Embedded Systems

02131

R-peak detection!

Bastian Buch, s113432
Jacob Gjerstruo, s113440

Abstract

The main task of this assignment was to develop the software for a Electro-Cardiogram (ECG) scanner using the Pan-Thompkins QRS-Detection Algorithm. By using this algorithm and writing a program in C we have created a program capable of determining a persons heartbeat from a set of data gathered from a ECG scanner. Our conclusions are that this is entirely feasible, and we recommend the implementation of this algorithm, though it is desirable to make the calculations using hardware, in order to optimize the running speed and make the calculations dynamic.

Indhold

1 Introduction

The company Medembed have hired us to investigate whether the Pan-Thompkins QRS detection algorithm might be used when the company is sending out their next product. This product is a wearable ElectroCardioGram scanner - hereafter just shortened to ECG scanner.
The first task we were given was to implement this algorithm in C, and hereafter, we were tasked with determining whether the algorithm is suited to be implemented in an embedded system.

The algorithm in itself consists of two parts - a filter-part, which consists of 5 different filters, and a detection-part, that consists of various functions for determining whether the input from the ECG-scanner is an R-peak. Once this is done, a program analysis will take place, in which we will analyse the time it takes for the program to run, as well as energy consumption and code size. Furthermore, during this program analysis, we will also discuss our choices, explaining why we have done as we have.

1.1 Requirements

For this assignment, we initially sat down and looked over the requirements, and in total, we found 5 functional requirements and one non-functional requirement. They are as follow:

Functional requirements for the application:

- Correct data acquisition in simulated real-time
- Implementation of the 5 filters
- Implementation of the RPeakDetection
- Correct output of relavant data to the user, based on the algorithm
- An analysis of our implementation, including an analysis of the critical parts, runtime and memory requirements.

Non-functional requirements for the application:

- The programming language used for this is C

2 Analysis

In designing the program for the wearable ECG scanner, there are a few things that needs to be considered:

1. How should the real-time data acquisition be simulated?
2. How should the filters be integrated?
3. How should the RPeakDetection be integrated?
4. Which data are relavant for the user, and how should it be shown?
5. How do you determine the critical parts?

2.1 Problem 1: Data acquisition in real-time

In the product specification for the ECG scanner, we were limited to working with a dataset and not a real patient. To ensure that the product is ready for implementation into a real scanner, one of the requirements is that we do not load the whole dataset into one array, and then work on this one. Instead, we were asked to make sure the data acquisition happens in real time, and this provides a few challenges, the biggest being how to ensure that the program runs through the entire dataset, no matter the size.

2.2 Problem 2: Integration of Filters

In the product specification, we were given 5 filters which all were to be implemented. 4 out of the 5 filters consists of a complicated formula that uses both the current data point, but also previous data points. This creates two issues:

1. How should the both the current and the previous data points be transferred from the program to the filters?
2. How to ensure there's no data under- and overflow?

To the first problem, there were two solutions: Either, one could use a struct or one could use two arrays, one with the data that we operate on and one with data that has already been calculated.
To the second problem, the easiest would be to implement the formula with strict if/else sentences, stating specificly when a certain subformula should be used.

2.3 Problem 3: Integration of RPeakDetection

After applying all filters to the data it remains to determine what constitutes a heartbeat and how to analyze the hearbeats of the patient. The data acquired from the filters contains peaks, which are analyzed and adapted on the fly to ensure that the patients heartbeats are correctly tracked. All peaks larger than a certain threshold are qualified as heartbeats - referred to as Rpeaks - and data is assembled based on this peak in order to classify further peaks as either heartbeats or noise with as great clarity as possible. This introduces the following issues:

1. How do we determine whether the patient suffers from irregular and/or weak heartbeasts?
2. How do we ensure that we pick up every heartbeat from the patient?

The problems are solved through the adaption of the criteria for what constitutes a heartbeat. By altering the threshold for Rpeaks depending on the estimated value of a heartbeat, it is possible to dynamically adapt peak detection to the patients heartbeat. If the threshold ends up exceedingly low, we can conclude that the patient has a weak heartbeat, and therefore is in need of medical examination. Furthermore, by establishing the time between each hearbeat, we can deduce the patients heartbeats per minute, which can also help determine if the patient needs examination or not. Finally, if the patients heartbeat strength (deduced from the peaks) differ wildly, the patient has an irregular heartbeat and should also see a doctor immediately.

2.4 Problem 4: Relavant data

To ensure that the patient are given early warnings of a pending heart problem, certain data must be shown to the patient. The requirements themself states that the program must show the value of the latest R-peak detected, the time-value at which it occured and the patients pulse. In addition to those, the requirements also states that the patient should be given a warning if the R-peak value is less than 2000 and if 5 successive RR-intervals has missed the RR-LOW and RR-HIGH values. How these data are shown, however, is not defined.

There are various ways to show these data. The first way to do so would be to just make a simple text-based screen where the information would be updated as we calculate them.
Alternatively, they could be plotted and a graph over the data could be shown.
These two could also be combined, giving both a graph that is updated in real time, as well as text-based information below the graph with the data specified in the requirements.
Furthermore, ideally the data and warnings should be displayed in a way easy for the user to understand.

2.5 Problem 5: Critical parts

When speaking of critical parts, there are two terms that needs to be discussed - memory requirements and time consumption. This is so because the more memory a program requires, the more physical memory the device must have and therefore, it becomes larger and more power-consuming (in this case, it is the latter that we are interested in). Likewise, as we were given a frequency rate of 250 reads per second, if our functions are very time consuming, we need a strong processor that will be more power-consuming. Therefore, the problem has the following challenges:

1. What is the estimation of memory usage, and can this be lowered?
2. How fast does the program run, and can it be improved?

3 Design

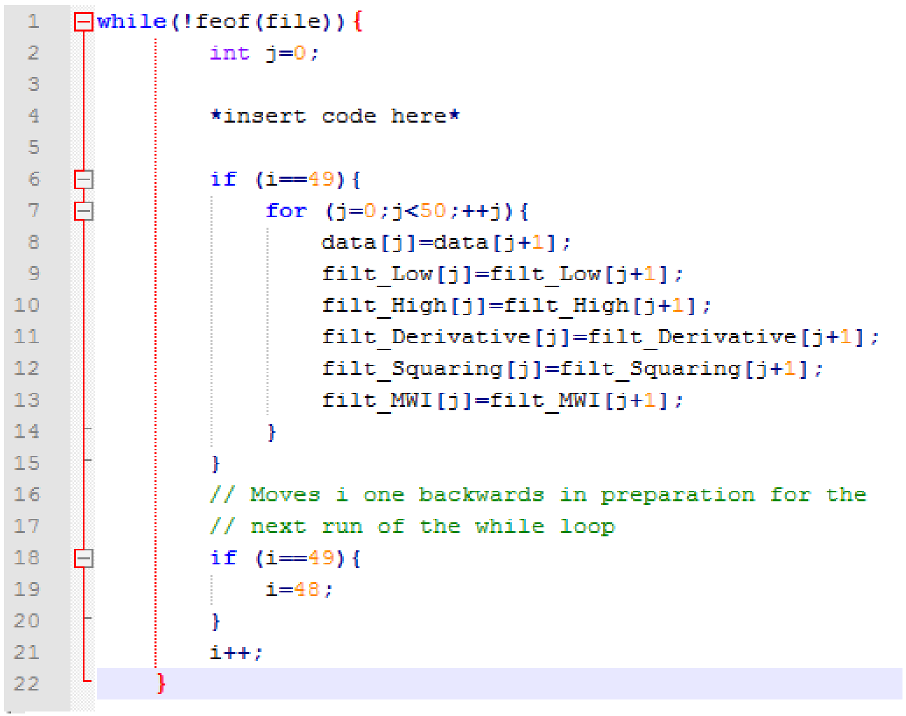
In designing our program, we very quickly decided to split the program into 4 smaller pieces. Each piece would then consist of functions native to the file - meaning that all filters were placed in the same file, all functions native to the RPeakDetection were placed in the same file, the function that scans the dataset were placed in the same file and they would all be run from a seperate file.

Futhermore, we decided that we would only hold up to 50 data points from the scanner at any given time - we decided that this was the optimal number, as if you went much lower, you risked not having enough data points for some of the filters, and having many more would just be a waste of memory. We also decided to go with 6 arrays with room for 50 data points - one to hold the ”raw”data, and five arrays to hold filtered data, one for each filter. This was a nescesity to ensure we had both the x- and the y-values needed in the formula for the filters.

4 Implementation

4.1 Solution one: Real-time data acquisition

As already described, the product specification stated that we needed to do the data acquisition in real time. To solve this, we looked back to the first introductionary exercises we did and used the while-loop we had created back then to run through the dataset. We then ensured that after each data-scan, the data was parsed to each filter and then to RPeakDetection, thus simulating that we received each data point one at a time, coinciding with each run through the filters with the while



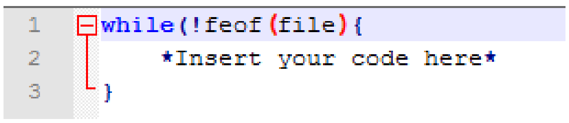
```
while(!feof(file)){
        int j=0;

        *insert code here*

        if (i==49){
            for (j=0;j<50;++j){
                data[j]=data[j+1];
                filt_Low[j]=filt_Low[j+1];
                filt_High[j]=filt_High[j+1];
                filt_Derivative[j]=filt_Derivative[j+1];
                filt_Squaring[j]=filt_Squaring[j+1];
                filt_MWI[j]=filt_MWI[j+1];
            }
        }
        // Moves i one backwards in preparation for the
        // next run of the while loop
        if (i==49){
            i=48;
        }
        i++;
    }
```

Figur 1: The code for the for-loop that moves all data points one step backwards, in preperation for the next run

loop. The most important part in this process is, of course, the while loop, that would look something like this:



```
while(!feof(file)){
        *Insert your code here*
}
```

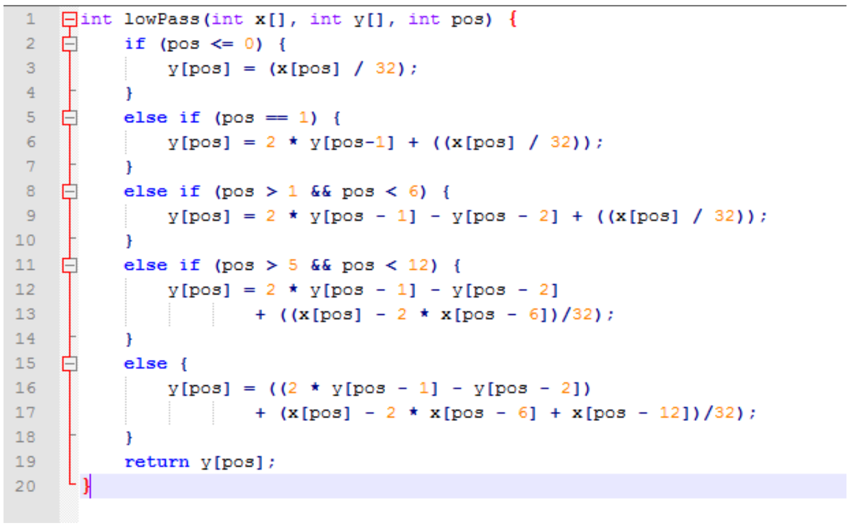
Figur 2: The while loop used for real-time data acquisition.

The condition in the while loop is simply that as long as fscanf returns true (that is, any other number than 0), it will continue to scan a new number. The fscanf will continue to return true as long as there's a new data point to be scanned, and thus, it will run to the end of the file. Within the body of this while loop would then be the function-call to each of the filters, as well as the function-call to the RPeakDetection.

4.2 Solution two: Integration of Filters

In regards to the first subproblem with the filters, we decided to solve the problems with the integration of the filters by using two arrays - one that holds the raw data that we needed to operate on, and one that holds the data that had already been

filtered. We then used these arrays as input parameters, as well as an integer that holds the position in the array that we are currently operating on.
In regards to the second, we decided to take the natural approach and implement each filter with if/else sentences, thus ensuring that we would not move beyond the lower bounds of the two input-arrays with data. Each formula were split up according to the various x_{n-z} where x is the array, n is the position we are operating on and z is an integer specified by the formula. Below is an example of this - it is our lowpass-filter, where each if-statement refers to a specific part of the formula.



```
int lowPass(int x[], int y[], int pos) {
    if (pos <= 0) {
        y[pos] = (x[pos] / 32);
    }
    else if (pos == 1) {
        y[pos] = 2 * y[pos-1] + ((x[pos] / 32));
    }
    else if (pos > 1 && pos < 6) {
        y[pos] = 2 * y[pos - 1] - y[pos - 2] + ((x[pos] / 32));
    }
    else if (pos > 5 && pos < 12) {
        y[pos] = 2 * y[pos - 1] - y[pos - 2]
                + ((x[pos] - 2 * x[pos - 6])/32);
    }
    else {
        y[pos] = ((2 * y[pos - 1] - y[pos - 2])
                + (x[pos] - 2 * x[pos - 6] + x[pos - 12])/32);
    }
    return y[pos];
}
```

Figur 3: An example of the implementation of one of the filters.

4.3 Solution three: Integration of RPeakDetection

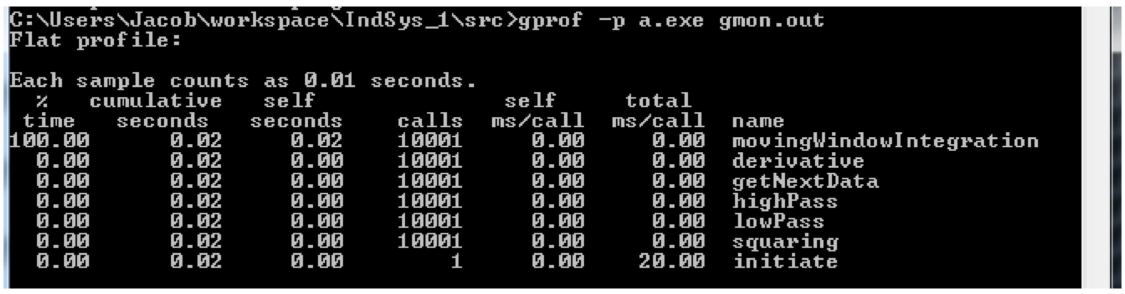
We implemented the detection by sorting our recieved data into an array consisting entirely of peaks. These peaks were then tested against our threshold for what constitied a peak. These values would change upon sucessfully finding a peak, dynamically altering the requirements for what constitutes a heartbeat. After confirming a heartbeat, the time between each heartbeat would be deduced through the "time" between each heartbeat in samples, knowing that for every 250 samples, a second has passed. Although the filters introduce a delay in the data treatment, this delay is applied to all heartbeats and therefore has no bearing on the heartbeats per minute, which can finally be deduced by scaling our results to 60 seconds. By creating and dynamically altering an average of the latest eight heartbeats, we could also determine when a heartbeat was "due" and thus determine if the heartbeats are irregular, if they appear too late or too early. Should no heartbeat appear, a search through all previous beats takes place to a reduced threshold, under the assumption that the heartbeat took place, but was simply too weak to be noticed by the initial threshold. The variables are dynamically altered in every case where a beat is found, thereby increasing the chance no further beats are missed.

4.4 Solution four: Relavant data

Due to time constraints, we decided to just go with a text-based screen that would show the required data and warnings in the initial version of the program. This was accomplished by a few simple print statements - however, in the finished product, we would implement the combined solution - giving the user both a graph (with appropriate markers for the thresholds) as well as a text based menu below with the exact data written out.

4.5 Solution five: Critical parts

To analyse the speed, we were given the gprof-tool, which is a profiler that checks the time spent on the entire program, as well as on each function. Figure 4 shows the results given by the profiler run on the main program, which consists of the reading of data from the dataset as well as the 5 filters. Originally, it should've held the function used for determining RPeaks, but due to time constraints, we did not manage to implement this.



```
C:\Users\Jacob\workspace\IndSys_1\src>gprof -p a.exe gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      0.02     0.02    10001     0.00     0.00  movingWindowIntegration
  0.00      0.02     0.00    10001     0.00     0.00  derivative
  0.00      0.02     0.00    10001     0.00     0.00  getNextData
  0.00      0.02     0.00    10001     0.00     0.00  highPass
  0.00      0.02     0.00    10001     0.00     0.00  lowPass
  0.00      0.02     0.00    10001     0.00     0.00  squaring
  0.00      0.02     0.00        1     0.00    20.00  initiate
```

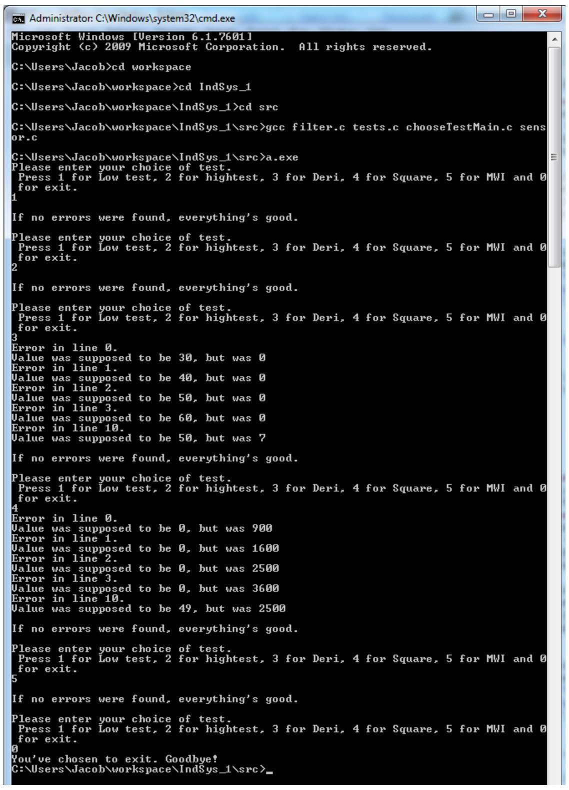
Figur 4: The table above shows the time spent in the various filters in our program

As can be seen, the 5 filters are each called 10001 times, which is the total number of datas in the dataset we operate on. What's also easily seen is that all the program in itself runs very fast - in fact, making these 50000 function calls takes only around 0.02 seconds, and of all those calls, the main time is spent in the movingWindowIntegration filter - in fact, all other filters takes less than 0.00 seconds, and thus are not even noteworthy of attention at this current time. It should be noted, though, that the current test were run on an intel core I7 with a high clock frequency, primarily used for high-end laptops, and as such, the functions will most likely take much longer time on the final device, as there is absolutely no reason to use such a high-end processor, considering that its power requirements are also quite high - at iddle, the processor uses around 30watt, and at load, it requires between 60 and 90 watt[2].

5 Results

5.1 Testresults of filters

Initially, we produced a simple test for the 5 filters. Each of these tests reads in 3000 data points from the a data set, starting out with the initial data given to us, and runs through the filters for these data. The tests then reads another 3000 data points into an array - these data points are found in the files given to us for checking, containing data that has already been run through the filters. We then check the data that we have found by running our filters, with the check-data and the results can be found below:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Jacob>cd workspace

C:\Users\Jacob\workspace>cd IndSys_1

C:\Users\Jacob\workspace\IndSys_1>cd src

C:\Users\Jacob\workspace\IndSys_1\src>gcc filter.c tests.c chooseTestMain.c sens
or.c

C:\Users\Jacob\workspace\IndSys_1\src>a.exe
Please enter your choice of test.
 Press 1 for Low test, 2 for hightest, 3 for Deri, 4 for Square, 5 for MWI and 0
 for exit.
1

If no errors were found, everything's good.

Please enter your choice of test.
 Press 1 for Low test, 2 for hightest, 3 for Deri, 4 for Square, 5 for MWI and 0
 for exit.
2

If no errors were found, everything's good.

Please enter your choice of test.
 Press 1 for Low test, 2 for hightest, 3 for Deri, 4 for Square, 5 for MWI and 0
 for exit.
3
Error in line 0.
Value was supposed to be 30, but was 0
Error in line 1.
Value was supposed to be 40, but was 0
Error in line 2.
Value was supposed to be 50, but was 0
Error in line 3.
Value was supposed to be 60, but was 0
Error in line 10.
Value was supposed to be 50, but was 7

If no errors were found, everything's good.

Please enter your choice of test.
 Press 1 for Low test, 2 for hightest, 3 for Deri, 4 for Square, 5 for MWI and 0
 for exit.
4
Error in line 0.
Value was supposed to be 0, but was 900
Error in line 1.
Value was supposed to be 0, but was 1600
Error in line 2.
Value was supposed to be 0, but was 2500
Error in line 3.
Value was supposed to be 0, but was 3600
Error in line 10.
Value was supposed to be 49, but was 2500

If no errors were found, everything's good.

Please enter your choice of test.
 Press 1 for Low test, 2 for hightest, 3 for Deri, 4 for Square, 5 for MWI and 0
 for exit.
5

If no errors were found, everything's good.

Please enter your choice of test.
 Press 1 for Low test, 2 for hightest, 3 for Deri, 4 for Square, 5 for MWI and 0
 for exit.
0
You've chosen to exit. Goodbye!
C:\Users\Jacob\workspace\IndSys_1\src>_
```

Figur 5: The results of our test cases of the 5 filters.

As can be seen, we also created a small menu for choosing how to navigate these tests, and the Low- and highpass filter, along with the MovingWindowIntegration filter runs without any problems. However, the derivative and the squaring filter reports an error in line 1 through 4, as well as in line 11 (it should here be noted that the program in itself shows line 0 through 3 + 10 - this is because the arrays that are checked start at the position of 0 rather than 1). We estimate that these errors are due to errors in the check-files, as we have manually computed the values for these specific data points and arrive at the same conclussion as our program does, and not as the check files (i.e. 0 for data point 1 through 4 and 7 for data point 11 in the derivative filter, and 900, 1600, 2500, 3600 and 2500 for the values of 1 through 4 and 11 for the squaring filter).

5.2 Testresults of RPeakDetection

In order to test our RPeakDetection, we tested a file of 3000 points of data. First, we used a function to determine all peaks and add them to a seperate array, then we ran RPeakDetection on that array.
Up untill a certain point, our results correlated with the expected results. After data point 2046, however, we recieved no further peaks. Through testing we have deduced that the error is related to SPKF and the way it interacts with our threshold for what constitutes a peak. Up untill this point, however, everything runs smoothly.

6 Discussion

Through testing of RPeakDetection we have determined our issue is in our Search-back function; after this function is called, our SPKF-value increases extremely, to a point wherein no further heartbeats will cross the threshold. Our results are still accurate, however.

7 Conclussion

We managed to create a program that fulfills the requirements for the filters – these runs and returns the correct value, which has been shown in the tests. We also created a program that, mostly, did as it was supposed to do in terms of finding R-peaks. This part were tested for the first 3000 data points, too, and did register the r-peaks for the first 2046 data points. However, after this point, something went inexplicedly wrong, and we did not have time to identify what this was. Up to this point, however, it calculates correctly and updates all the variables correctly – both arrays and integers. To be certain these data were saved correctly, we made various print statements during our implementation, both in the main body of the test, as well in the functions themselves, and these showed that the data were saved correctly.

Litteratur

[1] Michael Reibel Boesen, Linas Kaminskas, Paul Pop, Karsten Juul Frederiksen
Assignment 1: Software implementation of a personal ECG scanner
3rd Edition
2012.

[2] http://www.notebookcheck.net/typo3temp/pics/afd0f78c47.gif
Date of use: 09/10/2012

Appendix

A Who wrote what

Jacob Gjerstrup, s113440 wrote: Chapter 1, 2 (appart from 2.3), 3, 4 (appart from 4.3), 5.1, 7, appendix
Bastian Buch, s113432 wrote: Abstract, chapters: 2.3, 4.3, 5.2, 6.

B Sourcecode - introductionary exercises

B.1 ReadFromFile

Below is the sourcecode for the introductionary-exercise (From september the 5th) - more precisely, the ReadFromFile source-code.

```
/*
 * Created by Jacob Gjerstrup, s113440 and Bastian Buch, s113432
 */
#include <stdio.h>

int main (void){

        /*
         * Creates an array with the filename. Then, it initiates the
         * file to Null, and then opens the file. Also initiates
         * other variables used for the calculations
         */
        int i1=0, temp=0;
        static const char filename[] = "ECG.txt";
        FILE *file = NULL;
        file = fopen ( filename, "r" );

        /*
         * The if-sentence below checks if the file actually exists.
         * If it doesn't, a print is made.
         */
        if( file== NULL){
                printf("Couldn't open the file - name is incorrect");
                return 0;
        }
        /*
         * Scans the initial number, and then a while loop proceeds
         * to scan and check the rest of the numbers.
         */
        fscanf(file,"%i",&i1);
        while (fscanf(file,"%i",&temp)>0){
                fscanf(file,"%i",&temp);

                if (temp>i1){
                        i1=temp;
                }
        }

        /*
         * Closes the file, as we no longer need to read from it,
         * then prints the highest value and stops the program.
         */

        fclose(file);
        printf("The highest number is: %d",i1);
        return 0;
}
```

B.2 HelloWorld

The next is the sourcecode from the same exercise - this time, it's the sourcecode of our HelloWorld program.

```
#include <stdio.h>

/*
 * Created by Bastian Buch, s113432, and Jacob Gjerstrup, s113440
 */

int main (void){
  printf("Hello world!");
  return 0;
}
```

C Sourcecode - the real program

Below follows the sourcecode for each of the parts of our program, split into sections. The first part, the program, is where the various functions are called, and all our data is stored. The Filter.c contains the 5 different filters. The RPeakDetection contains the detection of each peak, along with the calculations of the various thresholds. The sensor is what scans data from the file, and thus simulates that we scan the patient, and finally, the header files is what contains all the prototypes for our functions.

C.1 Program

```
/*
 * Created by Jacob Gjerstrup, s113440
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "sensor.h"

        /*********************************************************
         * Initiate is what starts the entire program. It starts out
         * with opening our initial data file, then it proceeds to
         * create the arrays necessary for data treatment from raw
         * data to filtered data.
         *
         * After the treatment of each data point, it checks if the
         * loop has run for 49 times. If it has, it enters the while
         * loop, in which all data points are moved one step
         * backwards - thus ensuring that the last data point in the
         * arrays are a duplication of the second-to-last, and thus
         * free to be overwritten.
         *********************************************************/
void initiate(){
        clock_t start, end;
        double cpu_time_used;
        start = clock();

        static const char filename[] = "ECG.txt";
        FILE *file = NULL;
        int temp=0;
        file = fopen ( filename, "r" );
        int data[50]={0}, filt_Low[50]={0}, filt_High[50]={0},
                        filt_Derivative[50]={0}, filt_Squaring[50]={0},
                        filt_MWI[50]={0}, i=0, q=0;;

        while(!feof(file)){
                int j=0;
                data[i] = getNextData(file);

                filt_Low[i] = lowPass(data, filt_Low, i);
                filt_High[i] = highPass(filt_Low,
                                filt_High, i);
                filt_Derivative[i] = derivative(filt_High,
                                filt_Derivative, i);
                filt_Squaring[i] = squaring(
                                filt_Derivative[i]);
                filt_MWI[i] = movingWindowIntegration(
                                filt_Squaring, filt_MWI, i);

                if (i==49){
                        for (j=0;j<50;++j){
                                data[j]=data[j+1];
                                filt_Low[j]=filt_Low[j+1];
                                filt_High[j]=filt_High[j+1];
                                filt_Derivative[j]=filt_Derivative[j+1];
                                filt_Squaring[j]=filt_Squaring[j+1];
                                filt_MWI[j]=filt_MWI[j+1];
                        }
                }
                // Moves i one backwards in preparation for the
                // next run of the while loop
                if (i==49){
                        i=48;
                }
                i++;
        }

        //Closes the file, as we no longer need to read from it
        fclose(file);

        end = clock();
        cpu_time_used = ((double)(end-start)) / CLOCKS_PER_SEC;
        printf("Time spent on entire program: %f",
                        cpu_time_used);
}

int main (void){
        initiate();
        return 0;
}
```

C.2 Filters

```
/*
 * Created by Bastian Buch, s113432
 */
#include <stdlib.h>
#include <stdio.h>
#include "sensor.h"



/*
 * lowPass takes three input parameters, two of which are arrays
 * and one of which is an int. The two arrays contains the data
 * we need to calculate on (x), and the data that has already
 * been operated on previously (y). The integer is the position
 * in the array that we're currently working with.
 *
 * The data treatment itself is split into 5 if/else statements,
 * each splitting down the initial formula (seen in the final
 * else-statement) into the various compartments (depending on
 * the position in the arrays) to ensure that we do not exit the
 * boundaries of the arrays (the data before the first data
 * point is 0 and thus can be ignored)
 */
int lowPass(int x[], int y[], int pos) {
        if (pos <= 0) {
                y[pos] = (x[pos] / 32);
        }
        else if (pos == 1) {
                y[pos] = 2 * y[pos-1] + ((x[pos] / 32));
        }
        else if (pos > 1 && pos < 6) {
                y[pos] = 2 * y[pos - 1] - y[pos - 2] + ((x[pos] / 32));
        }
        else if (pos > 5 && pos < 12) {
                y[pos] = 2 * y[pos - 1] - y[pos - 2]
                                + ((x[pos] - 2 * x[pos - 6])/32);
        }
        else {
                y[pos] = ((2 * y[pos - 1] - y[pos - 2])
                                + (x[pos] - 2 * x[pos - 6] + x[pos - 12]
        }
        return y[pos];
}

/*
 * Like lowPass, highPass takes the same input parameters and
 * then treats the data much like lowPass, albeit out of a
 * different formula, where the full formula can be seen in the
 * final else-statement.
 */
int highPass(int x[], int y[], int pos) {
        if (pos <= 0) {
                y[pos] = ((x[pos]) / 32);
        }
        else if (pos >= 1 && pos < 16) {
                y[pos] = (y[pos - 1] - ((x[pos]) / 32));
        }
        else if (pos == 16) {
                y[pos] = (y[pos - 1] - (x[pos]) / 32 + x[pos - 16]);
        }
        else if (pos >= 17 && pos < 32) {
                y[pos] = (y[pos - 1] - (x[pos]) / 32 + x[pos - 16] -
                                x[pos - 17]);
        }
        else {
                y[pos] = (y[pos - 1] - ((x[pos]) / 32) + x[pos - 16] -
                                x[pos - 17] + (x[pos - 32]) / 32);
        }
        return y[pos];
}

/*
 * movingWindowIntegration takes the same input parameters as
 * high- and lowpass.
 */
int movingWindowIntegration(int x[], int y[], int pos){
        int i=0, k=0;
        for (i=30;i>=1;i--){
                y[pos]+=((x[pos-(30-i)])/30);
        }

        return y[pos];
}
```

```
/*
 * Like high- and lowPass, derivative takes the same 3 input
 * parameters. Also like high- and lowPass, it is composed of
 * various if/else statements, each set up to ensure we do not
 * go beyond the boundaries of the arrays.
 *
 * The full formula can be seen in the last else-statement.
 */
int derivative(int x[], int y[], int pos) {
        if (pos==0) {
                y[pos]=(2*x[pos])/8; // should be 30
        }
        else if (pos==1 || pos == 2) {
                y[pos]=(2*x[pos]+x[pos-1])/8; // should be 40 for pos==1
        }
        else if (pos == 3) {
                y[pos]=(2*x[pos]+x[pos-1]-x[pos-3])/8;
        }
        else {
                y[pos]=(2*(x[pos])+x[pos-1]-x[pos-3]-(2*x[pos-4]))/8;
        }
        return y[pos];
}

/*
 * Squaring takes an input parameter and returns the squared
 * value.
 */
int squaring(int value) {
        return ((value) * (value));
}
```

C.3 RPeakDetection

```
/*
 * RPeakDetection.c
 *
 *  Created on: 26/09/2012
 *      Author: Bastian
 */

#include <stdio.h>
#include <stdlib.h>
#include "sensor.h"
```

```
void findPeaks(int (*peaks)[1500][2], int data[]) {

        int peakPos = 0, i=0;

        for (i = 0; i < 2999; i++) {
                if (data[i-1] > data[(i - 2)] && data[i-1] > data[i]) {
                        (*peaks)[peakPos][0] = data[i-1];
                        (*peaks)[peakPos][1] = i-1;
                        if(data[i-1] > 4000) printf("%d at %d with pos %
                                        , data[i-1], peakPos, i);
                        peakPos++;
                }
        }
}


int calculateRRAverage(int (*RRArray)[8], int *pos) {
        if (*pos>8){
                return (((*RRArray)[0] + (*RRArray)[1] + (*RRArray)[2]
                       + (*RRArray)[3] + (*RRArray)[4] + (*RRArray)[5]
                       + (*RRArray)[6] + (*RRArray)[7]) / 8);
        }
        else{
                return (((*RRArray)[0] + (*RRArray)[1] + (*RRArray)[2] +
                                (*RRArray)[3] + (*RRArray)[4] + (*RRArra
                               + (*RRArray)[6] + (*RRArray)[7]) / *pos)
        }



}

void Searchback(int *spkf, int *npkf, int *threshold1,
                int *threshold2, int RR, int *RR_low, int *RR_high,
                int *RR_miss, int *counter1, int (*RRArray1)[8],
                int *RRAverage1, int (*peaks)[1500][2], int *lastPeak,
                int *totalPeakCounter) {
        int i = 0;
        for (i = *totalPeakCounter; i >= 0; i--) {
                if ((*peaks)[i][0] > *threshold2) {
                        (*RRArray1)[(*counter1 % 8)] = RR;
                        *counter1 = *counter1 + 1;
                        *RRAverage1 = calculateRRAverage(RRArray1,counte
                        *lastPeak = (*peaks)[i][0];
                        *RR_low = (0.92 * *RRAverage1);
                        *RR_high = (1.16 * *RRAverage1);
                        *RR_miss = (1.66 * *RRAverage1);
                        *spkf = ((*peaks)[i][0]*0.25) + (*spkf * 0.75);
                        *threshold1 = *npkf + ((*spkf - *npkf) / 4);
                        *threshold2 = (*threshold1 / 2);
                        printf("Peak = %d / Peaktime = %d / RRlow = %d /
                                        "RRhigh = %d / RR = %d\n",(*peak
                                        (*peaks)[(i)][1]+1, *RR_low, *RF
                        break;
                        }
                }
        }

void calculateRR(int *spkf, int *npkf, int *threshold1,
                int *threshold2, int *RR_low, int *RR_high,
                int *RR_miss, int *counter1, int *counter2,
                int (*RRArray1)[8], int (*RRArray2)[8], int *RRAverage1,
                int *RRAverage2, int (*peaks)[1500][2], int *lastPeak,
                int *lastRPeak, int *RRMissCounter,int *totalRR,
                int *irregularHeartbeat, int *totalPeakCounter) {
        (*peaks)[20][0] = 5000;
        int i = 0;
        for (i = 0; i < 1500; i++) {
                if((*peaks)[i][0] > 4000) printf("Peak at %d with "
                        "value %d to SPKF %d\n",i,(*peaks)[i][0]), *spkf
                if (*RRMissCounter >= 5) {
                        (*irregularHeartbeat) = 1;
                }
                *totalPeakCounter = *totalPeakCounter + 1;
                // peak = treshold is not specified in assignment;
                // assuming not Rpeak due to danger of false negative
                if ((*peaks)[i][0] > *threshold1) {
                        // to convert RR to seconds divide with 250, as
                        //this is the sample rate per second
                        int RR = (((*peaks)[i][1]-*lastPeak)*1000/250);
                        //1000 to convert 1s to ms and 250 because that'
                        //how many calculations per minute we do.
                        if (RR > *RR_low && RR < *RR_high) {
                                (*RRArray1)[*counter1 % 8] = RR;
                                *counter1 = *counter1 + 1;
                                *RRAverage1 = calculateRRAverage(RRArray
                                                counter1);
                                // since RR misses have to be consecutive
                                //the counter is reset
```

```
                                        *RRMissCounter = 0;
                                        (*RRArray2)[((*counter2) % 8)] = RR;
                                        *counter2 = *counter2 + 1;
                                        *RRAverage2 = calculateRRAverage(RRArray
                                                        counter2);
                                        *lastRPeak = (*peaks)[i][0];
                                        *RR_low = (0.92 * *RRAverage2);
                                        *RR_high = (1.16 * *RRAverage2);
                                        *RR_miss = (1.66 * *RRAverage2);
                                        *spkf = (((*peaks)[i][0] * 0.125) +
                                                        (spkf * 0.875));
                                        *threshold1 = (*npkf + ((*spkf - *npkf)
                                        *threshold2 = ((*threshold1) / 2);
                                        *totalRR = *totalRR + RR;
                                        printf("Peak = %d / Peaktime = %d / RRlo
                                                        "RRhigh = %d / RR = %d /
                                                        "%d\n",(*peaks)[i][0],(*
                                                        *RR_low, *RR_high, RR, *
                                        *lastPeak = (*peaks)[i][1];
                                } else if (RR <= *RR_miss) {
                                        *RRMissCounter = *RRMissCounter + 1;
                                        printf("SPKF = %d\n", *spkf);
                                } else if (RR > *RR_miss) {
                                        *RRMissCounter = *RRMissCounter + 1;
                                        Searchback(spkf, npkf, threshold1, thres
                                                        RR, RR_low, RR_high, RR_
                                                        RRArray1, RRAverage1, pe
                                                        totalPeakCounter);
                                }
                        } else {
                                *npkf = (((*peaks)[i][0] + 7 * *npkf) / 8);
                                *threshold1 = *npkf + ((*spkf - *npkf) / 4);
                                *threshold2 = ((*threshold1) / 2);
                        }
                }
        }
```

C.4 Sensor

```
/*
 * Created by Jacob Gjerstrup, s113440 and Bastian Buch, s113432
 */
#include <stdio.h>
#include <stdlib.h>
#include "sensor.h"
```

```
int getNextData(FILE *file){
        int i1=0;
        //The if-sentence below checks if the file actually exists.
        //If it doesn't, a print statement is sent.
        if( file== NULL){
                printf("couldnt open file");
                return 0;
        }

        // Scans the next number in the file, then returns it.
        fscanf(file,"%i",&i1);
        return i1;
}
```

C.5 Header files

C.5.1 sensor.h

Below is the first of our headerfiles, called sensor.h. This file contains only the prototype for our sensor.

```
#ifndef ADD_H_GUARD
#define ADD_H_GUARD
int getNextData();
#endif
```

C.5.2 filter.h

After this one, the next header file called filter.h comes - this file contains the prototypes of our filters.

```
#ifndef ADD_H_GUARD
#define ADD_H_GUARD
int lowPass(int x[], int y[], int pos);
int highPass (int x[], int y[], int pos);
int movingWindowIntegration(int x[], int y[], int pos);
int derivative(int x[], int y[], int pos);
int squaring(int value);
#endif
```

C.5.3 RPeakDetection.h

As the last headerfile, we've RPeakDetection.h that contains the prototypes for the functions nescesary for finding an RPeak.

```
/*
 * RPeakDetection.h
 *
 *  Created on: 02/10/2012
 *      Author: Bastian
 */

#ifndef RPEAKDETECTION_H_
#define RPEAKDETECTION_H_

void findPeaks(int (*peaks)[1500][2], int data[]);

/*
int calculateRRAverage(int RRArray[]);
int calculateRRAverage(int RRArray[]);
void Searchback(int *spkf, int *threshold1, int *threshold2, int RR,
                int *RR_low, int *RR_high, int *RR_miss, int *counter1,
                int *RRAverage1, int (*peaks)[][], int *lastPeak);
void calculateRR(int *spkf, int *npkf, int *threshold1, int *threshold2,
                int *RR_low, int *RR_high, int *RR_miss, int *counter1,
                int (*RRArray1)[], int (*RRArray2)[], int *RRAverage1, i
                int (*peaks)[][], int *lastPeak, int lastRPeak, int *RR
                int *totalRR, int *irregularHeartbeat);
*/
#endif /* RPEAKDETECTION_H_ */
```

C.6 Tests

We decided to do a run of tests, as discussed in the report. Below is the sourcecode for the tests:

C.7 Tests of RPeakDetection

```
/*
 * RPeakTest.c
 *
 *  Created on: 03/10/2012
 *      Author: Bastian
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include "sensor.h"
#include "RPeakDetection.h"


void findPeakTest(int (*peaks)[1500][2]) {

        static const char filename[] = "x_mwi.txt";
        FILE *file = NULL;
        file = fopen(filename, "r");
        int data[3000] = { 0 }, i = 0;

        for (i = 0; i < 3000; i++) {
                data[i] = getNextData(file);
        }

        findPeaks(peaks, data);
}

int main (void){
        int peaks[1500][2] = {0,0};
        int (*peaks_pointer)[1500][2] = &peaks;
        int spkf = 0;
        int npkf = 0;
        int threshold1 = 0;
        int threshold2 = 0;
        int RR_low = 0;
        int RR_high = 999999;
        int RR_miss = 0;
        int counter1 = 0;
        int counter2 = 0;
        int RRArray1[8] = { 0 };
        int RRArray2[8] = { 0 };
        int RRAverage1 = 0;
        int RRAverage2 = 0;
        int lastPeak = 0;
        int lastRPeak = 0;
        int RRMissCounter = 0;
        int totalRR = 0;
        int irregularHeartbeat = 0;
        int *spkf_pointer = &spkf;
        int *npkf_pointer = &npkf;
        int *threshold1_pointer = &threshold1;
        int *threshold2_pointer = &threshold2;
        int *RR_low_pointer = &RR_low;
        int *RR_high_pointer = &RR_high;
        int *RR_miss_pointer = &RR_miss;
        int *counter1_pointer = &counter1;
        int *counter2_pointer = &counter2;
        int (*RRArray1_pointer)[8] = &RRArray1;
        int (*RRArray2_pointer)[8] = &RRArray2;
        int *RRAverage1_pointer = &RRAverage1;
        int *RRAverage2_pointer = &RRAverage2;
        int *lastPeak_pointer = &lastPeak;
        int *lastRPeak_pointer = &lastRPeak;
        int *RRMissCounter_pointer = &RRMissCounter;
        int *totalRR_pointer = &totalRR;
        int *irregularHeartbeat_pointer = &irregularHeartbeat;
        int totalPeakCounter = 0;
        int *totalPeakCounter_pointer = &totalPeakCounter;

        int input=1;
        findPeakTest(peaks_pointer);
        calculateRR( spkf_pointer, npkf_pointer, threshold1_pointer, thr
                        counter1_pointer, counter2_pointer, RRArray1_poi
                        lastPeak_pointer, lastRPeak_pointer, RRMissCoun
        printf("rraverage=%d",*RRAverage1_pointer);
        /*
        printf("Latest R-peak: %d. Latest R-peak at: %d seconds.\nHeartb
        if (lastRPeak <= 2000) printf("Weak heartbeat./n");
        if (RRMissCounter >= 5) printf("Irregular heartbeat./n");
        if (RRMissCounter < 5 && lastRPeak > 2000) printf("No warnings."
        printf("/n/nPress 0 to exit.");
        scanf("%d", &input);*/

        return 0;
}
```

C.7.1 tests

```
#include <stdio.h>
#include <stdlib.h>
#include "sensor.h"

void testLow(){
        static const char filename[] = "ECG.txt";
        FILE *file = NULL;
        file = fopen ( filename, "r" );
        static const char filename2[] = "x_low.txt";
        FILE *file2 = NULL;
        file2 = fopen ( filename2, "r" );
        int Data[3000]={0}, FilteredData[3000]={0},
                        checkData[3000]={0}, i=0;

        for (i=0;i<3000;i++){
                Data[i] = getNextData(file);
                checkData[i]=getNextData(file2);
                FilteredData[i] = lowPass(Data, FilteredData, i);
        }

        for (i=0;i<3000;i++){
                if (checkData[i]!=FilteredData[i]){
                        printf("Error in line %d.\n"
                                "Value was supposed to be %d, but was %d
                                "\n", 1+i, checkData[i], FilteredData[i]
                }
        }
        printf("\nIf no errors were found, everything's good.\n\n");

        /*
         * Closes the file, as we no longer need to read from it,
         * then prints the highest value and stops the program.
         */
        fclose(file);

}

void testHigh(){
        static const char filename[] = "x_low.txt";
        FILE *file = NULL;
        file = fopen ( filename, "r" );
        static const char filename2[] = "x_high.txt";
        FILE *file2 = NULL;
        file2 = fopen ( filename2, "r" );
        int Data[3000]={0}, FilteredData[3000]={0},
                        checkData[3000]={0}, i=0;

        for (i=0;i<100;i++){
                Data[i] = getNextData(file);
                checkData[i]=getNextData(file2);
                FilteredData[i] = highPass(Data, FilteredData, i);
        }

        for (i=0;i<100;i++){
                if (checkData[i]!=FilteredData[i]){
                        printf("Error in line %d.\n"
                                "Value was supposed to be %d, but was %d
                                "\n", i, checkData[i], FilteredData[i]);
                }
        }
        printf("\nIf no errors were found, everything's good.\n\n");
        /*
         * Closes the file, as we no longer need to read from it,
         * then prints the highest value and stops the program.
         */
        fclose(file);
}

void testDerivative(){
        static const char filename[] = "x_high.txt";
        FILE *file = NULL;
        file = fopen ( filename, "r" );
        static const char filename2[] = "x_der.txt";
        FILE *file2 = NULL;
        file2 = fopen ( filename2, "r" );
        int Data[3000]={0}, FilteredData[3000]={0},
                        checkData[3000]={0}, i=0;

        for (i=0;i<3000;i++){
                Data[i] = getNextData(file);
                checkData[i]=getNextData(file2);
                FilteredData[i] = derivative(Data, FilteredData, i);
        }

        for (i=0;i<3000;i++){
                if (checkData[i]!=FilteredData[i]){
                        printf("Error in line %d.\n"
                                "Value was supposed to be %d, but was %d
                                "\n", i, checkData[i], FilteredData[i]);
                }
        }
        printf("\nIf no errors were found, everything's good.\n\n");
        /*
         * Closes the file, as we no longer need to read from it,
         * then prints the highest value and stops the program.
         */
        fclose(file);
}

void testSquaring(){
        static const char filename[] = "x_der.txt";
        FILE *file = NULL;
        file = fopen ( filename, "r" );
        static const char filename2[] = "x_sqr.txt";
        FILE *file2 = NULL;
        file2 = fopen ( filename2, "r" );
        int Data[3000]={0}, FilteredData[3000]={0},
                        checkData[3000]={0}, i=0;

        for (i=0;i<3000;i++){
                Data[i] = getNextData(file);
                checkData[i]=getNextData(file2);
                FilteredData[i] = squaring(Data[i]);
        }

        for (i=0;i<3000;i++){
                if (checkData[i]!=FilteredData[i]){
                        printf("Error in line %d.\n"
                                "Value was supposed to be %d, but was %d
                                "\n", i, checkData[i], FilteredData[i]);
                }
        }
        printf("\nIf no errors were found, everything's good.\n\n");
        /*
         * Closes the file, as we no longer need to read from it,
         * then prints the highest value and stops the program.
         */
        fclose(file);
}

void testMWI(){
        static const char filename[] = "x_sqr.txt";
        FILE *file = NULL;
        file = fopen ( filename, "r" );
        static const char filename2[] = "x_mwi.txt";
        FILE *file2 = NULL;
        file2 = fopen ( filename2, "r" );
        int Data[3000]={0}, FilteredData[3000]={0},
                        checkData[3000]={0}, i=0;
        for (i=0;i<3000;i++){
                Data[i] = getNextData(file);
                checkData[i]=getNextData(file2);
                FilteredData[i] = movingWindowIntegration(
                                Data, FilteredData, i);
        }

        for (i=0;i<3000;i++){
                if (checkData[i]!=FilteredData[i]){
                        printf("Error in line %d.\n"
                               "Value was supposed to be %d, but was %d
                               "\n", i, checkData[i], FilteredData[i]);
                }
        }
        printf("\nIf no errors were found, everything's good.\n\n");
        /*
         * Closes the file, as we no longer need to read from it,
         * then prints the highest value and stops the program.
         */
        fclose(file);
}
```

C.7.2 Main function for test cases

```
#include <stdio.h>
#include <stdlib.h>
#include "sensor.h"

void chooseTest(int input){
        if (input==1){
                testLow();
        }
        else if (input == 2){
                testHigh();
        }
        else if (input == 3){
                testDerivative();
        }
        else if (input == 4){
                testSquaring();
        }
        else if (input == 5){
                testMWI();
        }
        printf("Please enter your choice of test.\n Press 1 for Low test
        scanf("%d", &input);
        chooseTest(input);
}

int main (void){
        int input=0;
        printf("Please enter your choice of test.\n Press 1 for Low test
        scanf("%d", &input);

        chooseTest(input);
        return 0;
}
```