Embedded Systems

02131

R-peak detection!
2013

Jakob Welner, s124305
Jacob Gjerstruo, s113440

Abstract

The task of this assignment was to develop an Electro-Cardiogram (ECG) scanner using the Pan-Thomkins QRS-Detection Algorithm, and then implement this algorithm in the language C. Using this algorithm, a program was created that could determine a persons heartbeat and give warnings when either the intensity or the heartrate drops below a certain threshold. The program has initially been based on sample data gathered from an ECG scanner and read from a file. However, care has been taken in making it easy to replace the data source at a later point without interfering with the usability. All algorithms provided by the assignment were implemented successfully although certain issues persisted.

Indhold

1 Introduction

This report will investigate the Pan-Thomas QRS detection algorithm, more specificly, if it is possible to implement this algorithm into the company Medembed's next product, which is a wearable ElectroCardioGram scanner (from now on simply called ECG).
The algorithm will be implemented in the programming language C and this report will discuss the following topics: Data acquisition, implementing filters, implementing the peak-detection algorithm, outputting the data to the user and finally, an analysis of the algorithm in terms of power consumption, speed (clock cycles per second) and code size.
For the purpose of this report, we will only implement the C program, and will simulate the data acquisition in real time through data files. Furthermore, the program was developed and tested on an all-purpose processor, whereas in the final product of Medembed, a dedicated processor with limited resources will be used.

1.1 Requirements

Below follows a list of functional and non-functional requirements:

Functional requirements for the application:

- Data acquisition in simulated real-time (Discrete data)
- Implementation of the 5 filters
- Implementation of the R-peak detection
- Output of relevant data to the user, based on the algorithm
- An analysis of our implementation as well as the critical parts, runtime and memory requirements.

Non-functional requirements for the application:

- Using the C-programming language for software development

2 Theory

In order to initiate the structure- and design-process of the program, a number of questions needed to be answered first:

1. How should the real-time data acquisition be simulated?
2. How should the filters be integrated?
3. How should the R-peak detection be integrated?
4. Which data would be relevant for the user, and how should it be shown?
5. How do you determine the critical parts of the software?

2.1 Problem 1: Data acquisition

As specified in the introduction, the dataset used was only sample data and not current readings. To ensure that the program would work on live data as well as samples, a method for reading single datapoints sequentially was implemented, thus simulating real data acquisition. Loading a continous stream of datapoints while being able to work with current as well as previous samples simultaneously.

2.2 Problem 2: Implementation of Filters

To use the QRS-algorithm to it's full extent, the raw data should go through a list of 5 filters, cleaning up the data, amplifying peaks and removing unwanted noise. 4 of these filters use multiple datapoints simultaneously, both current and previous samples, while lowpass and highpass requires access to both current and previous input data as well as previous filterede datapoints from it's own output.

2.3 Problem 3: Implementation of R-peak detection

Once all the filters had been implemented, the actual QRS-algorithm was the next step. The QRS-algorithm would be the one to determines what consitutes a heartbeat, and how to analyse it. Furthermore, it would serve to determine how R-peaks are identified, and after each peak, it will update certain variables to ensure the heartbeats are tracked correctly. These variables will be the ones determining when a heartbeat is certain than a threshold, and if it is, this peak is referred to as an R-peak. Once an R-peak has been determined, data is updated further to classify the next peaks as either heartbeats or noise. This algorithm, however, introduces three challenges - how to determine whether the patient simply has irregular and/or weak heartbeaks or whether the patient is having a heart attack; and how to ensure that every heartbeat is detected correctly. The final challenge was that the algorithm requires that all peaks are stored, and as there is no knowing how many peaks will be found, a list of flexible length is needed.

2.4 Problem 4: Relevant data

The requirement states that the program must show at the very least the value of the latest R-peak that was detected, and also, it must show when this R-peak happened, plus the patients pulse. Furthermore, the patient must be given a warning of the R-peaks value is less than 2000 (as this is a sign of an impending heart attack) and finally, if 5 successive RR-intervals has missed the RR- LOW and HIGH values, another warning must be shown. However, how these data are to be shown has not been defined, and therefore must be specified.

There are several ways to show these data, of which the simplest is to make a text-based screen with the information necessary, and keep this screen updated in real-time, showing the data as we calculate them. Alternatively, the data can be plotted and shown in a graph, or one could combine these two, giving both a graph that keeps getting updated in real-time, as well as text-based information.

2.5 Problem 5: Critical parts

The final issue that was to be clarified is the critical parts of the program. Here, there are three points to discuss, memory requirement, time consumption and power consumption.
Memory requirement is important as the bigger the program is, that is, the more memory it requires, the more physical memory must be implemented into the final device, and therefore, the device will become larger and more cumbersome to wear. Furthermore, more memory also means that the device becomes more power-consuming.
Time consumption is important as the device must be able to read at least 250 data points a second, and if the functions are very time consuming, a stronger processor is needed that requires more power. Furthermore, it is also important that the processor used does not process the data too fast, either, as this will mean that the processor will idle and use power for nothing, meaning a smaller processor can be used that requires less power.
Power consumption is important as it determines how often the user must either recharge the battery or be issued a new battery.

3 Design

When designing the program, it was quickly decided that multiple files should be used for easier readability. Each file would consist of functions concerning a single topic, ie: the filter.c would contain only filter-related methods. The files used were: filter.c, buffer.c, main.c, RRhandling.c and sensor.c.

Furthermore, it was decided that the data from each filter, as well as the raw data, would each be stored for a set amount of time. An amount of 33 samples was choosen, given that no filters request values older than 32 readings before current. Allocating more memory would therefore be a waste and having less samples stored would break the algorithm.

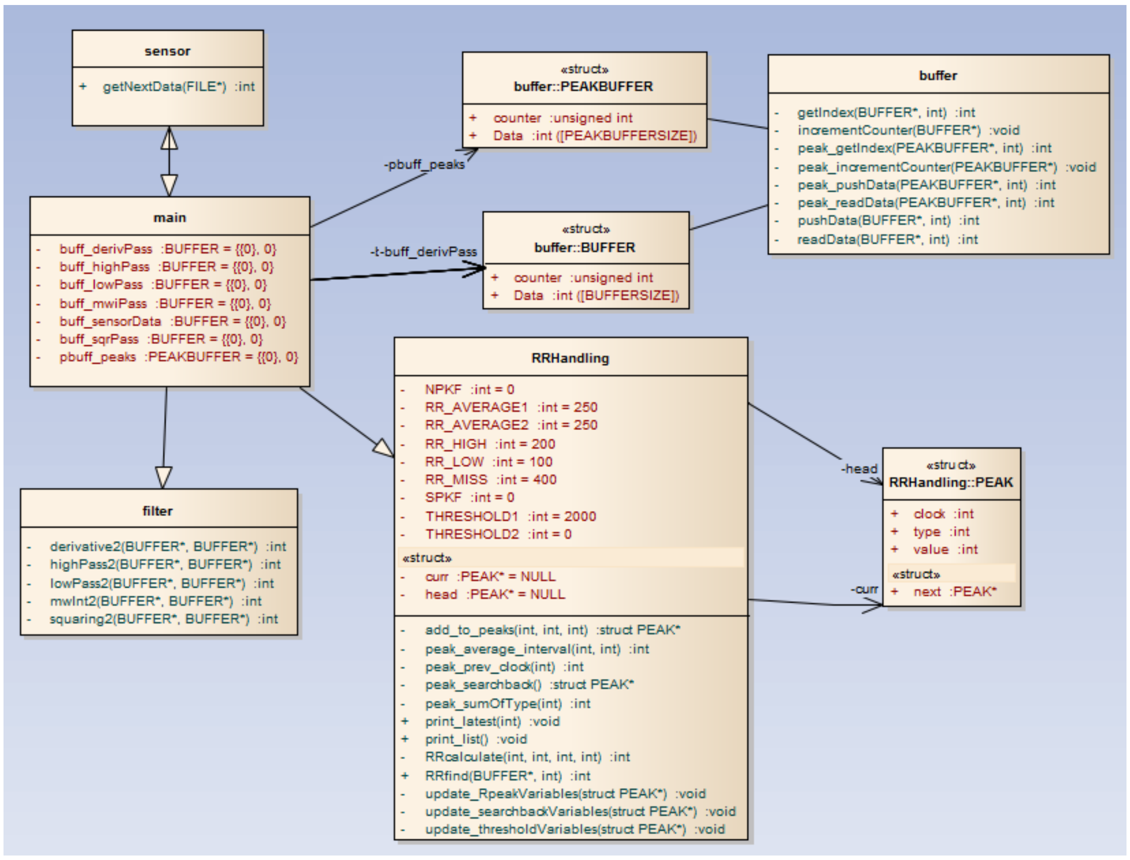
To handle the actual data a buffer was created. The mainloop would run through the data-set, scanning in a point and passing it to the buffer. The buffer would then handle storing and retrieving data through a struct type and 4 dedicated buffer-methods. This array would then be passed on to the filters which would sample individual datapoints, apply the appropriate math and update the buffer-values accordingly.

For a more direct look at how the program is designed, a class diagram has been created, as can be seen on figure 1. This diagram also shows what function is placed where, as well as how each class is relating to eachother.

4 Implementation

4.1 Core structure

To support the overall functionality and structure of the program, a list of core functionalities needed to be created. First, the data flow should be handled - data coming from the sensor should be stored for later processing. Storing such data could be handled in many ways but it was decided to construct a type of buffer which would allow for easy pushing and pulling data, while only keeping a certain amount of history.
The buffer would consist of an array as well as a counter which in conjunction with the buffer-methods would create a circular buffer, where first data to get in would be the first to come out, a so called FIFO list. Should the counter exceed the size of the array, it would return to the first place in the array and continue from there, thus overwriting data older than the length of the array. In order to get around returning huge lists of data between functions, the buffer would instead update the input variables in-place through pointers.
Finally a struct-list was added to contain the R-Peaks. Given that all R-peaks should be saved, the list would be of an arbitrary length. While the C-language is


sensor
+ getNextData(FILE*) :int
«struct»
buffer::PEAKBUFFER
+ counter :unsigned int
+ Data :int ([PEAKBUFFERSIZE])
buffer
- getIndex(BUFFER*, int) :int
- incrementCounter(BUFFER*) :void
- peak_getIndex(PEAKBUFFER*, int) :int
- peak_incrementCounter(PEAKBUFFER*) :void
- peak_pushData(PEAKBUFFER*, int) :int
- peak_readData(PEAKBUFFER*, int) :int
- pushData(BUFFER*, int) :int
- readData(BUFFER*, int) :int
-pbuff_peaks
main
- buff_derivPass :BUFFER = {{0}, 0}
- buff_highPass :BUFFER = {{0}, 0}
- buff_lowPass :BUFFER = {{0}, 0}
- buff_mwiPass :BUFFER = {{0}, 0}
- buff_sensorData :BUFFER = {{0}, 0}
- buff_sqrPass :BUFFER = {{0}, 0}
- pbuff_peaks :PEAKBUFFER = {{0}, 0}
-t-buff_derivPass
«struct»
buffer::BUFFER
+ counter :unsigned int
+ Data :int ([BUFFERSIZE])
RRHandling
- NPKF :int = 0
- RR_AVERAGE1 :int = 250
- RR_AVERAGE2 :int = 250
- RR_HIGH :int = 200
- RR_LOW :int = 100
- RR_MISS :int = 400
- SPKF :int = 0
- THRESHOLD1 :int = 2000
- THRESHOLD2 :int = 0
«struct»
- curr :PEAK* = NULL
- head :PEAK* = NULL
- add_to_peaks(int, int, int) :struct PEAK*
- peak_average_interval(int, int) :int
- peak_prev_clock(int) :int
- peak_searchback() :struct PEAK*
- peak_sumOfType(int) :int
+ print_latest(int) :void
+ print_list() :void
- RRcalculate(int, int, int, int) :int
+ RRfind(BUFFER*, int) :int
- update_RpeakVariables(struct PEAK*) :void
- update_searchbackVariables(struct PEAK*) :void
- update_thresholdVariables(struct PEAK*) :void
-head
«struct»
RRHandling::PEAK
+ clock :int
+ type :int
+ value :int
«struct»
-curr
+ next :PEAK*
filter
- derivative2(BUFFER*, BUFFER*) :int
- highPass2(BUFFER*, BUFFER*) :int
- lowPass2(BUFFER*, BUFFER*) :int
- mwInt2(BUFFER*, BUFFER*) :int
- squaring2(BUFFER*, BUFFER*) :int

Figur 1: The above diagram shows how each of the classes are related to eachother

very poor at handling arrays with ill-defined length, a linked list was used instead. The linked list works by defining a Struct type which contains a variable pointer to another struct of the same type. Adding each new peak instance to the previous by a pointer variable, would then create a so-called linked list of arbitrary length.

4.2 Real-time data acquisition

As described, the program had to acquire the data and process it in real-time. To ensure this simulation, the program would load one data point, then pass it to a buffer which in turn would get passed to one filter at a time. Finally it would be passed to the peak detection and subsequently, to the R-peak detection. Once all these calculations had been done, it would load a new data point and repeat the loop.
This while-loop would run through the entire data-set, and would not stop before it had reached the end of the file (in the finished product, it would of course run until the battery runs dry).
Therefore, the combination of the while-loop that runs forever, and the continual loading and subsequent calculations through the buffer solves the challenge of real-time data acquisition.

4.3 Implementation of Filters

The first implementation of the filters were simply done with strict if/else sentences. However, this was changed to using an adaptive index that, should it reach negative values, moves to the end of the array instead. This way, the program should never encounter data under- or overflow, and thereby, the first of the challenges were solved.
The second challenge, to ensure the correct transfer of data from one filter to another, was solved by using two arrays, one for the raw data and one for the filtered data. Each array is then passed to the next part of the program, ensuring all filters have both raw and processed data to be operated on, as 4 of them requires.

Therefore, the use of two arrays and an adaptive index solves the challenge of the implementation of the filters.

To allow for easy access to the saved datapoints at different given points in time, the readData buffer method was extended to include a time offset input value. This would work as an index for the data array, though combined with the built-in counter, would amount to a number of steps back in time. ReadData(buffer, 0) would read latest pushed data from 'buffer' and readData(buffer, 3) would read the data stored 3 loops earlier. Furthermore, passing pointers of buffer structs to the buffer-methods allowed to change the provided buffers inplace
Having extended the method successfully while simply returning 0-values when requesting data before 1st loop, implementing the filters was a matter of writing the equations from the assignment and requesting the appropriate datapoints directly. Allowing lowpass and highpass to read from their own output was then elementary. Passing a pointer of the input- and another of the output-buffer would give each filter access to read/write on both

4.4 Implementation of RPeakDetection

When implementing the RPeakDetection, the filtered data was run through a simple local-maxima detection and saved in a new list. As noted in Problem 3 - Implementation of RPeakDetection, this list needed to be of flexible length, and looking at the different options for implementing such a list, the decision fell on linked lists through structs. This struct contains a 'next'-variable with a pointer to the same type of struct. For each new instance of the struct, the intern next-value is set to point at the previous struct, thus linking the two together. Having implemented the linked list, 3 variables were added to the struct; VALUE, CLOCK and TYPE. Accordingly, these would contain the signal-strength of the peak, at what time the sample had occurred and what type it had been classified as. TYPE was handled as integer values where -1 meant noise-peak, 0 meant a local-maxima peak, 1 meant an R-peak and 2 meant a regular R-peak. This would allow for all the peaks to be stored in a single list while enabling searching through the list and filtering for several types at a time while comparing results. After this had been set up, the algorithm for determining r-peaks was implemented. To allow a few samples to appear before trying to enhance variable values, a MINSAMPLES value was defined.

Therefore, the combination of the algorithm and the linked list solves the challenges of the implementation of RPeakDetection.

4.5 Relevant data

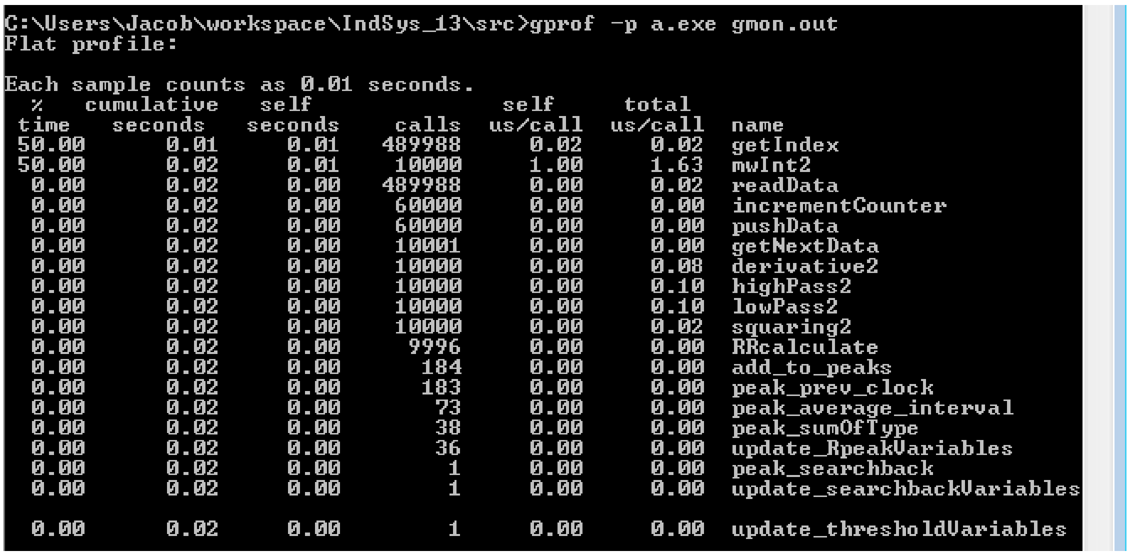
To display the relevant data, a simple text-based display was created. This would show the required information, that is, the value of the latest R-peak, the time of the R-peak, the patients pulse. Furthermore, it would also display the warning and the RR-LOW and RR-HIGH values if more than 5 RR-intervals had been missed.

Therefore, the challenge of the relevant data has been solved.

4.6 Critical parts

As mentioned in the theory, the three critical parts of the program is memory consumption, power consumption and time consumption. The program was initially tested on a machine running a powerful all-purpose Intel Core i7-2630M CPU with a clock speed of 2GHz and consumes a maximum of 45 Watt.
As can be seen on figure 2, the processor manages to process 10.000 data points on 0.02 miliseconds which is far more than what is required (250 data points per second).
In terms of power, as said, the current processor on which the program was tested consumes a maximum of 45 Watt. This can be reduced by either reducing the calculation-power of the processor (clock frequency) or the size of the processor.

In regards to size, figure 3 shows the console after 10000 data samples, and of particular interest is the "Size of Stuff" and the "elements in peak", as these denominate the size of the linked list of peaks (in bytes), as well as the amount of elements in that list. As can be seen, after 10000 data points, 98 peaks has been found, and the total size is 1568 bytes, making it by far the largest part of the program (the second-to-largest part is the arrays of the filters, each having the size of 136 bytes)

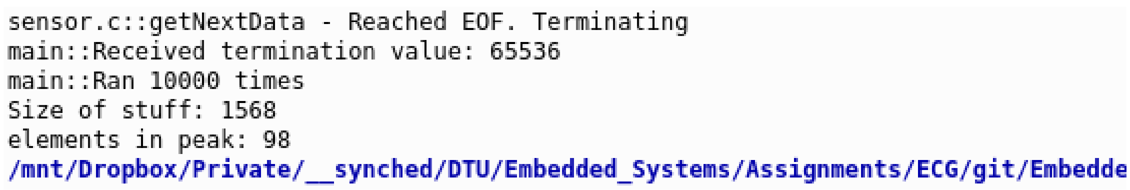


```
C:\Users\Jacob\workspace\IndSys_13\src>gprof -p a.exe gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 50.00      0.01     0.01   489988     0.02     0.02  getIndex
 50.00      0.02     0.01    10000     1.00     1.63  mwInt2
  0.00      0.02     0.00   489988     0.00     0.02  readData
  0.00      0.02     0.00    60000     0.00     0.00  incrementCounter
  0.00      0.02     0.00    60000     0.00     0.00  pushData
  0.00      0.02     0.00    10001     0.00     0.00  getNextData
  0.00      0.02     0.00    10000     0.00     0.08  derivative2
  0.00      0.02     0.00    10000     0.00     0.10  highPass2
  0.00      0.02     0.00    10000     0.00     0.10  lowPass2
  0.00      0.02     0.00    10000     0.00     0.02  squaring2
  0.00      0.02     0.00     9996     0.00     0.00  RRcalculate
  0.00      0.02     0.00      184     0.00     0.00  add_to_peaks
  0.00      0.02     0.00      183     0.00     0.00  peak_prev_clock
  0.00      0.02     0.00       73     0.00     0.00  peak_average_interval
  0.00      0.02     0.00       38     0.00     0.00  peak_sumOfType
  0.00      0.02     0.00       36     0.00     0.00  update_RpeakVariables
  0.00      0.02     0.00        1     0.00     0.00  peak_searchback
  0.00      0.02     0.00        1     0.00     0.00  update_searchbackVariables

  0.00      0.02     0.00        1     0.00     0.00  update_thresholdVariables
```

Figur 2: The table above shows the time spent in the various filters in our program. It should be noted that this profiling was done without code optimization.



```
sensor.c::getNextData - Reached EOF. Terminating
main::Received termination value: 65536
main::Ran 10000 times
Size of stuff: 1568
elements in peak: 98
/mnt/Dropbox/Private/__synched/DTU/Embedded_Systems/Assignments/ECG/git/Embedde
```

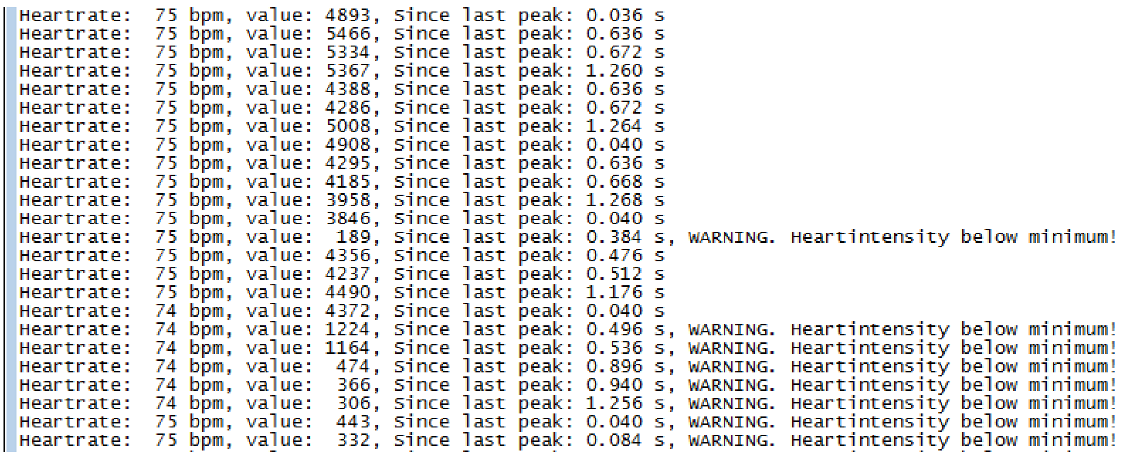
Figur 3: The screenshot above is a screenshot from the console after the program has been run on 10000 data samples.

5 Results

Figure 4 shows a screenshot of the output from the RPeak detection algorithm. It shows the output, both when the user has a normal heartbeat (the first part), the warning when the users heartbeat suddenly drops dangerously low, and the warning when the user has a heart attack. Furthermore, it always shows the heartrate (in Beats Per Minute (BMP)), as well as the RPeak values and the time since the last RPeak. The full output can be seen in section B

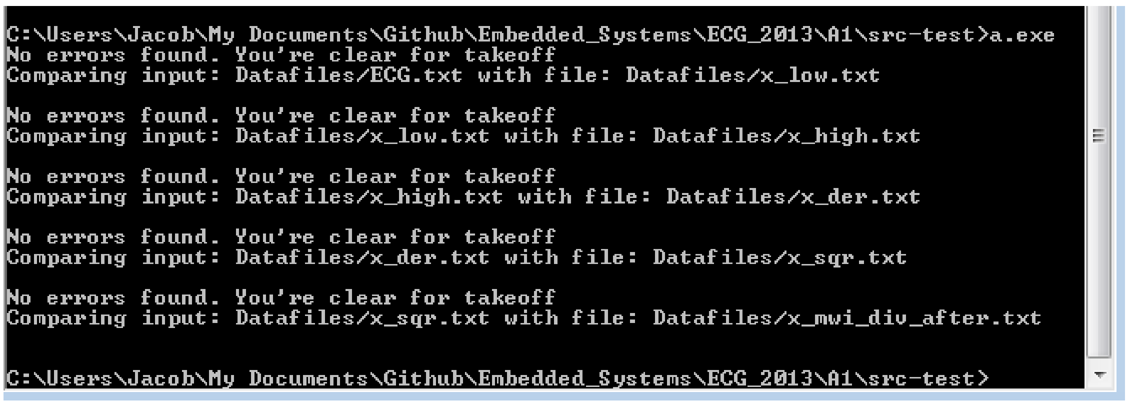
5.1 Test results of filters

For the purpose of ensuring the correctness of the filters, a small program was created that would go through each datapoint, load in the value from its supposed data source, run this point through the filter and compare it to the result. Figure 5 shows the result of this program, showing that all filters calculates correctly. The code for this program can be seen in section D.6



```
Heartrate:  75 bpm, value: 4893, Since last peak: 0.036 s
Heartrate:  75 bpm, value: 5466, Since last peak: 0.636 s
Heartrate:  75 bpm, value: 5334, Since last peak: 0.672 s
Heartrate:  75 bpm, value: 5367, Since last peak: 1.260 s
Heartrate:  75 bpm, value: 4388, Since last peak: 0.636 s
Heartrate:  75 bpm, value: 4286, Since last peak: 0.672 s
Heartrate:  75 bpm, value: 5008, Since last peak: 1.264 s
Heartrate:  75 bpm, value: 4908, Since last peak: 0.040 s
Heartrate:  75 bpm, value: 4295, Since last peak: 0.636 s
Heartrate:  75 bpm, value: 4185, Since last peak: 0.668 s
Heartrate:  75 bpm, value: 3958, Since last peak: 1.268 s
Heartrate:  75 bpm, value: 3846, Since last peak: 0.040 s
Heartrate:  75 bpm, value:  189, Since last peak: 0.384 s, WARNING. Heartintensity below minimum!
Heartrate:  75 bpm, value: 4356, Since last peak: 0.476 s
Heartrate:  75 bpm, value: 4237, Since last peak: 0.512 s
Heartrate:  75 bpm, value: 4490, Since last peak: 1.176 s
Heartrate:  74 bpm, value: 4372, Since last peak: 0.040 s
Heartrate:  74 bpm, value: 1224, Since last peak: 0.496 s, WARNING. Heartintensity below minimum!
Heartrate:  74 bpm, value: 1164, Since last peak: 0.536 s, WARNING. Heartintensity below minimum!
Heartrate:  74 bpm, value:  474, Since last peak: 0.896 s, WARNING. Heartintensity below minimum!
Heartrate:  74 bpm, value:  366, Since last peak: 0.940 s, WARNING. Heartintensity below minimum!
Heartrate:  74 bpm, value:  306, Since last peak: 1.256 s, WARNING. Heartintensity below minimum!
Heartrate:  75 bpm, value:  443, Since last peak: 0.040 s, WARNING. Heartintensity below minimum!
Heartrate:  75 bpm, value:  332, Since last peak: 0.084 s, WARNING. Heartintensity below minimum!
```

Figur 4: A screenshot of the output of the RPeak detection.



```
C:\Users\Jacob\My Documents\Github\Embedded_Systems\ECG_2013\A1\src-test>a.exe
No errors found. You're clear for takeoff
Comparing input: Datafiles/ECG.txt with file: Datafiles/x_low.txt

No errors found. You're clear for takeoff
Comparing input: Datafiles/x_low.txt with file: Datafiles/x_high.txt

No errors found. You're clear for takeoff
Comparing input: Datafiles/x_high.txt with file: Datafiles/x_der.txt

No errors found. You're clear for takeoff
Comparing input: Datafiles/x_der.txt with file: Datafiles/x_sqr.txt

No errors found. You're clear for takeoff
Comparing input: Datafiles/x_sqr.txt with file: Datafiles/x_mwi_div_after.txt


C:\Users\Jacob\My Documents\Github\Embedded_Systems\ECG_2013\A1\src-test>
```

Figur 5: A screenshot of the output of the tests of the filters.

6 Discussion

As mentioned in the critical parts, the processor which the program is tested on is more than powerful enough to fulfill the requirements of 250 data points per second. Therefore, the final processor that is to be used for the ECG scanner should be much less powerful - it makes little sense to use a very powerful processor if it idles most of the time, using power for nothing - and thus, the power consumption will be reduced greatly.
However, if the processor had been too slow, the code could be optimized for speed. To do this, the two functions getIndex and mwInt2 should be examined, as it was determined through a profiler (the GNU Gprof profiler) that it is in these two functions that the most time is spent (see figure 2).
Another way to optimize power would be to use the technical improvements of modern time to make the battery recharge during daily use, for instance by converting the kinetic energy of walking into power for the scanner.

Regarding the size, the program is generally optimized mostly in terms of speed - currently, an array of 33 data points is used for each filter, but this could be optimized down to using only 2 arrays - one for input, one for output. However, this would also increase the complexity of the task by a great deal and therefore, it was decided to simply use an array for each filter.

In terms of what should be made as software and what should be put into hardware, it should first be determined which tasks are the most processor-heavy. However, due to the profiler, this is already known - the tasks are getIndex and mwInt2. Therefore, it would also make sense to develop a special processor for these two specific tasks, using a coprocessor for each of these tasks, and a more general-purpose-like processor for the rest of the tasks.

6.1 Improvements

There are two types of improvements to be discussed - Hardware improvements and software improvements.

Within the domain of software improvements, several improvements could be done. The first improvement would be to start the QRS algorithm with dynamically allocated values, rather than hardcoded values, for all the variables. This would improve the correctness of the initial part of the algorithm, ensuring also that it works well on both males and females with varying heart-sizes.
The second improvement would be to set a maximum size for the list of peaks - currently, it is unlimited and will only grow as the device lives on - setting a maximum size and using the same theory as on the arrays for the filters could decrease the size of this list.
The third improvement would be to implement a timer that starts when a user has a heart attack, and continues to update and show the data so that when rescuers come to assist the user, they will know precisely how long the heart attack has lasted.
The final improvement that was identified were regarding the warning showed during heart attacks - currently, this warning only lasts until 5 regular heartbeats has been detected in a row, but in the final device, a doctor should be questioned upon just how many regular heartbeats needs to be detected before it is safe to assume that the patient is stable again.

Within the domain of hardware improvements, there are two main upgrades that can be done. The first is to convert the entire project to run on a dedicated processor, using co-processors for the requiring tasks (getIndex and mwInt2 in this case). This will improve both power consumption and calculation speed.
The second improvement would be to implement modern technologies of converting kinetic energy from, for instance, walking, into power and using this to recharge the battery of the device, thus increasing its lifespan.

7 Conclusion

Implementing the algorithm has been done successfully, enabling the detection of heartbeats according to time and fulfilling all requirements. Furthermore, the processor used in the final product can easily be downscaled, using a lower clock frequency to reduce power consumption. Finally, all the challenges discovered and discussed in the Theory has been solved.

Litteratur

[1] Michael Reibel Boesen, Linas Kaminskas, Paul Pop, Karsten Juul Frederiksen
Assignment 1: Software implementation of a personal ECG scanner
3rd Edition
2013.

[2] http://www.notebookcheck.net/Intel-Core-i7-2630QM-Notebook-Processor.41483.0.html
Date of use: 25/09/2013

Appendix

A Who wrote what

Jacob Gjerstrup, s113440 wrote: Abstract, Introduction, Theory, Discussion (50%), Conclusion
Jakob Welner, s124305 wrote: Design, Implementation, Results, Discussion (50%)

B Output of the RPeakDetection

```
Heartrate: 60 bpm, Intencity: 5062, Sample nr.:  153
Heartrate: 60 bpm, Intencity: 4783, Sample nr.:  312
Heartrate: 60 bpm, Intencity: 5018, Sample nr.:  470
Heartrate: 94 bpm, Intencity: 5404, Sample nr.:  627
Heartrate: 94 bpm, Intencity: 4846, Sample nr.:  785
Heartrate: 94 bpm, Intencity: 4910, Sample nr.:  943
Heartrate: 94 bpm, Intencity: 4496, Sample nr.: 1102
Heartrate: 94 bpm, Intencity: 4904, Sample nr.: 1261
Heartrate: 94 bpm, Intencity: 4805, Sample nr.: 1269
Heartrate: 94 bpm, Intencity: 4983, Sample nr.: 1419
Heartrate: 94 bpm, Intencity: 4754, Sample nr.: 1577
Heartrate: 94 bpm, Intencity: 4691, Sample nr.: 1585
Heartrate: 94 bpm, Intencity: 4810, Sample nr.: 1733
Heartrate: 95 bpm, Intencity: 4944, Sample nr.: 1891
Heartrate: 94 bpm, Intencity: 4866, Sample nr.: 1899
Heartrate: 94 bpm, Intencity: 4792, Sample nr.: 2048
Heartrate: 95 bpm, Intencity: 4654, Sample nr.: 2164
Heartrate: 95 bpm, Intencity: 4420, Sample nr.: 2342
Heartrate: 95 bpm, Intencity: 4209, Sample nr.: 2502
Heartrate: 95 bpm, Intencity: 5341, Sample nr.: 2669
Heartrate: 95 bpm, Intencity: 4494, Sample nr.: 2979
Heartrate: 95 bpm, Intencity: 4614, Sample nr.: 3137
Heartrate: 70 bpm, Intencity: 4529, Sample nr.: 3145
Heartrate: 70 bpm, Intencity: 5249, Sample nr.: 3296
Heartrate: 70 bpm, Intencity: 4862, Sample nr.: 3454
Heartrate: 70 bpm, Intencity: 4982, Sample nr.: 3614
Heartrate: 70 bpm, Intencity: 5452, Sample nr.: 3772
Heartrate: 59 bpm, Intencity: 5358, Sample nr.: 3928
Heartrate: 59 bpm, Intencity: 4380, Sample nr.: 4086
Heartrate: 59 bpm, Intencity: 5001, Sample nr.: 4243
Heartrate: 59 bpm, Intencity: 4295, Sample nr.: 4400
```

```
Heartrate:  51 bpm, Intencity: 3950, Sample nr.:  4559
Heartrate:  51 bpm, Intencity: 4351, Sample nr.:  4677
Heartrate:  51 bpm, Intencity: 4483, Sample nr.:  4852
Heartrate:  51 bpm, Intencity: 1219, Sample nr.:  4975   ## Heartintensity below minimum! ##
Heartrate:  52 bpm, Intencity: 1172, Sample nr.:  4984   ## Heartintensity below minimum! ##
Heartrate:  52 bpm, Intencity:  477, Sample nr.:  5074   ## Heartintensity below minimum! ##
Heartrate:  52 bpm, Intencity:  447, Sample nr.:  5173   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  52 bpm, Intencity:  399, Sample nr.:  5272   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  52 bpm, Intencity:  713, Sample nr.:  5371   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  52 bpm, Intencity: 1257, Sample nr.:  5471   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  52 bpm, Intencity:  446, Sample nr.:  5561   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity:  551, Sample nr.:  5570   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1283, Sample nr.:  5672   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1198, Sample nr.:  5682   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1228, Sample nr.:  5772   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1080, Sample nr.:  5782   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1105, Sample nr.:  5872   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  53 bpm, Intencity:  914, Sample nr.:  5972   ## Heartintensity below minimum! ##  ## Missed more than 5
```

```
peaks. Dude, you're dying! ##
Heartrate:  53 bpm, Intencity:  990, Sample nr.:  6073   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  53 bpm, Intencity: 1371, Sample nr.:  6173   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1662, Sample nr.:  6274   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1440, Sample nr.:  6284   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity:  706, Sample nr.:  6365   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity:  904, Sample nr.:  6374   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1193, Sample nr.:  6474   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  53 bpm, Intencity:  938, Sample nr.:  6574   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  53 bpm, Intencity:  783, Sample nr.:  6584   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  53 bpm, Intencity: 1084, Sample nr.:  6675   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  53 bpm, Intencity: 1002, Sample nr.:  6777   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1204, Sample nr.:  6879   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1152, Sample nr.:  6889   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity:  806, Sample nr.:  6980   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
```

```
Heartrate:  50 bpm, Intencity:  697, Sample nr.:  6991  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1337, Sample nr.:  7082  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1069, Sample nr.:  7184  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1217, Sample nr.:  7286  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1095, Sample nr.:  7297  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  50 bpm, Intencity: 1261, Sample nr.:  7386  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1162, Sample nr.:  7488  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1318, Sample nr.:  7589  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1429, Sample nr.:  7690  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1019, Sample nr.:  7791  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1482, Sample nr.:  7894  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity:  577, Sample nr.:  7987  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity:  786, Sample nr.:  7996  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity:  589, Sample nr.:  8007  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity:  904, Sample nr.:  8098  ## Heartintensity below minimum! ##  ## Missed more than 5
```

```
peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1044, Sample nr.:  8200   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1089, Sample nr.:  8302   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1128, Sample nr.:  8403   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1456, Sample nr.:  8505   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1036, Sample nr.:  8606   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1199, Sample nr.:  8708   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1354, Sample nr.:  8810   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1240, Sample nr.:  8911   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1409, Sample nr.:  9013   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1579, Sample nr.:  9116   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1332, Sample nr.:  9126   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1321, Sample nr.:  9218   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1286, Sample nr.:  9319   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1103, Sample nr.:  9421   ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
```

```
Heartrate:  49 bpm, Intencity: 1136, Sample nr.:  9524  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1342, Sample nr.:  9626  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1396, Sample nr.:  9728  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1814, Sample nr.:  9830  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1349, Sample nr.:  9933  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
Heartrate:  49 bpm, Intencity: 1124, Sample nr.:  9944  ## Heartintensity below minimum! ##  ## Missed more than 5 peaks. Dude, you're dying! ##
sensor.c::getNextData - Reached EOF. Terminating
main::Received termination value: 65536
main::Ran 10000 times
```

C Sourcecode - introductionary exercises

C.1 ReadFromFile

Below is the sourcecode for the introductionary-exercise (From september the 4th) - more precisely, the ReadFromFile source-code.

```c
#include <stdio.h>

int main(void)
{
    printf("Hey, I'm reading a file!\n");

    static const char filename[] = "ECG.txt";
    FILE *file = fopen ( filename, "r");
    int line, rVal;
    int max;

    rVal = fscanf(file, "%i", &line);
    max = line;

    while(rVal != EOF) {

        rVal = fscanf(file, "%i", &line);
        if (max < line) {
            printf("Found a new larger number: %i \n", line);
            max=line;
        }
    }
    printf("Finally, this is the largest numer: %i\n", max);
    return 0;
}
```

C.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;
}
```

D 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.

D.1 Buffer

```c
#include <stdio.h>
#include "buffer.h"

// Bufferindex
int pushData(BUFFER* buffer, int data)
{
    //printf("Buffer::pushData: Adding stuff to the buffer
       !: %i\n", data);
    (*buffer).Data[(*buffer).counter] = data;
    incrementCounter(buffer);

    return 0;
}


// Getting values based on index offset
// compared to current buffer counter
int readData(BUFFER* buffer, int offset)
{
    int index = getIndex(buffer, offset);
    return (*buffer).Data[index];
}


void incrementCounter(BUFFER* buffer)
{
    (*buffer).counter++;
    if ((*buffer).counter >= BUFFERSIZE) {
        (*buffer).counter -= BUFFERSIZE;
    }
}


int getIndex(BUFFER *buffer, int offset) {
    int index = (*buffer).counter - offset -1;
    if (index < 0){
        index = index+BUFFERSIZE;
    }

    return index;
}
```

D.2 Filters

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



int lowPass2(BUFFER* inputBuffer, BUFFER* filtered) {
    /*
     *  GroupDelay: 25 ms
     */

    // Retrieving values
    int x = readData(inputBuffer, 0);
    int x6 = readData(inputBuffer, 6);
    int x12 = readData(inputBuffer, 12);
    int y1 = readData(filtered, 0);
    int y2 = readData(filtered, 1);

    // Filter math
    int y = (2*y1-y2) + ((x - 2*x6 + x12) / 32);

    // pushing data back to buffer object
    pushData(filtered, y);

    return 0;
}


int highPass2(BUFFER* inputBuffer, BUFFER* filtered) {
    /*
     *  GroupDelay: 80 ms
     */

    // Retrieving values
    int x = readData(inputBuffer, 0);
    int x16 = readData(inputBuffer, 16);
    int x17 = readData(inputBuffer, 17);
    int x32 = readData(inputBuffer, 32);
    int y1 = readData(filtered, 0);

    // Filter math
    int y = y1-(x/32)+x16-x17+(x32/32);

    // Pushing data back to buffer object
    pushData(filtered, y);

    return 0;
}


int derivative2(BUFFER* inputBuffer, BUFFER* filtered) {
    /*
     *  GroupDelay: 10 ms
     */

    // Retrieving values
    int x = readData(inputBuffer, 0);
    int x1 = readData(inputBuffer, 1);
    int x3 = readData(inputBuffer, 3);
    int x4 = readData(inputBuffer, 4);

    // Filter math
    int y = (2*x+x1-x3-2*x4) / 8;

    // Pushing data back to buffer object
    pushData(filtered, y);

    return 0;
}


int squaring2(BUFFER* inputBuffer, BUFFER* filtered) {
    /*
     *  GroupDelay: 0 ms
     */

    // Retrieving values
    int x = readData(inputBuffer, 0);

    // Filter math
    int y = x*x;

    // Pushing data back to buffer object
    pushData(filtered, y);

    return 0;
}
```

```c


int mwInt2(BUFFER* inputBuffer, BUFFER* filtered) {
    /*
     *  GroupDelay: 72.5 ms
     */

    int N = 30;

    // Dynamic Retrieving of values
    int i = 0;
    int sum = 0;

    for (i = 0; i < N; i++) {
        sum += readData(inputBuffer, i);
    }

    // Filter math
    int y = sum / N;

    // Pushing data back to buffer object
    pushData(filtered, y);

    return 0;
}
```

D.3 RPeakDetection

```c
#include <stdio.h>
#include <stdlib.h>

#define MINSAMPLES 3 // Minimum number of samples before
    updating variables
#define VERBOSE 1

struct PEAK {
    int clock;          // If older than buffer size cannot
        be reevaluated
    int type;           // -1 noise, 0 peak, 1 R-peak, 2
       regular R-peak
    int value;          // speaks for itself
    struct PEAK *next;
};


```

```
struct PEAK *head = NULL;
struct PEAK *curr = NULL;


int RR_AVERAGE1 = 250;
int RR_AVERAGE2 = 250;
int RR_LOW = 100;
int RR_HIGH = 200;
int RR_MISS = 400;

int SPKF = 0;
int NPKF = 0;
int THRESHOLD1 = 2000;
int THRESHOLD2 = 0;

int missed_peaks = 0;



// Adding node to beginning of list
struct PEAK* add_to_peaks(int clock, int val, int type) {

    struct PEAK *ptr = (struct PEAK*)malloc(sizeof(struct
      PEAK));

    ptr->value = val;
    ptr->clock = clock;
    ptr->type = type;
    ptr->next = NULL;

    if(NULL == head) head = curr = ptr;
    else {
        ptr->next = head;
        head = ptr;
    }
    return ptr;
}
```

```


int peak_prev_clock(int type)
{
    struct PEAK *ptr = head;
    ptr = ptr->next; // Skipping current element
    while (NULL != ptr){
        if (ptr->type == type) {
            return ptr->clock;
        };
        ptr = ptr->next;
    }
    return 0;
}





int peak_sumOfType(int type)
{
    struct PEAK *ptr = head;
    int count = 0;
    while (NULL != ptr){
        if (ptr->type >= type) {
            count++;
        };
        ptr = ptr->next;
    }
    return count;
}



struct PEAK* peak_searchback()
{
    struct PEAK *ptr = head;


    int i = 0;

    while (NULL != ptr){
        i++;
        if (ptr->type >= 0) {
            if (ptr->value > THRESHOLD2) {
                ptr->type = 2;
                printf("Searchback'ed %i levels deep\n", i)
                    ;
                return ptr;
            }
        };

        ptr = ptr->next;
    }
    return NULL;
}



int peak_average_interval(int type, int amount)
{

    struct PEAK *ptr = head;
    if (NULL != ptr) {
        int clock = ptr->clock;

        int clockSum = 0;
        int i = 0;
        ptr = ptr->next;
        while (NULL != ptr){
        if (ptr->type >= type) {
            int newClock = ptr->clock;
            int tempDiff = clock - newClock;

            clockSum += tempDiff; // Summing value of
                found peaks
            clock = newClock;
            i++;
        };
        if (amount == i) break;
        ptr = ptr->next;
        }
        if (i) {
            int avg = clockSum/i;
            return avg;
        }
    }
    return 0;
```

```
}






void update_RpeakVariables(struct PEAK* ptr) {
     // get latest 8 RR-intervals
     if (NULL != ptr) {
          int peakVal = ptr->value;
          SPKF = 0.125 * peakVal + 0.875 * NPKF;
          RR_AVERAGE2 = peak_average_interval(2, 8);
          RR_AVERAGE1 = peak_average_interval(1, 8);
          RR_LOW = 0.92 * RR_AVERAGE2;
          RR_HIGH = 1.16 * RR_AVERAGE2;
          RR_MISS = 1.66 * RR_AVERAGE2;
          THRESHOLD1 = NPKF + 0.25 * (SPKF-NPKF);

          if (VERBOSE > 1){
               printf("SPKF: %i\nRR_AVERAGE2: %i\nRR_AVERAGE1:
                   %i\nRR_LOW: %i\nRR_HIGH: %i\nRR_MISS: %i\
                   nTHRESHOLD1: %i\n",
                         SPKF,
                         RR_AVERAGE2,
                         RR_AVERAGE1,
                         RR_LOW,
                         RR_HIGH,
                         RR_MISS,
                         THRESHOLD1);
          }
     }
     else printf("ERROR: update_RpeakVariables:: input ptr ==
        NULL\n");

     return;
}


void update_searchbackVariables(struct PEAK* ptr) {
     // get latest 8 RR-intervals
     if (NULL != ptr) {
          int peakVal = ptr->value;
          SPKF = 0.25 * peakVal + 0.75 * SPKF;
        RR_AVERAGE1 = peak_average_interval(1, 8);
        RR_LOW = 0.92 * RR_AVERAGE1;
        RR_HIGH = 1.16 * RR_AVERAGE1;
        RR_MISS = 1.66 * RR_AVERAGE1;

        THRESHOLD1 = NPKF + 0.25 * (SPKF-NPKF);
        THRESHOLD2 = 0.5 * THRESHOLD1;

        if (VERBOSE > 1){
            printf("\nUpdate_searchbackVariables:\nSPKF: %i
                \nRR_AVERAGE1: %i\nRR_AVERAGE2: %i\nRR_LOW: %
                i\nRR_HIGH: %i\nRR_MISS: %i\nTHRESHOLD1: %i\
                nTHRESHOLD2: %i\n",
                    SPKF,
                    RR_AVERAGE1,
                    RR_AVERAGE2,
                    RR_LOW,
                    RR_HIGH,
                    RR_MISS,
                    THRESHOLD1,
                    THRESHOLD2);
        }
    }
    else {
        printf("ERROR: update_searchbackVariables:: input_ptr
            == NULL\n");
    }
    return;
}



void update_thresholdVariables( struct PEAK* ptr ) {
    if (NULL != ptr) {
        int peakVal = ptr->value;
        NPKF = 0.125 * peakVal + 0.875 * NPKF;
        THRESHOLD1 = NPKF + 0.25 * (SPKF - NPKF);
        THRESHOLD2 = 0.5 * THRESHOLD1;
    }
    else printf("ERROR: update_thresholdVariables:: input
        ptr == NULL\n");

    return;
}
```

```c


/***** Sampling 3 points on each side of test-point *****/
int checkPeak(int samples[]) {

    if ( samples[0] < samples[1] &&
         samples[1] < samples[2] &&
         samples[2] < samples[3] &&
         samples[3] > samples[4] &&
         samples[4] > samples[5] &&
         samples[5] > samples[6] ) return 1;

    /*
    if (
         samples[2] < samples[3] &&
         samples[3] > samples[4] ) return 1;*/
    return 0;
}






int RRcalculate(int x1, int samples[], int clock)
{

    struct PEAK *ptr = NULL;

    // Checks for peak
    if ( checkPeak(samples) ) {
        ptr = add_to_peaks(clock-1, x1, 0); // Point is a
            peak. Save in list, type 0

        if (ptr->value > THRESHOLD1) {
            ptr->type = 1; // Classifying as R-peak


            //printf("clock: %i, r-peak val: %i\n", clock
                -1, x1);
            int lastClock = peak_prev_clock(2);
            int timediff = clock - lastClock;


            /************ USER output *************/
            // For every R-peak print this:
            if (VERBOSE == 1) {
                printf("Heartrate: %3i bpm, Intencity: %4i, Last peak: %.3fs", (int)(1.0/RR_AVERAGE2*60.0*250.0), x1, timediff/250.0);
                //printf("Heartrate: %3i bpm, value: %4i, Since last peak: %i s", (RR_AVERAGE2*60)/250, x1, timediff);
                if (x1 < 2000) printf("   ## Heartintensity below minimum! ##");
                if (missed_peaks > 5) printf("   ## Missed more than 5 peaks. Dude, you're dying! ##");
                printf("\n");
            }
            else if (VERBOSE == 0) {
                FILE *file;
                file = fopen("output.txt","a+");
                fprintf(file,"Heartrate: %3i bpm, Intencity: %4i, Last peak: %.3fs", (int)(1.0/RR_AVERAGE2*60.0*250.0), x1, timediff/250.0);
                if (x1 < 2000) fprintf(file,", WARNING. Heartintensity below minimum!");
                fprintf(file,"\n");
                fclose(file);
            }
            /************************************/


            if (RR_LOW < timediff && timediff < RR_HIGH ) {
                if (missed_peaks > 0) missed_peaks--;

                ptr->type = 2; // Classify as regular R-peak
                // Only updating variables if more than MINSAMPLES are available

                if (peak_sumOfType(2) >= MINSAMPLES) {
                    update_RpeakVariables( ptr );
                }
            }
```

```
            else if (RR_MISS < timediff){
                // searchback
                update_searchbackVariables( peak_searchback
                   () );
            }
            else {
                if (missed_peaks < 10) missed_peaks++;
            }
        }
        else {
            update_thresholdVariables( ptr );
        }
    }
    return 0;
}




/* Function for determining size of PEAK array */
int sizeof_peaks() {
    struct PEAK *ptr = head;
    int size = 0;
    int i = 0;

    while(ptr != NULL)
    {
        size += sizeof(struct PEAK);
        ptr = ptr->next;
        i++;
    }
    printf("Size␣of␣stuff:␣%i\n", size);
    printf("elements␣in␣peak:␣%i\n", i);

}






/******* Mainly for testing purposes *********/
void print_list(void)
{
```

```c
    struct PEAK *ptr = head;

    printf("\n --------Printing list Start-------- \n");
    while(ptr != NULL)
    {
        printf("\ntime: %i, Value: %d \n",ptr->clock, ptr->value);
        ptr = ptr->next;
    }
    printf("\n --------Printing list End-------- \n");

    return;
}



void print_latest(int backwards)
/*
 * Prints the latest [backwards] number of peaks in list
 * Latest first
 */
{
    struct PEAK *ptr = head;

    int i;
    for (i=0; i<backwards; i++){
        if(NULL == ptr) {
            break;
        }
        printf("\ntime: %i, Value: %d, type: %i \n",ptr->clock, ptr->value, ptr->type);
        ptr = ptr->next;
    }
    return;
}
```

D.4 Sensor

```c
/*
 * Todo:
 *  Restricting speed to 250 requests pr second
 *
 *
 *
 */
```

```

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

// returning INT16_MAX will terminate main loop.
int getNextData(FILE *file){

    signed int line = 0;
    if( file == NULL){
        printf("sensor.c::getNextData - couldnt open file. Terminating");
        return INT16_MAX;
    }

    fscanf(file,"%i",&line);

    if (feof(file)) {
        printf("sensor.c::getNextData - Reached EOF. Terminating\n");
        fclose(file);
        return INT16_MAX;
    }
    return line;
}
```

D.5 Header files

D.5.1 sensor.h

Below is the first of the header files, called sensor.h. This file contains the prototype for the sensor as well as the peak detection.

```
#ifndef ADD_H_GUARD
#define ADD_H_GUARD
#define INT16_MAX 1 << 16
int getNextData(FILE *file);
#endif
```

D.5.2 buffer.h

After this one, the next header file called filter.h comes - this file contains the prototypes of the filters as well as for the buffer.

```
#define BUFFERSIZE 33

typedef struct {
    int Data[BUFFERSIZE];
    unsigned int counter;
} BUFFER;

int pushData(BUFFER* buffer, int data);
void incrementCounter(BUFFER* buffer);
int getIndex(BUFFER* buffer, int offset);
int readData(BUFFER* buffer, int offset);

int lowPass2(BUFFER* inputBuffer, BUFFER* filtered);
int highPass2(BUFFER* inputBuffer, BUFFER* filtered);
int derivative2(BUFFER* inputBuffer, BUFFER* filtered);
int squaring2(BUFFER* inputBuffer, BUFFER* filtered);
int mwInt2(BUFFER* inputBuffer, BUFFER* filtered);
```

D.6 Tests

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

void testing(int choice, char inputpath[], char comparepath
    []){

    FILE *inputfile = fopen ( inputpath, "r");
    FILE *comparefile = fopen ( comparepath, "r");

    BUFFER buff_source = {{0}, 0};
    BUFFER buff_filtered = {{0}, 0};

    int errorFound = 0;
    int lineNumber = 0;
    int ran = 0;


    int inputdata = getNextData(inputfile);
    int comparedata = getNextData(comparefile);



    while(inputdata != INT16_MAX && comparedata != INT16_MAX) {
        lineNumber++;
        pushData(&buff_source, inputdata);
        switch (choice) {
            case 1: lowPass2(&buff_source, &buff_filtered); break;
            case 2: highPass2(&buff_source, &buff_filtered); break;
            case 3: derivative2(&buff_source, &buff_filtered); break;
            case 4: squaring2(&buff_source, &buff_filtered); break;
            case 5: mwInt2(&buff_source, &buff_filtered); break;
        }


        int filterVal = readData(&buff_filtered, 0);
        if (filterVal != comparedata) {
            printf("Value mismatch on line %i. filtered: %i, comparedata: %i\n", lineNumber, filterVal, comparedata);
            errorFound++;
        }
        inputdata = getNextData(inputfile);
        comparedata = getNextData(comparefile);
        ran = 1;

    }

    if (!errorFound && ran) printf("No errors found. You're clear for takeoff\nComparing input: %s with file: %s\n\n", inputpath, comparepath);
    else if (!ran) printf("Didn't run. No conclusion\n\n");
    else printf("ERROR: %i value-mismatch found\nComparing input: %s with file: %s\n\n", errorFound, inputpath, comparepath);

    return;
}



```

```c

int main(){

    testing(1, "Datafiles/ECG.txt","Datafiles/x_low.txt");
    testing(2, "Datafiles/x_low.txt","Datafiles/x_high.txt"
        );
    testing(3, "Datafiles/x_high.txt","Datafiles/x_der.txt"
        );
    testing(4, "Datafiles/x_der.txt","Datafiles/x_sqr.txt")
        ;
    testing(5, "Datafiles/x_sqr.txt","Datafiles/
        x_mwi_div_after.txt");

    return 0;
}
```