Matthew Hu
SUID: matthu
TA: Alexander Gonzalez

Heart-Rate Detector Report

1. Introduction

In this project, I will be using the DSP Shield in order to calculate a patient's heart rate. Electrocardiography (ECG) is the measurement of the electrical activity of the heart using electrodes. This data is often extremely important in helping medical doctors diagnose heart conditions. There is a lot of very important data that is stored in the ECG signal that can be extracted using various signal processing techniques. For this specific project, I will be focusing primarily on extracting the heart rate from the ECG signal.

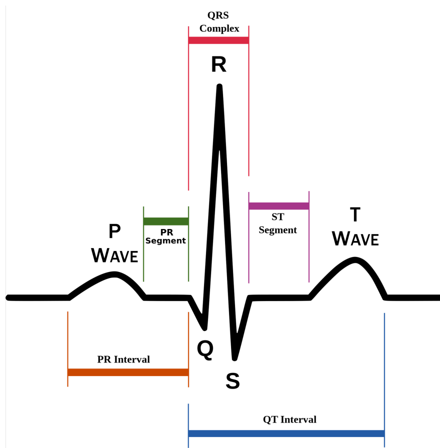

Figure 1: ECG Signal for 1 Period. Contains a P wave, QRS complex, and T Wave.

The ECG signal is usually periodic and can be separated into three primary sections: the P wave, the QRS complex, and the T wave. The P wave is the first positive deflection which corresponds to the depolarization of the atria. The P wave is then always followed by the QRS complex, or a series of three deflections. This is often the most visible section of the signal and contains a downwards Q deflection, followed by a sharp positive R deflection, and then another negative S wave deflection. The QRS complex corresponds to the depolarization of the heart's ventricles. Finally, the QRS wave is often followed by a much smaller T wave, which is another positive deflection corresponding to the repolarization of the ventricles.

In order to get a accurate estimation of the heart rate, we will be looking at the time interval between the R-peaks in the QRS complex. The goal of this project will then be to extract the locations of the R peaks and then calculate the average distance between the R peaks. The heart rate will then be computed by dividing the average sample distance between R peaks with the sampling frequency of the data.

Determining the location of the R peaks is slightly more difficult than what it initially seems. There are a variety of factors that get thrown in when analyzing different ECG signals. For example, there is a tendency for the baseline of the signal to drift, there is often a lot of noise introduced during the measurement process, and every patient produces a slightly different ECG signal, so this algorithm must take into account many different variables. In order to obtain a robust heart rate detector, I will be utilizing the wavelet transform which will be further discussed in future sections.

2. Timeline

For my original timeline, I proposed the following:

02/09 - 02/15: Research and setting up hardware to see if we can send the ECG signals from the electrodes to the DSP shield. Testing hardware to see if the DSP can accurately read and process the input signals.

02/15 - 02/29: Implementing the code in Matlab to get the test code up and running.

02/29 - 03/13: Implement code on the DSP shield based on the Matlab code. Test the results by comparing Matlab results with the DSP output.

And for the midway presentation, I planned to demonstrate the Matlab simulation to the T.A. The actual timeline ended up looking pretty similar:

02/09 - 02/16: Research and learning about the Wavelet Transform. After further discussion with Alex, I decided not to implement any additional hardware because of safety concerns and workload.

02/18 - 03/02: Implementing a basic simulation on Matlab in order to test the DSP Shield

03/02: Midway Demo: I demonstrated the working Matlab simulation to Alex.

03/02 - 03/12: Implement the code onto the DSP Shield for the demo.

03/12 - 03/19: Generating results and plots for the final report

Having the midway check as a hard deadline was very helpful in keeping me on track with my project.

3. Class Concepts Used

3.1 Wavelet Transform

For this project, I used the wavelet transform, which is very similar to the Short Time Fourier Transform (STFT) that we learned in class. Like the STFT, the wavelet transform provides a information in both time and frequency. When the signal is not stationary, we need a transform which provides information on the energy of different frequencies at different times. The STFT utilizes a sliding window in order to take the Fourier Transform of a small window. However, this window is constant in size, which can cause some problems in terms of resolution. At low frequencies, a window starts to lose frequency resolution because a window of size 1s may be appropriate for a 100 Hz signal, but it is not long enough to resolve a 0.1 Hz signal.

Therefore the wavelet transform is useful, because we can both dilate and translate the wavelet function. The wavelet transform is given by the equation:

$$W_a x(b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{+\infty} x(t)\, \psi\left(\frac{t-b}{a}\right) dt,\ a > 0.$$

As a, the scale factor increases, the wavelet widens and we get more information about lower frequency components of the signal. As the scale factor a, decreases, the wavelet shrinks, giving us better time resolution. We end up with good frequency resolution at low frequencies and good time resolution at high frequencies.

3.2 FIR Filtering

To take the wavelet transform of the ECG signals, I utilized the paper by Martinez on "A Wavelet-Based ECG Delineator: Evaluation on Standard Databases". This paper provided two FIR filters:

$$h[n] = \frac{1}{8} \cdot \{\delta[n+2] + 3\delta[n+1] + 3\delta[n] + \delta[n-1]\}$$
$$g[n] = 2 \cdot \{\delta[n+1] - \delta[n]\}\,.$$

Which where then utilized in the block diagram shown to the right.

Martinez's paper gave two possible implementations of the wavelet transform. The first way, shown as the first block diagram, involves a series of filtering and downsampling to achieve the wavelet transform for that specific scale. The second method, involves interpolating the filter impulse responses and removing the decimation stage. I decided to use the second algorithm because it was simpler to implement.

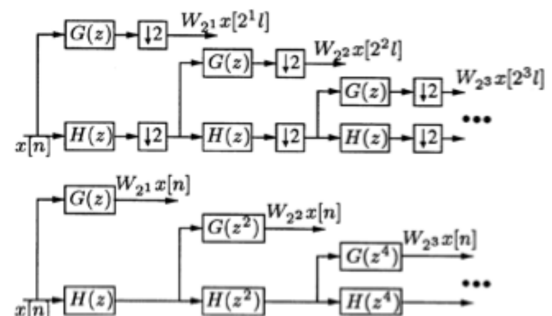

Figure 2: Block diagrams to implement the wavelet transform.

One additional issue was that the filters provided by the paper were not causal. As a result, I shifted the filters to make them causal and then realigned the output transforms afterwards.

3.3 Fixed Point Representation

Finally, to implement the system on the DSP Shield, I converted all the information to Q15 fixed point notation. All the ECG signals and the filter coefficients were converted to this format when being implemented on the processor. The coefficients for the g[n] filtered were scaled to less than 1 to maintain Q15 representation, and then output of the filter was then rescaled to balance

out the coefficients. The ECG signals were also scaled to fall between [-1, 1) so that they could also be represented as Q15 fixed point.

4. Implementation

4.1 Physionet Database of ECG Signals

The website **http://www.physionet.org/** contains a database of ECG signals that can be pulled using Matlab in order to analyze. These are real measurements taken from patients, so they will serve as an accurate way of testing my algorithm.

4.2 Matlab Simulation

Because the DSP Shield will be implemented using Q15 fixed point representation, I first had to scale the ECG signal to fall between the interval [-1, 1). Next, the filters detailed in the paper by Martinez were designed for an ECG signal sampled at 250 Hz. However, the ECG signals in the physionet database were sampled at 360 Hz. Therefore it is necessary to resample the signal to 250 Hz using Matlab's resample function. As a result, we will have an ECG signal at 250 Hz that can be represented in Q15 notation. For the Matlab simulation, we will ignore fixed point notation and compare the differences with the embedded processor afterwards.

To find the R-peaks of the signal, we look at four scales of the wavelet transform. Most of the energy of the QRS complex is located at the scales 2^k where k = 1…, 4. Therefore we will only look at these four scales when trying to find the R-peaks. Using the following equation, which was derived from the block diagram in Figure 2, I generated the filter coefficients needed to extract the first four scales of the wavelet transform.

$$Q_k\left(e^{j\omega}\right) = \begin{cases} G\left(e^{j\omega}\right), & k = 1 \\ G\left(e^{j2^{k-1}\omega}\right) \cdot \prod_{l=0}^{k-2} H\left(e^{j2^{l}\omega}\right) & k \geq 2 \end{cases}$$

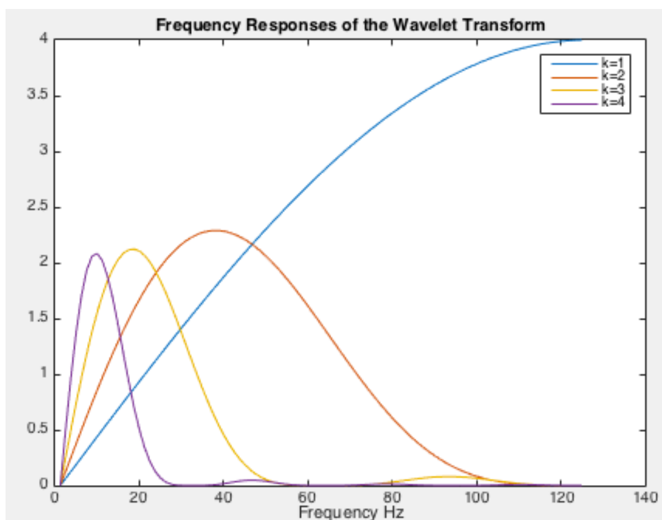

Figure 3: Frequency Responses of Wavelet Transforms

With the filter coefficients generated by this equation, I plotted the frequency response for each scale. As we can see from our graph on the above, the higher scale has a narrower frequency band and therefore often contains less noise. The output of these filters will be the wavelet transform at that particular scale k. The result is shown below:

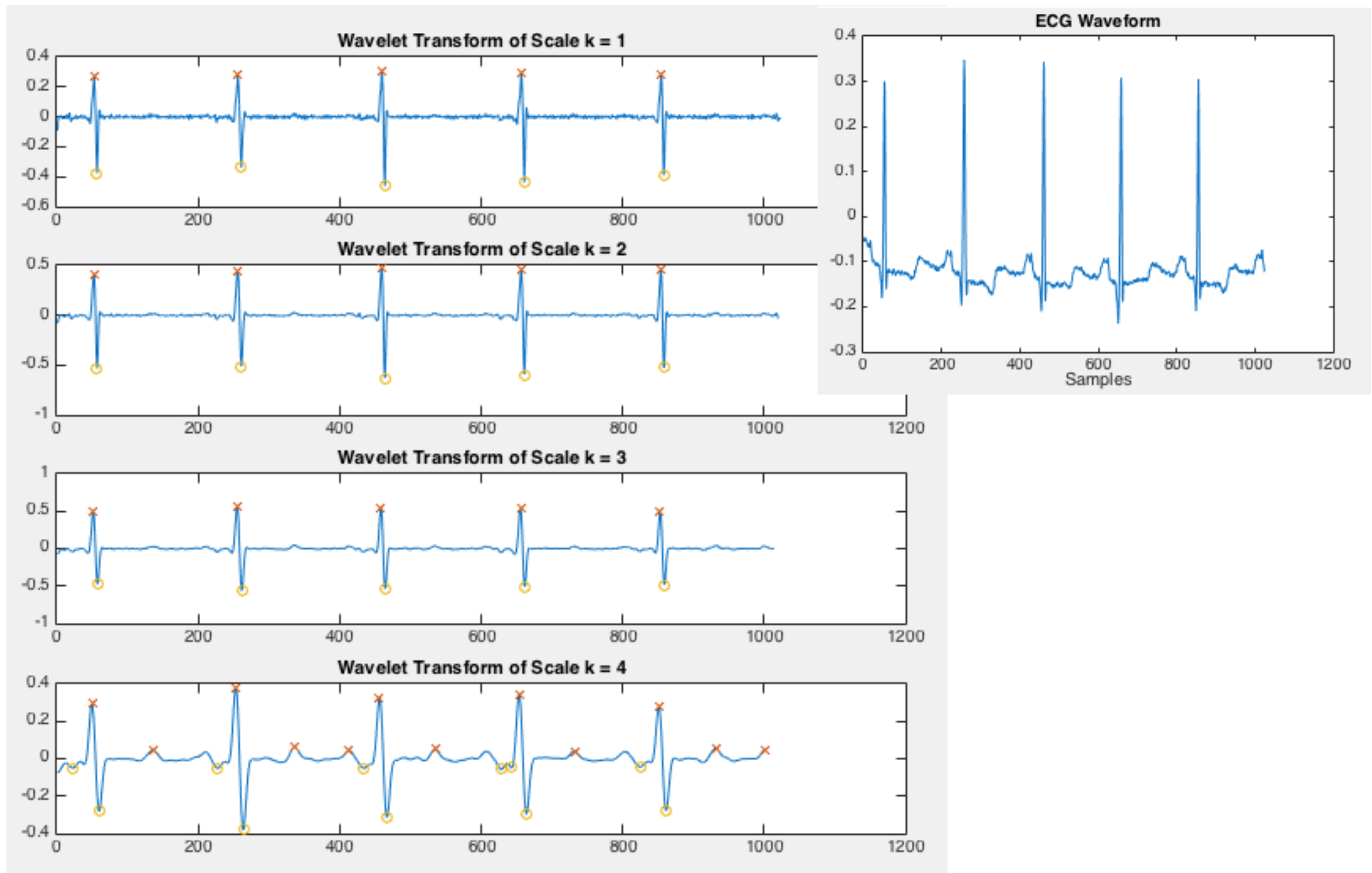

Figure 4: Plot of the wavelet transform of the ECG signal according to different scales k = 1…4. The positive peaks are labelled with an ‘x’ and the negative peaks are ‘o’

In the figures above, we can see the original ECG waveform and the corresponding wavelet transforms. We can see from the wavelet transform that for every QRS complex, there is a corresponding positive and negative peak pair in the wavelet transform at each scale. The location of the R-peak corresponds to the zero crossing of these peak pairs. Therefore in order to calculate heart rate, I will be searching for pairs of positive and negative peaks in all four scales, and then use that information in order to calculate the location of the R peak.

As we can see in the transforms above, with a higher scale, the signal contains less noise and tends to be smoother. We will therefore start our peak search with the highest scale $k = 4$. $k = 4$ will be the smoothest signal with the fewest peaks, and so the first step of the algorithm is to find

all positive in k = 4 where the signal is greater than 1/2 * the root mean squared of the signal. For negative peaks, we do the same but search for valleys where the signal is < -1/2*rms.

We next use the peaks and valleys in scale k = 4 to determine the peaks/valleys in k = 3. For every peak discovered in k = 4, we search for a nearby peak in k = 3. This ensures that the signal we are tracking in k = 4 also appears in k = 3. When choosing the new peaks, we weigh taller peaks more heavily if they are above a certain threshold. If not, then we simply choose whichever peak is closest to the peak found in k = 4.

We then iterate through this process so that the peaks found in k = 3 are used to find peaks in k = 2. Then those peaks are used to find peaks in k = 1. Eventually, we should be left with pairs of positive and negative peaks which have been present in all four scales. At k = 1, we then find the zero-crossing of each pair of positive and negative peaks in order to determine the location of the R-peaks in the original input signal. The difference between these peaks are then averaged and dividing by the sampling rate (250 Hz) in order to get the average heart rate.

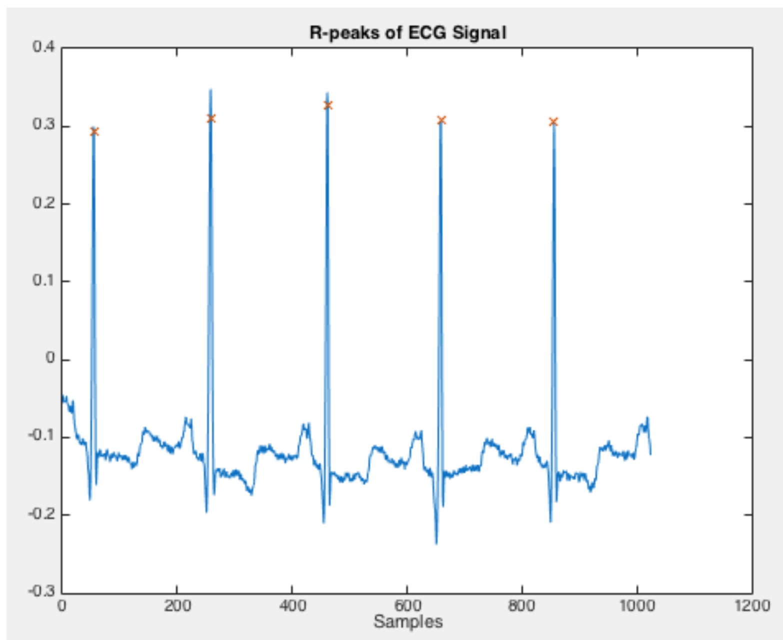

Figure 5: Plot of ECG Signal with R-peaks calculated. We can see that the algorithm accurately finds the R-peaks of each QRS complex

4.3 DSP Shield Implementation

The general implementation of the DSP shield is almost identical to that done in Matlab. The main differences are that the embedded processor utilizes Q15 fixed point notation, which can cause some issues with rounding. Also, the method of finding peaks is implemented differently on the processor because I used Matlab's peakfinder method for the Matlab implementation.

Initially, I stored the ECG signals as .wav files in the SD card. The general idea was to have the processor read blocks of the data from the SD card and then calculate the heart rate for that block in real-time. However, reading and processing data from the SD card ended up being a very slow process. As a result, it became more practical to store the signals in Matlab and send blocks of 1024 samples to the DSP shield after it was done processing the previous block. The DSP Shield then prints out the heart rate for that 1024 block on its display and sends the results to Matlab to record. In this way, we can view the heart rate over time and average the results to get the average heart rate.

5. Results

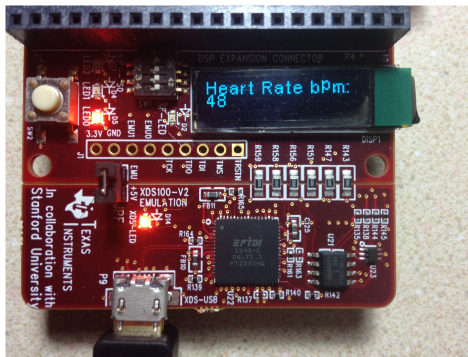

Figure 6: Picture of the DSP shield calculating heart rate in bpm

The DSP Shield accepts the ECG signal from Matlab in a Q15 fixed point format. It then processes the data it receives to print out the average heart rate of that 1024 sample block. It also returns the value back to Matlab to store. The units for the heart rate is bpm (number of beats per minute). Because the display cannot display decimals, the printed value is rounded to be as close as possible. However, the value sent to Matlab is more accurate.

To compare the DSP model with the Matlab model, I ran simulations for both models for 30 blocks of 1024 samples. That is a little over 2 minutes of data. The results of 2 different ECG signals: ECG 100 and ECG 103 are shown below:

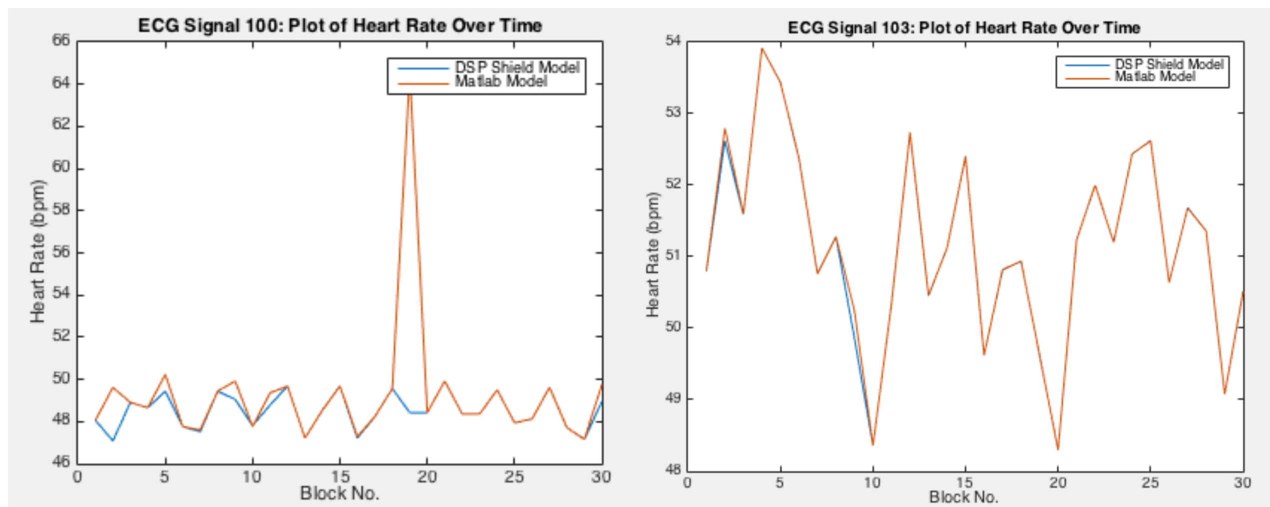

Figure 7: Plots of heart rate over time. Each block contains 1024 samples (or ~ 4s of data). The blue plot corresponds to the output of the DSP Shield. The orange plot corresponds to the output of the Matlab model.

As we can see from the graphs above, the DSP Shield model closely follows the Matlab model. As we can see from the outputs of ECG Signal 103, the amount of quantization noise introduced when using fixed point is minimal and barely affects the output of the model. For the ECG signal 100, we see that there is a spike in the heart rate for the Matlab model only. This difference between the DSP model and Matlab model is likely due to the difference in the peak finder algorithm. It seems like Matlab most likely missed a single ECG peak, and as a result, the heart rate shot up for a single block. However, the heart rate quickly returns back to a fairly consistent value afterwards in sync with the DSP model. The peak finder in the DSP shield implementation therefore seems to be more consistent than the one used in Matlab.

Both ECG signals 100 and 103 seem to have a fairly steady heart rate. The heart rate calculations vary slightly per block, but they overall seem to be fairly consistent around a mean value.
For ECG Signal 100: Average of Matlab Model = 49.2 bpm
Average of DSP model = 48.5 bpm

For ECG Signal 103: Average of Matlab Model = 51.1 bpm
Average of DSP model = 51.1 bpm

The averages between both models are very similar so the error between the two models is minimal. Next, I manually estimated the average heart rate by counting 30 peaks in the signal. I managed to get an ECG Signal 100 average of 48.7 bpm and a ECG Signal 103 average of 52.3. Therefore the averages computed by the algorithm are well within the expected range.

6. Future Work

I would like to keep the DSP shield in order to further work on this problem. There is a lot of additional work that can be done in this project. First of all, I can try to improve the speed of the peak finder function in order to get the process to work in real-time. Currently, it takes ~1.5s to process 4s of data, so it is still somewhat slow. One possible method of improving speed would be to downsample the ECG signal. This would require us to downsample the filter coefficients as well, but it would drastically reduce the number of iterations required to find the peaks. Ideally, I would be able to read all the ECG signals from the SD card and be able to output the heart rate while reading in new signals simultaneously.

There is other information you can extract from the QRS complex location. Heart rate is simply one piece of information we can extract using this technique. I would be interested in investigating what else we can learn from the ECG signal.

Also, there is the possibility of adding the actual hardware to the DSP shield. I would like to find some way of feeding the electrode inputs into the shield so that I actually measure people's heart-rate in real-time.

HeartRate2.cpp File Reference

```
#include "Audio_new.h"
#include "OLED.h"
#include "serial_array.h"
#include <dsplib.h>
#include <math.h>
```

Functions

SerialCmd	**cmd** (**maxDataLength**)
void	**setup** ()
void	**loop** () Main application loop. More...
void	**parse** (int c)
void	**processData** ()
int *	**peakFinder** (int *wv1, int *pout, int *nout, int length, int rms)
int *	**peakFinder2** (int *wv1, int *pout, int *nout, int *ppeakInd, int *npeakInd, int *peaks, int rms)
int	**isPeak** (int *wv, int index, int rms)
int	**isPeakn** (int *wv, int index, int rms)

Variables

const int	**WavBufferLength** = 1024
int	**q1** [4] = {0, 16384, -16384, 0}
int	**q2** [11] = {0, 0, 8192, 24576, 16384, -16384, -24576}
int	**q3** [26]
int	**q4** [57]
const long	**baudRate** = 115200
const int	**maxDataLength** = 2048
int *	**w1**
int *	**w2**
int *	**w3**
int *	**w4**
int	**w1Off**
int	**w2Off**

int	**w3Off**
int	**w4Off**
int *	**blockData**

Function Documentation

SerialCmd cmd (maxDataLength)

int isPeak (int * wv,
int index,
int rms
)

isPeak is a function that checks to see if the index corresponds to a relative peak in the signal wv is the wavelet transform signal index is the index of the sample you are checking outputs 1 if the index corresponds to a peak. Outputs 0 otherwise

int isPeakn (int * wv,
int index,
int rms
)

isPeakn is a function that checks to see if the index corresponds to a negative peak in the signal wv is the wavelet transform signal index is the index of the sample you are checking outputs 1 if the index corresponds to a negative peak. Outputs 0 otherwise

void loop ( )

Main application loop.

Receives commands from Matlab Processes commands from Matlab

```
void parse ( int c )
```

Parse Function Processes the command from Matlab If the Cmd 0: Accept the ECG Signal from Matlab Copy the input array to the blockData array Call ProcessData in order to calculate heart rate from blockData

Else print out unknown command

```
int* peakFinder ( int * wv1,
                  int * pout,
                  int * nout,
                  int   length,
                  int   rms
                )
```

PeakFinder Function Searches through wavelet transform k = 4 in order to find peaks wv1 is the wavelet transform signal pout is the resulting positive peaks nout is the resulting negative peaks length is the length of the signal , rms is the root mean square calculated earlier

```
int* peakFinder2 ( int * wv1,
                   int * pout,
                   int * nout,
                   int * ppeakInd,
                   int * npeakInd,
                   int * peaks,
                   int   rms
                 )
```

peakFinder2 Function This peakfinder is used for scales k = 3, 2, and 1 Uses peaks of the previous scale to find new peaks in current scale wv1 is the wavelet transform signal pout is the output positive peaks to be found nout is the output negative peaks to be found ppeakInd corresponds to the indices of the positive peaks of the previous scale npeakInd corresponds to the indices of the negative peaks of the previous scale peaks is a vector where peaks[0] is the number of positive peaks. peaks[1] is number of negative peaks rms is the rms of this wavelet transform

void processData ( )

ProcessData Function

Prior to this, the data we want analyze should be stored in blockData Will filter the data with the 4 wavelet transform filter coefficients stored above Wavelet transforms will be stored in w1...w4 Will find the peaks for all the wavelet transforms and use these peaks to calculate the location of the R-peaks. Once we have the R-peak locations, we calculate the heart rate

void setup ( )

Setup Function: Initialize LEDs, Initialize wavelet arrays Initialize the wavelet offset values Connect to Matlab

Variable Documentation

const long baudRate = 115200

int* blockData

const int maxDataLength = 2048

int q1[4] = {0, 16384, -16384, 0}

int q2[11] = {0, 0, 8192, 24576, 16384, -16384, -24576}

int q3[26]

Initial value:

```
= {0, 0, 0, 0, 1024, 3072, 6144, 10240, 11264, 9216, 4096, -4096, -9216,
       -11264,
    -10240, -6144, -3072, -1024, 0, 0, 0, 0, 0, 0, 0, 0}
```

int q4[57]

Initial value:

```
= {0, 0, 0, 0, 0, 0, 0, 0, 128,  384,  768, 1280, 1920, 2688, 3584, 4608,
       5248,
    5504, 5376, 4864, 3968, 2688, 1024, -1024, -2688, -3968, -4864, -5376,
       -5504, -5248, -4608,
    -3584, -2688, -1920, -1280, -768, -384, -128, 0, 0, 0, 0}
```

int* w1

int w1Off

int* w2

int w2Off

int* w3

int w3Off

int* w4

int w4Off

const int WavBufferLength = 1024

Project: Heart Rate Detector Author: Matthew Hu

Receives ECG Signals from Matlab through Cmd 0 Once it receives these signals, processess and analyzes them to find the average heart rate for a block of 1024 samples Displays heart rate on LED and sends heart rate * 2^6 back to Matlab

Allows Interface with Matlab Cmd 0: Matlab sends the ECG data block to the DSP

Here is a list of all file members with links to the files they belong to:

- baudRate : **HeartRate2.cpp**
- blockData : **HeartRate2.cpp**
- cmd() : **HeartRate2.cpp**
- isPeak() : **HeartRate2.cpp**
- isPeakn() : **HeartRate2.cpp**
- loop() : **HeartRate2.cpp**
- maxDataLength : **HeartRate2.cpp**
- parse() : **HeartRate2.cpp**
- peakFinder() : **HeartRate2.cpp**
- peakFinder2() : **HeartRate2.cpp**
- processData() : **HeartRate2.cpp**
- q1 : **HeartRate2.cpp**
- q2 : **HeartRate2.cpp**
- q3 : **HeartRate2.cpp**
- q4 : **HeartRate2.cpp**
- setup() : **HeartRate2.cpp**
- w1 : **HeartRate2.cpp**
- w1Off : **HeartRate2.cpp**
- w2 : **HeartRate2.cpp**
- w2Off : **HeartRate2.cpp**
- w3 : **HeartRate2.cpp**
- w3Off : **HeartRate2.cpp**
- w4 : **HeartRate2.cpp**
- w4Off : **HeartRate2.cpp**
- WavBufferLength : **HeartRate2.cpp**

Generated on Thu Mar 19 2015 22:59:38 for HeartRate2 by doxygen 1.8.9.1

References

Chun-Lin, Liu “A Tutorial of the Wavelet Transform”

Kohler, Hennig, and Orglmeister “The Principles of Software QRS Detection”

Li, Zheng, and Tai “Detection of ECG Characteristic Points Using Wavelet Transforms”

Martinez, Almeida, Olmos, Rocha, and Laguna “A Wavelet-Based ECG Delineator: Evaluation on Standard Databases”