Embedded Systems

02131

R-peak detection!

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, initially using simulated data gathered from an ECG scanner. The conclusion is that it is easily possible to do this, and can recommend the implementation of this algorithm. However, it is desirable to have as many calculations completed through hardware rather than software, as this optimizes the running speed by several magnitudes.

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 costumer and finally, an analyzation 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 processor will be designed to do this task.

1.1 Requirements

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

Functional requirements for the application:

- Correct data acquisition in simulated real-time
- Implementation of the 5 filters
- Implementation of the RPeakDetection
- Correct output of relevant 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 Theory

Before we designed the code we were to implement, there were several topics that must be specified first. They are:

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 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 is simulated and not a real patient. To ensure the capability of the program, and to ease the transition to hardware-coding, a requirement was that the whole dataset must not be loaded into one array - instead, it must be loaded one data-point at a time. This introduces a few challenges, the biggest being how to ensure that the entire dataset is processed, no matter the size.

2.2 Problem 2: Implementation of Filters

To use the QRS-algorithm, the dataset must first be run through 5 filters, each of which must be implemented. Of these filters, 4 of them requires both the current data point, but also previous data points. This creates two challenges - one being how to ensure that the program does not encounter data under- and overflow, and the other being how the data points are to be transferred from the program to the filters.

To the first of these challenges, the easiest way to do this would be to implement the filters with strict if/else sentences. Alternatively, when the index reaches the first negative value, one could add the size of the array and have the index moving backwards through the end of the array of the dataset currently used.
To the second of these challenges, there are two solutions: Either, two arrays, x and y, could be used. Here, x will hold the data would operate on and y will hold the data that has already been filtered. Alternatively, a struct could be created.

2.3 Problem 3: Implementation of RPeakDetection

Once all the filters has been implemented, the actual QRS-algorithm has to be implemented. This QRS-algorithm will be the one that determines what consitutes a heartbeat, and how to analyse this heartbeat. Furthermore, it will determine what is a peak and what is not, 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 two 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.

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).

Therefore, the combination of the while-loop that runs forever, and the continual loading and subsequent calculations 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

In order to detect the R-peaks in the filtered data, a few new challenged appeared. Firstly the filtered data was run through a simple local-maxima detection and saved in a new list. However, given that there is no way of knowing how many peaks you will eventually find, a list of flexible length was needed. Looking at the different options for implementing such a list the decision fell on linked lists through struct. In this way you create a struct that contains a 'next'-variable with a pointer to the same type of struct. For each new instance of the struct you then set the intern next-value to point at the previous struct, thus linking the 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.

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

5 Results

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.

```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;
}
```

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 Buffer

```
#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);

    //printf("Buffer::readData - offset: %i\n", offset);
    //printf("  Index: %i\n", index);
    //printf("  Value - Buffer[index]: %i\n", buffer[index
       ]);

    return (*buffer).Data[index];
}


void incrementCounter(BUFFER* buffer)
{
    (*buffer).counter++;
    if ((*buffer).counter >= BUFFERSIZE) {
        //printf("Buffer::IncrementCounter - Buffer reached
            maximum. Looping\n");
        (*buffer).counter -= BUFFERSIZE;
    }
}


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

    return index;
}
```

C.2 Filters

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

```c
#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;
}


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 = N; i >= 0; --i) {
        sum += readData(inputBuffer, i);
    }

    // Filter math
    int y = sum / N;

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

    return 0;
}
```

C.3 RPeakDetection

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

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


int isPeak(int dataPointOne, int dataPointTwo, int
   dataPointThree){
    if (dataPointOne < dataPointTwo && dataPointTwo >
       dataPointThree){
        return 1;
        }
    return 0;
}

void searchForPeaks(int dataset[]){
    int i=0, j=0, datasetLength=0;
        int data[10000]={0}; //replace with actual data
            structure for peak storing
        datasetLength = sizeof(dataset)/sizeof(int);
        for (i=1; i<datasetLength; i++){
                if(isPeak(dataset[i-1], dataset[i], dataset
                    [i+1])){
                        data[j]=dataset[i];
                        j++;
                }
        }
}
```

RR Handling:

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

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

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


int RR_AVERAGE1 = 0;
int RR_AVERAGE2 = 0;
int RR_LOW = 0;
int RR_HIGH = 0;
int RR_MISS = 0;

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


// Initiating first element of linked list
struct PEAK* create_peakList(int index, int val) {
```

```c
    struct PEAK *ptr = (struct PEAK*)malloc(sizeof(struct
        PEAK));
    if(NULL == ptr) {
        return NULL;
    }
    ptr->atIndex = index;
    ptr->type = 0;
    ptr->value = val;

    ptr->next = NULL;

    head = curr = ptr;
    return ptr;
}


// Adding node to beginning of list
struct PEAK* add_to_list(int index, int val) {

    if(NULL == head) {
        return (create_peakList(index, val));
    }

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


    ptr->value = val;
    ptr->atIndex = index;
    ptr->next = NULL;

    ptr->next = head;
    head = ptr;

    return ptr;
}


void print_latest(int backwards)
{
    struct PEAK *ptr = head;

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

int RRfind(BUFFER* inputData, int runCount)
{
    int x2 = readData(inputData, 2);
    int x1 = readData(inputData, 1);
    int x0 = readData(inputData, 0);

    // Checks for peak
    //printf("\nX0: %i\nX1: %i\nX2 %i\n\n", x0, x1, x2);
    if (x0 < x1 && x1 > x2) {           // Could be expanded
         to 5 evalpoints allowing for multiple equal val
        add_to_list(runCount-1, x1); // overflow vulnerable

        printf("Printing latest 3");
        print_latest(3);
        return 1;
    }
    return 0;
}



void print_list(void)
{
    struct PEAK *ptr = head;

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

    return;
}




int RRdetermine(void)
{
    printf("Prove to me that you are indeed a peak!\n");
    return 0;
}

int RRsearchback(void)
{
    printf("No peaks in sight!.. I guess I should look 
       closer\n");
    return 0;
}
```

C.4 Sensor

```
/*
 * 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;
}
```

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.

```
#ifndef ADD_H_GUARD
#define ADD_H_GUARD
#define INT16_MAX 1 << 16

int getNextData(FILE *file);

// remove us before handing in
void testLow();
void testHigh();
void testDerivative();
void testSquaring();
void testMWI();

void searchForPeaks(int dataset[]);
int isPeak(int dataPointOne, int dataPointTwo, int
    dataPointThree);
#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.

```
#define BUFFERSIZE 50

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);
```

C.6 Tests

C.7 Tests of RPeakDetection

C.7.1 tests

C.7.2 Main function for test cases