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 has been created that can determine a persons heartbeat and give warnings when either the intensity or the heartrate falls 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. 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 (from now on simply called ECG) scanner.
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 will be run 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
- 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:

- The programming language used for this is C

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?

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 single datapoints while needing to handle both the current data as well is the biggest challenge when it comes to data acquisition.

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 use previous samples from their own filtered output. 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

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 was 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, these 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 native to the file (for instance, the filter.c would contain filters only). The files used are: filter.c, buffer.c, main.c, RRhandling.c, sensor.c, peakDetect.c.

Furthermore, it was decided that the data from each filter, as well as the raw data, would each be stored in an array of up to 50 data points - having more than this would be a waste of memory, and having less than this presented the risk of not having enough data points for the filters.

To load the actual data into the arrays, a buffer was created in combination with a loop. The loop would run through the data-set, scanning in a point and then passing it to the buffer. The buffer would then store this data point in the correct array, and this array would then be passed on to the filters. Once the filters calculate the correct value, the value would be returned to the buffer that would store this in the correct place in the array.
The buffer will also keep track of where in the array the data is, and should it exceed the size of the array, it will return to the first place in the array (indexed to 0) and overwrite the data there.

4 Implementation

4.1 Core structure

Buffer struct, updating via pointers and global variables, and linked lists

4.2 Real-time data acquisition

As described, the program must acquire the data and calculate in real-time. To ensure this simulation, the program loads one data point, then passes it and the corresponding array to the one filter at a time before it finally passes it to the peak detection and subsequently, to the R-peak detection. Once all these calculations have been done, it loads in a new data point and restarts through a while loop.
This while-loop will run through the entire data-set, and will not stop before it reaches the end of the file (in the finished product, it will of course run until the battery runs dry).

Furthermore, it was found that in order to always keep a list of previous samples while maintaining easy access to their relation to the current data, structs and methods could be used to create a buffer - more specificly, this buffer was implemented using a struct containing an integer array as well as a counter. Using a list of simple method-functions to handle pushing of data as well as reading, a simple circular First In First Out (FIFO)-list was created.

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.

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 will also display the warning and the RR-LOW and RR-HIGH values if 5 RR-intervals has 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 the figure below, it is more than powerful enough to run the algorithm on the required time (250 data points per second), as it manages to process 10.000 data points on 0.02 miliseconds. Therefore, on the final processor, the power consumption can be brought down a great deal by reducing the clock speed of the CPU from the current 2GHz. Furthermore, as can be seen in the profiling, if the code was to be optimized speedwise, it would have to be done either in the function getIndex or the function mwInt2, though currently, it would be more important to

focus on bringing the power consumption down.

```
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 1: 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.

In regards to size....

5 Results

```
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 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.

5.1 Testresults of filters

5.2 Testresults of RPeakDetection

6 Discussion

7 Conclussion

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:
Jakob Welner, s124305 wrote:

B Sourcecode - introductionary exercises

B.1 ReadFromFile

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

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      printf("Hey, I'm reading a file!\n");
6
7      static const char filename[] = "ECG.txt";
8      FILE *file = fopen ( filename, "r");
9      int line, rVal;
10     int max;
11
12     rVal = fscanf(file, "%i", &line);
13     max = line;
14
15     while(rVal != EOF) {
16
17         rVal = fscanf(file, "%i", &line);
18         if (max < line) {
19             printf("Found a new larger number: %i \n", line
                   );
20             max=line;
21         }
22     }
23     printf("Finally, this is the largest numer: %i\n", max)
           ;
24     return 0;
```

```
25 }
```

B.2 HelloWorld

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

```
1  #include <stdio.h>
2
3  /*
4   * Created by Bastian Buch, s113432, and Jacob Gjerstrup,
        s113440
5   */
6
7  int main (void){
8    printf("Hello world!");
9    return 0;
10 }
```

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 Buffer

```
1  #include <stdio.h>
2  #include "buffer.h"
3
4  // Bufferindex
5  int pushData(BUFFER* buffer, int data)
6  {
7      //printf("Buffer::pushData: Adding stuff to the buffer
           !: %i\n", data);
8      (*buffer).Data[(*buffer).counter] = data;
```

```
 9       incrementCounter(buffer);
10
11       return 0;
12   }
13
14
15   // Getting values based on index offset
16   // compared to current buffer counter
17   int readData(BUFFER* buffer, int offset)
18   {
19       int index = getIndex(buffer, offset);
20
21       //printf("Buffer::readData - offset: %i\n", offset);
22       //printf("  Index: %i\n", index);
23       //printf("  Value - Buffer[index]: %i\n", buffer[index
             ]);
24
25       return (*buffer).Data[index];
26   }
27
28
29   void incrementCounter(BUFFER* buffer)
30   {
31       (*buffer).counter++;
32       if ((*buffer).counter >= BUFFERSIZE) {
33           //printf("Buffer::IncrementCounter - Buffer reached
                 maximum. Looping\n");
34           (*buffer).counter -= BUFFERSIZE;
35       }
36   }
37
38
39   int getIndex(BUFFER *buffer, int offset) {
40       int index = (*buffer).counter - offset -1;
41       if (index < 0){
42           index = index+BUFFERSIZE;
43       }
44
45       return index;
46   }
```

C.2 Filters

```
1  #include <stdlib.h>
2  #include <stdio.h>
```

```
3  #include "buffer.h"
4
5
6
7  int lowPass2(BUFFER* inputBuffer, BUFFER* filtered) {
8      /*
9       *  GroupDelay: 25 ms
10      */
11
12     // Retrieving values
13     int x = readData(inputBuffer, 0);
14     int x6 = readData(inputBuffer, 6);
15     int x12 = readData(inputBuffer, 12);
16     int y1 = readData(filtered, 0);
17     int y2 = readData(filtered, 1);
18
19     // Filter math
20     int y = (2*y1-y2) + ((x - 2*x6 + x12) / 32);
21
22     // pushing data back to buffer object
23     pushData(filtered, y);
24
25     return 0;
26 }
27
28
29 int highPass2(BUFFER* inputBuffer, BUFFER* filtered) {
30     /*
31      *  GroupDelay: 80 ms
32      */
33
34     // Retrieving values
35     int x = readData(inputBuffer, 0);
36     int x16 = readData(inputBuffer, 16);
37     int x17 = readData(inputBuffer, 17);
38     int x32 = readData(inputBuffer, 32);
39     int y1 = readData(filtered, 0);
40
41     // Filter math
42     int y = y1-(x/32)+x16-x17+(x32/32);
43
44     // Pushing data back to buffer object
45     pushData(filtered, y);
46
```

```
47      return 0;
48  }
49
50
51  int derivative2(BUFFER* inputBuffer, BUFFER* filtered) {
52      /*
53       *  GroupDelay: 10 ms
54       */
55
56      // Retrieving values
57      int x = readData(inputBuffer, 0);
58      int x1 = readData(inputBuffer, 1);
59      int x3 = readData(inputBuffer, 3);
60      int x4 = readData(inputBuffer, 4);
61
62      // Filter math
63      int y = (2*x+x1-x3-2*x4) / 8;
64
65      // Pushing data back to buffer object
66      pushData(filtered, y);
67
68      return 0;
69  }
70
71
72  int squaring2(BUFFER* inputBuffer, BUFFER* filtered) {
73      /*
74       *  GroupDelay: 0 ms
75       */
76
77      // Retrieving values
78      int x = readData(inputBuffer, 0);
79
80      // Filter math
81      int y = x*x;
82
83      // Pushing data back to buffer object
84      pushData(filtered, y);
85
86      return 0;
87  }
88
89
90  int mwInt2(BUFFER* inputBuffer, BUFFER* filtered) {
```

```
91      /*
92       *  GroupDelay: 72.5 ms
93       */
94
95      int N = 30;
96
97      // Dynamic Retrieving of values
98
99      int i = 0;
100     int sum = 0;
101
102     for (i = N; i >= 0; --i) {
103         sum += readData(inputBuffer, i);
104     }
105
106     // Filter math
107     int y = sum / N;
108
109     // Pushing data back to buffer object
110     pushData(filtered, y);
111
112     return 0;
113 }
```

C.3 RPeakDetection

R-Peak detection consists of two files - peak detection and RR Handling.
Peak detection:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include "sensor.h"
5
6
7  int isPeak(int dataPointOne, int dataPointTwo, int
       dataPointThree){
8      if (dataPointOne < dataPointTwo && dataPointTwo >
           dataPointThree){
9          return 1;
10         }
11     return 0;
12 }
13
14 void searchForPeaks(int dataset[]){
```

```
15      int i=0, j=0, datasetLength=0;
16          int data[10000]={0}; //replace with actual data
               structure for peak storing
17          datasetLength = sizeof(dataset)/sizeof(int);
18          for (i=1; i<datasetLength; i++){
19                  if(isPeak(dataset[i-1], dataset[i], dataset
                       [i+1])){
20                          data[j]=dataset[i];
21                          j++;
22                  }
23          }
24  }
```

RR Handling:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "buffer.h"
4
5  struct PEAK {
6      int atIndex;          // If older than buffer size cannot
           be reevaluated
7      int type;             // 0 for any peak, 1 for R-peak
8      int value;            // speaks for itself
9      struct PEAK *next;
10 };
11
12 struct PEAK *head = NULL;
13 struct PEAK *curr = NULL;
14
15
16 int RR_AVERAGE1 = 0;
17 int RR_AVERAGE2 = 0;
18 int RR_LOW = 0;
19 int RR_HIGH = 0;
20 int RR_MISS = 0;
21
22 int SPKF = 0;
23 int NPKF = 0;
24 int THRESHOLD1 = 0;
25 int THRESHOLD2 = 0;
26
27
28 // Initiating first element of linked list
29 struct PEAK* create_peakList(int index, int val) {
```

```
30      struct PEAK *ptr = (struct PEAK*)malloc(sizeof(struct
            PEAK));
31      if(NULL == ptr) {
32          return NULL;
33      }
34      ptr->atIndex = index;
35      ptr->type = 0;
36      ptr->value = val;
37
38      ptr->next = NULL;
39
40      head = curr = ptr;
41      return ptr;
42  }
43
44
45  // Adding node to beginning of list
46  struct PEAK* add_to_list(int index, int val) {
47
48      if(NULL == head) {
49          return (create_peakList(index, val));
50      }
51
52      struct PEAK *ptr = (struct PEAK*)malloc(sizeof(struct
            PEAK));
53
54
55      ptr->value = val;
56      ptr->atIndex = index;
57      ptr->next = NULL;
58
59      ptr->next = head;
60      head = ptr;
61
62      return ptr;
63  }
64
65
66  void print_latest(int backwards)
67  {
68      struct PEAK *ptr = head;
69
70      int i;
71      for (i=0; i<backwards; i++){
```

```
72          if(NULL == ptr) {
73              break;
74          }
75          printf("\ntime: %i, Value: %d \n",ptr->atIndex, ptr
                ->value);
76          ptr = ptr->next;
77      }
78      return;
79  }
80
81  int RRfind(BUFFER* inputData, int runCount)
82  {
83      int x2 = readData(inputData, 2);
84      int x1 = readData(inputData, 1);
85      int x0 = readData(inputData, 0);
86
87      // Checks for peak
88      //printf("\nX0: %i\nX1: %i\nX2 %i\n\n", x0, x1, x2);
89      if (x0 < x1 && x1 > x2) {           // Could be expanded
            to 5 evalpoints allowing for multiple equal val
90          add_to_list(runCount-1, x1); // overflow vulnerable
91
92          printf("Printing latest 3");
93          print_latest(3);
94          return 1;
95      }
96      return 0;
97  }
98
99
100
101 void print_list(void)
102 {
103     struct PEAK *ptr = head;
104
105     printf("\n --------Printing list Start-------- \n");
106     while(ptr != NULL)
107     {
108         printf("\ntime: %i, Value: %d \n",ptr->atIndex, ptr
                ->value);
109         ptr = ptr->next;
110     }
111     printf("\n --------Printing list End-------- \n");
112
```

```
113     return;
114 }
115
116
117
118
119 int RRdetermine(void)
120 {
121     printf("Prove to me that you are indeed a peak!\n");
122     return 0;
123 }
124
125 int RRsearchback(void)
126 {
127     printf("No peaks in sight!.. I guess I should look
          closer\n");
128     return 0;
129 }
```

C.4 Sensor

```
1  /*
2   * Todo:
3   *  Restricting speed to 250 requests pr second
4   *
5   *
6   *
7   */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include "sensor.h"
12
13 // returning INT16_MAX will terminate main loop.
14 int getNextData(FILE *file){
15
16     signed int line = 0;
17     if( file == NULL){
18         printf("sensor.c::getNextData - couldnt open file.
              Terminating");
19         return INT16_MAX;
20     }
21
22     fscanf(file,"%i",&line);
```

```
23
24      if (feof(file)) {
25          printf("sensor.c::getNextData - Reached EOF. Terminating\n");
26          fclose(file);
27          return INT16_MAX;
28      }
29      return line;
30  }
```

C.5 Header files

C.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.

```
1  #ifndef ADD_H_GUARD
2  #define ADD_H_GUARD
3  #define INT16_MAX 1 << 16
4
5  int getNextData(FILE *file);
6
7  // remove us before handing in
8  void testLow();
9  void testHigh();
10 void testDerivative();
11 void testSquaring();
12 void testMWI();
13
14 void searchForPeaks(int dataset[]);
15 int isPeak(int dataPointOne, int dataPointTwo, int
       dataPointThree);
16 #endif
```

C.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.

```
1  #define BUFFERSIZE 50
2
3  typedef struct {
```

```
4       int Data[BUFFERSIZE];
5       unsigned int counter;
6   } BUFFER;
7
8
9   int pushData(BUFFER* buffer, int data);
10  void incrementCounter(BUFFER* buffer);
11  int getIndex(BUFFER* buffer, int offset);
12  int readData(BUFFER* buffer, int offset);
13
14  int lowPass2(BUFFER* inputBuffer, BUFFER* filtered);
15  int highPass2(BUFFER* inputBuffer, BUFFER* filtered);
16  int derivative2(BUFFER* inputBuffer, BUFFER* filtered);
17  int squaring2(BUFFER* inputBuffer, BUFFER* filtered);
18  int mwInt2(BUFFER* inputBuffer, BUFFER* filtered);
```

C.6 Tests

C.7 Tests of RPeakDetection

C.7.1 tests

C.7.2 Main function for test cases