrV = 0;
char breathV = 0;

// enter every 100Hz
interrupt [TIM0_COMP] void getAD(void)
{
 // sample A/D and store
 val = ADCH;
 // start a/d converter
 switch(measure)
 {
  case PULSE:
   pulseV = val << 1;
   nextmeasure = GSR;
   break;
  case GSR:
   gsrV = val;
   nextmeasure = BREATH;
   break;
  case BREATH:
   breathV = val;
   nextmeasure = PULSE;
   break;
 }

 switch(nextmeasure)
 {
  case PULSE:  ADMUX = 0b01100001; break;
  case GSR:    ADMUX = 0b01100010; break;
  case BREATH: ADMUX = 0b01100011; break;
 }
 measure = nextmeasure;
 ADCSR.6 = 1;
}

// send signals to MATLAB for drawing
void transmit()
{
 printf("%d ", (int)pulseV);
 printf("%d ", (int)gsrV);
 printf("%d ", (int)breathV);
 printf("\r"); // end packet
 PORTD.7 = ~PORTD.7;
}

// read A/D converter and communicate with MATLAB
void main(void)
{
 char inchar;

 initialize();

 while(1)
 {
  // when signaled by MATLAB, return current values
  if (UCSRA.7)
  {
   inchar = UDR;
   if (inchar=='s') {
    transmit();
   }
  }
 }
}

// setup
void initialize(void)
{
 // comm indicator LED
 DDRD.7 = 1;
 PORTD.7 = 0;

 // serial RS-232  setup for debugging using printf, etc.
 UCSRB = 0x18;
 UBRRL = 103;

 // set up timer0 to sample a/d at about 100Hz
 TCCR0 = 0b00000101;
 TIMSK = 0b00000010;
 OCR0 = 156;

 // set up a/d for external Vref, channel 0
 // channel zero / left adj / AVcc Reference
 // A1=PULSE, A2=GSR, A3=BREATH
 ADMUX = 0b01100001;
 // enable ADC and set prescaler to 1/128*16MHz=125kHz
 // and clear interupt enable
 // and start a conversion
 ADCSR = 0b11000111;

 measure = PULSE;

 // and start the show
 #asm("sei")
}
```

MATLAB

```
% CLOSE LEFTOVER SERIAL CONNECTIONS
clear all
try
        fclose(instrfind) %close any bogus serial connections
end

% SET UP SERIAL CONNECTION THROUGH RS-232
% set its rate to 9600 baud
% SR830 terminator character is carriage return (ACSII 13)
s = serial('COM15', 'baudrate', 9600, 'terminator', 13);
fopen(s);
disp('serial communication setup complete');

% CONSTANTS
WARMUP_TIME             = 200;
PULSE_WINDOW_SIZE       = 100;
BREATH_WINDOW_SIZE      = 300;
PLOT_SIZE               = 1000;
PDELTA                  = 0.25;
BDELTA                  = 0.25;

vect = zeros(1000,3);

% SET UP PLOTS AND GRAB HANDLES
figure(1);

subplot(3, 1, 1);
hPulse = plot(vect(:,1), 'r');
hold on;
hpPeaks = scatter(vect(:,1), vect(:,2));
hold off;
axis([1 1000 1 4]);
hPulseTitle = title('Pulse');

subplot(3, 1, 2);
hGSR = plot(vect(:,2), 'g');
%axis([1 1000 0 2.5]);
hGSRTitle = title('GSR');
```

```
subplot(3, 1, 3);
hBreath = plot(vect(:,3), 'b');
hold on;
hbPeaks = scatter(vect(:,1), vect(:,2),'r');
hold off;
axis([1 1000 0 5]);
hBreathTitle = title('Breath');

set(hPulse,  'Erase', 'xor');
set(hpPeaks, 'Erase', 'xor');
set(hbPeaks, 'Erase', 'xor');
set(hGSR,    'Erase', 'xor');
set(hBreath, 'Erase', 'xor');



% ENTER INFINITE PLOTTING LOOP
i = 1;
j = 1;
k = 1;
time_100 = 1;
lbRate = 1;
lpRate = 1;
init_cnt = 1;
tempP = (1:2);
tempG = (1:2);
tempB = (1:2);

pfilt = [0.0039  0.0313  0.1094  0.2188  0.2734  0.2188  0.1094  0.0313  0.0039];          % approximate gaussian filter for pulse signal
bfilt = ones(1,10) .* 0.1;            % uniform filter for breath signal
while (1)
        % SIGNAL MCU THAT WE WANT VALUES
        fprintf(s,'s');
        % READ VALUES FROM MCU
        vect(i,:) = 5*fscanf(s,'%d')/255;

        % FILTER SIGNALS FOR SMOOTHING
        filtPulse  = conv(vect(:,1), pfilt);
        filtGSR    = conv(vect(:,2), pfilt);
        filtBreath = conv(vect(:,3), bfilt);

        % UPDATE SIGNAL PLOTS


        set(hPulse,  'YData', filtPulse);
        set(hGSR,    'YData', filtGSR(10:1000));
        set(hBreath, 'YData', filtBreath);
        drawnow;

        % COMPUTE BREATHING AND PULSE RATES & INSTANTANEOUS GSR
        if (init_cnt > WARMUP_TIME)

                % COMPUTE INSTANTANEOUS GSR
                Inst_GSR = 1/((5 - filtGSR(i))/(1000000*filtGSR(i)) - 1/(1000000));

                % CALCULATE AVERAGE BREATH RATE OVER ENTIRE PLOT
```

```
bPeaks = peakdet(filtBreath, BDELTA);
nbPeaks = size(bPeaks);
temp2 = bPeaks(2:nbPeaks);

bRate = mean(temp2' - bPeaks(1:nbPeaks-1,1));
bSTD = std(temp2' - bPeaks(1:nbPeaks-1,1));

lfiltBreath(k) = filtBreath(i);

% COMPUTE AND DISPLAY INSTANTANEOUS BREATH RATE
if(k > BREATH_WINDOW_SIZE)
    lbPeaks = peakdet(lfiltBreath, BDELTA);
    lnbPeaks = size(lbPeaks);
    ltemp2 = lbPeaks(2:lnbPeaks);
    if(size(ltemp2,1) ~= 0 && size(lbPeaks,1) ~= 0)
        lbRate = mean(ltemp2' - lbPeaks(1:lnbPeaks-1,1));
    end
    Inst_Breath = (60/lbRate)*(BREATH_WINDOW_SIZE/(time_100*(BREATH_WINDOW_SIZE/PULSE_WINDOW_SIZE)));        %scaling functi
    bTemp = (60/bRate)*(BREATH_WINDOW_SIZE/(time_100*(BREATH_WINDOW_SIZE/PULSE_WINDOW_SIZE)));        %scaling function

    bTitle = sprintf('Breath (Current: %4.2f [Breaths/Min], Mean: %4.2f, SD: %4.2f)', Inst_Breath, bTemp, bSTD);
    set(hBreathTitle, 'String', bTitle);

    set(hbPeaks, 'XData', bPeaks(:,1));
    set(hbPeaks, 'YData', bPeaks(:,2));

    k = 1;
end
k = k + 1;



% CALCULATE AVERAGE PULSE RATE
pPeaks = peakdet(filtPulse, PDELTA);                                        % (x,y) values of peaks in pulse signal
npPeaks = size(pPeaks);
temp = pPeaks(2:npPeaks);

pRate = mean(temp' - pPeaks(1:npPeaks-1,1));                               % average Pulse Rate
pSTD = std(temp' - pPeaks(1:npPeaks-1,1));                                  % STD of Pulse Rate

lfiltPulse(j) = filtPulse(i);

% COMPUTE AND DISPLAY PULSE RATE
if (j > PULSE_WINDOW_SIZE)
    lpPeaks = peakdet(lfiltPulse, PDELTA);                                  % (x,y) values of local peaks
    lnpPeaks = size(lpPeaks);                                               % number of local peaks
    ltemp = lpPeaks(2:lnpPeaks);
    if (size(ltemp,1) ~= 0 && size(lpPeaks,1) ~= 0)
        lpRate = mean(ltemp' - lpPeaks(1:lnpPeaks-1,1));
    end
    Inst_Pulse = (60/lpRate)*(PULSE_WINDOW_SIZE/time_100);                  % instantaneous pulse rate

    set(hpPeaks, 'XData', pPeaks(:,1));
    set(hpPeaks, 'YData', pPeaks(:,2));
```

```
            pTemp = (60/pRate)*(PULSE_WINDOW_SIZE/time_100);                    % scaling function
            pTemp2 = (60/pSTD)*(PULSE_WINDOW_SIZE/time_100);                    % scaling function

            %CALCULATES INSTANTANEOUS GSR
            gTemp = mean(filtGSR(10:1000));
            gTemp2 = std(filtGSR(10:1000));

            gAve = 1/((5 - gTemp)/(1000000*gTemp) - 1/(1000000));
            gSTD = 1/((5 - gTemp2)/(1000000*gTemp2) - 1/(1000000));

            %DISPLAYS READING INFO FOR ALL READINGS
            gTitle = sprintf('GSR (Current: %6.2f [Kilo Ohms], Mean: %4.2f, SD: %4.2f)', Inst_GSR/1000, gAve/1000, gSTD/1000);
            set(hGSRTitle, 'String', gTitle);

            pTitle = sprintf('Pulse (Current: %5.2f [Beats/Min], Mean: %4.2f, SD: %4.2f)', Inst_Pulse, pTemp, pSTD);
            set(hPulseTitle, 'String', pTitle);

            j = 1;
        end
        if(j == 1)
            tic;
        end
        if(j == PULSE_WINDOW_SIZE)
            time_100 = toc;
        end

        j = j + 1;
    end

    if (i == PLOT_SIZE + 1)
        i = 1;
    end

    init_cnt = init_cnt + 1;
    i = i + 1;
end

% CLOSE THE SERIAL CONNECTION (UNFORTUNATELY WE NEVER GET HERE)
fclose(s);
```

Audio

C

```
/*** INCLUDES ***/
#include <mega32.h>
#include <stdio.h>
#include <delay.h>
#include <math.h>
```

```
/*** DEFINES ***/
#define int2fix(a)   (((int)(a))<<8)
#define float2fix(a) ((int)((a)*256.0))
#define fix2float(a) ((float)(a)/256.0)
#define pi 3.14159
#define interval 100

/*** PROTOTYPES ***/
void initialize();

/*** GLOBALS ***/

/* DCT variables */
int  m1, m2, m3, m4 ;   /* 8 sample twiddle factors -- needed for 32 points */
int c1,c3,c5,c7,c9,c11,c13,c15; //16 sample factors -- needed for 32 points
int c17,c19,c21,c23,c25,c27,c29,c31; //32 sample factors
int a[8];  /* the 8 sample inputs -- needed for 32 points*/
int S[8];  /* the 8 sample outputs -- needed for 32 points */
int a16[16];  /* the 16 sample inputs  -- needed for 32 points*/
int S16[16];  /* the 16 sample outputs  -- needed for 32 points*/
int a32[32];  /* the 32 sample inputs */
int S32[32];  /* the 32 sample outputs */

char icount = 0;
char aproc = 0;  // flag to process the audio signal

int power = 0;       // power in the signal
int avg[32];         // average DCT coefficient
int n=0;
int c=0;

// enter every 8KHz
interrupt [TIM0_COMP] void getAD(void)
{
 // sample A/D and store
 a32[icount] = (int)ADCH;

 icount++;
 if (icount == 32)
 {
  icount = 0;
  aproc  = 1;
 }
 ADCSR.6 = 1;
}

//==Fast fixed multiply=================================
#pragma warn-
int multfix(int a, int b)
{
  #asm
 push r20
 push r21
 LDD  R22,Y+2  ;load a
 LDD  R23,Y+3

 LD   R20,Y    ;load b
 LDD  R21,Y+1

 muls r23, r21  ; [signed)ah * [signed)bh
 mov  r31, r0          ;
 mul  r22, r20  ; al * bl
 mov      r30, r1          ;
                         ;mov      r16, r0
 mulsu r23, r20  ; [signed)ah * bl
 add  r30, r0          ;
 adc  r31, r1          ;
 mulsu r21, r22  ; [signed)bh * al
 add  r30, r0          ;
 adc  r31, r1          ;

 pop r21
 pop r20
  #endasm
}
#pragma warn+


void DCT1D_02(void)
{
    int b0, b1, b2, b3, b4, b5, b6, b7;
    int c0, c1, c2, c3, c4, c5, c6;
    int d0, d1, d3, d4;
    int e2, e3, e4, e6, e7;
    int f2, f3, f4, f5, f6, f7;

    /* Step 1 */
    b0 = a[0] + a[7];
    b1 = a[1] + a[6];
    b2 = a[3] - a[4]; /* corrected */
    b3 = a[1] - a[6];
    b4 = a[2] + a[5];
    b5 = a[3] + a[4];
    b6 = a[2] - a[5];
    b7 = a[0] - a[7];  /***/

    /* Step 2 */
    c0 = b0 + b5;
    c1 = b1 - b4;
    c2 = b2 + b6;  /***/
    c3 = b1 + b4;
    c4 = b0 - b5;
    c5 = b3 + b7;  /***/
    c6 = b3 + b6;  /***/

    /* Step 3 */
    d0 = c0 + c3;  /***/
    d1 = c0 - c3;  /***/
    d3 = c1 + c4;
    d4 = c2 - c5;

    /* Step 4 */
    e2 = multfix(m3,c2);  /*c2*/
    e3 = multfix(m1,c6);  /*c6*/
    e4 = multfix(m4,c5);  /*c5*/
    e6 = multfix(m1,d3);
    e7 = multfix(m2,d4);

    /* Step 5 */
    f2 = c4 + e6;  /*c4*/
    f3 = c4 - e6;  /*c4*/
    f4 = e3 + b7;  /*b7*/
    f5 = b7 - e3;  /*b7*/
    f6 = e2 + e7;
    f7 = e4 + e7;

    /* Step 6 */
    S[0] = d0;        /*d0*/
    S[1] = f4 + f7;
    S[2] = f2;
    S[3] = f5 - f6;
    S[4] = d1;        /*d1*/
    S[5] = f5 + f6;
    S[6] = f3;
    S[7] = f4 - f7;
}

// From the algorithm to combine 2 8-point DCTs given in
// KR Rao and P Yip:
// "Discrete Cosine Transform", Academic Press 1990
// pages 60-61
void DCT1D_16(void)
{
    //segment 16 samples into first 8
    //then call DCT1D_02
    //and assign outputs to correct frequencies
    a[0] = a16[0] + a16[15];
    a[1] = a16[1] + a16[14];
    a[2] = a16[2] + a16[13];
    a[3] = a16[3] + a16[12];
    a[4] = a16[4] + a16[11];
    a[5] = a16[5] + a16[10];
    a[6] = a16[6] + a16[9];
    a[7] = a16[7] + a16[8];
    DCT1D_02();
    S16[0] = S[0];
    S16[2] = S[1];
    S16[4] = S[2];
    S16[6] = S[3];
```

```
        S16[8] = S[4];
        S16[10] = S[5];
        S16[12] = S[6];
        S16[14] = S[7];

        //segment 16 samples into second 8
        //premult by factors
        //then call DCT1D_02
        //and assign outputs to correct frequencies
        a[0] = multfix(c1, (a16[0] - a16[15]));
        a[1] = multfix(c3, (a16[1] - a16[14]));
        a[2] = multfix(c5, (a16[2] - a16[13]));
        a[3] = multfix(c7, (a16[3] - a16[12]));
        a[4] = multfix(c9, (a16[4] - a16[11]));
        a[5] = multfix(c11, (a16[5] - a16[10]));
        a[6] = multfix(c13, (a16[6] - a16[9]));
        a[7] = multfix(c15, (a16[7] - a16[8]));
        DCT1D_02();
        S16[1] = S[0] + S[1];
        S16[3] = S[1] + S[2];
        S16[5] = S[2] + S[3];
        S16[7] = S[3] + S[4];
        S16[9] = S[4] + S[5];
        S16[11] = S[5] + S[6];
        S16[13] = S[6] + S[7];
        S16[15] = S[7] ;
    }

    // From the algorithm to combine 2 16-point DCTs given in
    // KR Rao and P Yip:
    // "Discrete Cosine Transform", Academic Press 1990
    // pages 60-61
    void DCT1D_32(void)
    {
        //segment 16 samples into first 16
        //then call DCT1D_16
        //and assign outputs to correct frequencies
        a16[0] = a32[0] + a32[31];
        a16[1] = a32[1] + a32[30];
        a16[2] = a32[2] + a32[29];
        a16[3] = a32[3] + a32[28];
        a16[4] = a32[4] + a32[27];
        a16[5] = a32[5] + a32[26];
        a16[6] = a32[6] + a32[25];
        a16[7] = a32[7] + a32[24];
        a16[8] = a32[8] + a32[23];
        a16[9] = a32[9] + a32[22];
        a16[10] = a32[10] + a32[21];
        a16[11] = a32[11] + a32[20];
        a16[12] = a32[12] + a32[19];
        a16[13] = a32[13] + a32[18];
        a16[14] = a32[14] + a32[17];
        a16[15] = a32[15] + a32[16];
        DCT1D_16();
```

```
S32[0] = S16[0];
S32[2] = S16[1];
S32[4] = S16[2];
S32[6] = S16[3];
S32[8] = S16[4];
S32[10] = S16[5];
S32[12] = S16[6];
S32[14] = S16[7];
S32[16] = S16[8];
S32[18] = S16[9];
S32[20] = S16[10];
S32[22] = S16[11];
S32[24] = S16[12];
S32[26] = S16[13];
S32[28] = S16[14];
S32[30] = S16[15];

//segment 16 samples into second 16
//premult by factors
//then call DCT1D_16
//and assign outputs to correct frequencies
a16[0] = multfix(c1, (a32[0] - a32[31]));
a16[1] = multfix(c3, (a32[1] - a32[30]));
a16[2] = multfix(c5, (a32[2] - a32[29]));
a16[3] = multfix(c7, (a32[3] - a32[28]));
a16[4] = multfix(c9, (a32[4] - a32[27]));
a16[5] = multfix(c11, (a32[5] - a32[26]));
a16[6] = multfix(c13, (a32[6] - a32[25]));
a16[7] = multfix(c15, (a32[7] - a32[24]));
a16[8] = multfix(c17, (a32[8] - a32[23]));
a16[9] = multfix(c19, (a32[9] - a32[22]));
a16[10] = multfix(c21, (a32[10] - a32[21]));
a16[11] = multfix(c23, (a32[11] - a32[20]));
a16[12] = multfix(c25, (a32[12] - a32[19]));
a16[13] = multfix(c27, (a32[13] - a32[18]));
a16[14] = multfix(c29, (a32[14] - a32[17]));
a16[15] = multfix(c31, (a32[15] - a32[16]));
DCT1D_16();
S32[1] = S16[0] + S16[1];
S32[3] = S16[1] + S16[2];
S32[5] = S16[2] + S16[3];
S32[7] = S16[3] + S16[4];
S32[9] = S16[4] + S16[5];
S32[11] = S16[5] + S16[6];
S32[13] = S16[6] + S16[7];
S32[15] = S16[7] + S16[8];
S32[17] = S16[8] + S16[9];
S32[19] = S16[9] + S16[10];
S32[21] = S16[10] + S16[11];
S32[23] = S16[11] + S16[12];
S32[25] = S16[12] + S16[13];
S32[27] = S16[13] + S16[14];
S32[29] = S16[14] + S16[15];
```

```
    S32[31] = S16[15];
}

// send signals to Matlab for drawing
void transmit()
{
 int i;

 c++;
 if (c == interval)
 {
  for (i=1; i<32; i++) {
   printf("%d ", avg[i]);
  }
  c = 0;
 }
 else {
  // audio spectrum
  for (i=1; i<32; i++) {
   printf("%d ", S32[i]);
  }
  printf("%d ", power);
 }
 printf("\r"); // end packet
 PORTD.7 = ~PORTD.7;
}

// audio processing
void audioproc()
{
 int i=0;

 DCT1D_32();
 aproc = 0;

 // convert fixed back to char
 power = 0;
 for (i=1; i<32; i++) power += abs(S32[i]);

 // this is speech or something like it
 if (power > 10)
 {
     // update overall average signal
     for (i=0; i<32; i++)
     {
            avg[i] = 0.9*avg[i] + 0.1*S32[i];
        }
 }
}

// read A/D converter
void main(void)
{
 char inchar;
 initialize();
 while(1)
 {
     // perform DCT and other processing every 32 samples
  if (aproc) audioproc();

  // when signaled by Matlab, return current values
  if (UCSRA.7)
  {
   inchar = UDR;
   if (inchar=='s') {
    transmit();
   }
  }
 }
}

void initialize(void)
{
 int i=0;

 // comm indicator LED
 DDRD.7 = 1;
 PORTD.7 = 0;

 // serial setup for debugging using printf, etc.
 UCSRB = 0x18;
 UBRRL = 103;

 // set up timer0 to sample a/d at about 8kHz
 TCCR0 = 0b00000010;
 TIMSK = 0b00000010;
 OCR0 = 250;

 // set up a/d for external Vref, channel 0
 // channel zero / left adj / Aref is Vcc
 ADMUX = 0b01100000;

 // enable ADC and set prescaler to 1/128*16MHz=125kHz
 // and clear interupt enable
 // and start a conversion
 ADCSR = 0b11000110;

 //constants for 8 sample DCT [also used by 16 sample)
 m1 = float2fix(cos(4*pi/16));
 m2 = float2fix(cos(6*pi/16));
 m3 = float2fix(cos(2*pi/16) - cos(6*pi/16));
  m4 = float2fix(cos(2*pi/16) + cos(6*pi/16));
 // constants for 16 bit DCT
 c1 = float2fix(0.5*cos(1.0*pi/32.0));
 c3 = float2fix(0.5*cos(3.0*pi/32.0));
 c5 = float2fix(0.5*cos(5.0*pi/32.0));
 c7 = float2fix(0.5*cos(7.0*pi/32.0));
 c9 = float2fix(0.5*cos(9.0*pi/32.0));
 c11 = float2fix(0.5*cos(11.0*pi/32.0));
 c13 = float2fix(0.5*cos(13.0*pi/32.0));
 c15 = float2fix(0.5*cos(15.0*pi/32.0));
 c17 = float2fix(0.5*cos(17.0*pi/32.0));
 c19 = float2fix(0.5*cos(19.0*pi/32.0));
 c21= float2fix(0.5*cos(21.0*pi/32.0));
 c23= float2fix(0.5*cos(23.0*pi/32.0));
 c25= float2fix(0.5*cos(25.0*pi/32.0));
 c27 = float2fix(0.5*cos(27.0*pi/32.0));
 c29 = float2fix(0.5*cos(29.0*pi/32.0));
 c31 = float2fix(0.5*cos(31.0*pi/32.0));

 for (i=0; i<32; i++) avg[i] = 0;

    // and start the show
    #asm("sei")
}
```

MATLAB

```
% CLEAN UP LEFTOVER CONNECTIONS
clear all
try
    fclose(instrfind)
end

% OPEN A SERIAL CONNECTION
% set its rate to 9600 baud
% SR830 terminator character is  (ACSII 13)
% use fprintf(s,['m' mode]) to write
% and fscanf(s) to read the SR830
s = serial('COM6', 'baudrate', 9600, 'terminator', 13);
fopen(s)
disp('serial communication setup complete');

PLOT_SIZE = 300;
INTERVAL  = 100;    % must match interval in C code

vals  = zeros(32,1);
coeff = zeros(31,PLOT_SIZE);
power = zeros(PLOT_SIZE,1);

% SET UP PLOT AND ACQUIRE HANDLES
figure(1);

subplot(3,1,1);
hAudio = imshow(coeff(1:31,:));
colormap('copper');
set(hAudio, 'EraseMode', 'xor');
title('DCT Coefficient Heatmap');
```

```
subplot(3,1,2);
hPower = plot(power);
axis([1 PLOT_SIZE 0 40]);
set(hPower, 'Erase', 'xor');
title('Audio Power');

subplot(3,1,3);
hCoeff = stem(coeff(1:31,:));
set(hCoeff, 'Erase', 'xor');
title('Average Speech Distribution');

i = 1;
% COMMUNICATION AND PLOTTING LOOP
while (1)
  % signal MCU that we want a value
  fprintf(s,'s');
  % read value from MCU
  vals = fscanf(s,'%d');
  vals = abs(vals) ./ 128;
  coeff(:,i) = vals(1:31);

  % IF THIS IS AN AVERAGE DISTRIBUTION, PLOT IT IN THE APPROPRIATE PLACE
  if (size(vals,1) == 31)
    set(hCoeff, 'YData', vals);
    c = 0;
  % OTHERWISE, UPDATE THE HEATMAP AND POWER PLOT
  else
    coeff(:,i) = (coeff(:,i) .* (coeff(:,i) > mean(coeff(:,i)))) ./ max(coeff(:,i));
    set(hAudio, 'CData', coeff);
    power(i) = vals(32);
    set(hPower, 'YData', power);
  end
  drawnow;

  i = i+1;
  if (i == PLOT_SIZE + 1)
    i = 1;
  end
end

% CLOSE THE SERIAL CONNECTION (UNFORTUNATELY, WE NEVER GET HERE)
fclose(s);
```

Appendix B: Schematics

The hardware schematics are given in the Hardware Design section. The links to these images are given again below.

--- plethysmograph --- GSR --- breathing rate --- audio ---

Appendix C: Costs

This project turned out to be quite cheap. As this was not known a priori, we took care to sample expensive items and to use a previously owned microphone to reduce cost. A price list is given in the table below.

Part	Cost
custom PC board	$5.00
MAX233CPP for PC board	sampled
RS232 connector for PC board	$1.00
Mega32	$8.00
LM358 x3	$1.50
6in solder board	$2.50
2in solder board	$1.00
battery pack	$2.00
batteries	previously owned
microphone	previously owned
facemask	previously owned
circuit components	lab supplies
wood board	scrounged
screws, washers, spacers, etc.	lab supplies
TOTAL	$21.00

Appendix D: Tasks

Group Member Tasks

Jordan	Edwin
MATLAB audio analysis code	MATLAB PGB analysis code writing and testing
plethysmograph circuit layout, soldering, and testing	picture taking and editing
breathing rate circuit design, layout, soldering, and testing	breathing mask design and construction
GSR circuit layout and soldering	GSR circuit design and testing
audio circuit layout, soldering, and testing	audio circuit testing
C coding	power supply implementation and testing
prototype board construction and testing	
webpage design and report writing	report writing
polygraph wood board construction	polygraph wood board construction

References

Data Sheets

- LM358 Dual Op-Amp - http://www.national.com/pf/LM/LM358.html
- Infrared LED - http://rocky.digikey.com/WebLib/Lite-on/Web%20Data/E4208.pdf
- Infrared Phototransistor - http://rocky.digikey.com/WebLib/Lite-on/Web%20Data/LTR-4206E.pdf

Vendor Sites

- Maxim Semiconductor - http://www.maxim-ic.com/
- DigiKey - http://www.digikey.com/

Borrowed Designs

Thanks to Rishi Sinha (rks33) and William Chung (wc227) for the audio preamplifier circuit design, Dr G. J. Compton for the design of the plethysmograph, and Eli Billauer for the MATLAB program peakdet().

Acknowledgements

Thanks to Maxim Semiconductor for free samples of the MAX233CPP RS232 driver and Freescale Semiconductor for free prototype boards. Special thanks also to John Belina for general design advice