Heart Rate Monitor

Ubiquitous Computing Mini Project

Fabian Meyer Jens Gansloser

July 23, 2014
HTWG Konstanz

The aim of this project is to build a heart rate monitor (HRM) device. There are already a lot of devices commercially available which measure the heart rate. However, the internal functionality of these devices is not exposed to the user, so there cannot be made any statements about their precision and the quality of the results. Additionally, most devices need to be worn on the users chest or the user must place his finger on it. This document describes the principle and implementation of a heart rate monitor device, which is able to detect the heart rate with high precision and can be worn as a wrist band.

Contents

1 Introduction

1.1 State of the Art

It is difficult to find out how commercial heart rate measurement devices/oximeters work. The internal functionality is not available and it is therefore hard, to make any statements about their precision and performance. In general, less information about heart rate devices and its implementation are available. Additionally, the available devices are very expensive. Most semi-professional devices need to be worn at the chest or as a finger-clip. Both types are uncomfortable and are not optimal for longer measurements. Also it is not always possible, to adapt these device for other projects. If the heart rate data needs to processed further, an interface for getting the data is required. Most device offer no or only a hard accessible interface.

1.2 Motivation

The motivation for this project is, to build an own heart rate device from scratch. The used technology should be open source and well known. It is important, to only use the minimal required hardware/software. No full-blown raspberry pie/linux system should be used. All requirements to the project are summarized in the following list:

- Capable of detecting the heart rate with high precision
- Low performance requirements
- Less hardware requirements (no special hardware)
- Cheap hardware
- Use only open source software
- Hardware setup should be flexible to use (finger, wrist band, ...)
- Simplistic and understandable design
- Easy to use in further projects
- Easy expandable

2 Principle of Operation

There are several ways to measure the heart rate. For this project, a technique is required which uses small and cheap hardware and can be used mobile. Also the performance requirements should be low, to allow an implementation with small energy-saving micro controllers. This chapter describes the method used for HRM to get the heart rate.

2.1 Heart Rate

A method to measure the heart rate is the photoplethysmogram (PPG) technique. This method measures the change of the blood volume through the absorption or reflection of light. A light emitting diode (LED) shines through a thin amount of tissue (e.g. fingertip, earlobe). The wavelength of the light should be in near infra-red area. On the other side a photo-diode registers the intensity of light that traversed the tissue. Since blood changes its volume with each heart beat, more or less light of the LED gets absorbed by it. If the heart pushes blood through the vessels, more blood flows between the LED and photo-diode (it has a higher volume) and more light is absorbed by it. That means less light is registered by the photo-diode. If blood flows from the vessels to the heart, less light is absorbed by it (the volume declines) and the photo-diode detects more light. As a result the registered intensity of light changes continuously with the pulse. By measuring the time between two intensity peaks the current pulse can be estimated. The setting for the pulse rate monitor is displayed in figure 2.1.

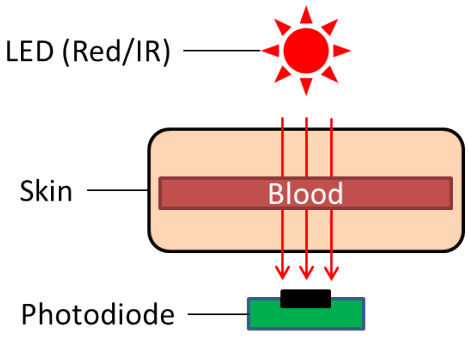

Figure 2.1: Pulse Rate Monitor Setting - Light through skin

An alternative way to measure the heart rate with the same technique is to detect the reflected light. The LED and the photo-diode can be placed on the same side. This setting is displayed in figure 2.2

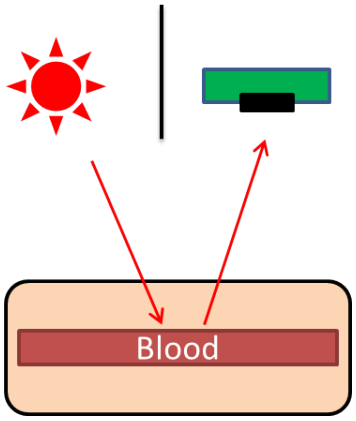

Figure 2.2: Pulse Rate Monitor Setting - Light reflected

Based on these two techniques, the application offers high flexibility to the kind of measurement. The heart rate can be measured through the fingertip, at the ear or at other places. For the prototype, the measurement principle with the fingertip is used.

2.2 Oxygen Saturation

The oxygen saturation of the blood can be measured by using 2 light emitting diodes. One LED emits light with a wavelength of 660nm (red light), the other emits light with a wavelength of 940nm (infra-red light). The absorption of light by blood changes corresponding to its oxygen saturation. Oxygenated blood absorbs more infra-red light and less red light. With de-oxygenated blood it is the other way around [1]. The LEDs blink alternating and a photo-diode is used to measure the light intensity after the light traversed the tissue. These measurements in combination with the Lambert-Beer-Law are used to calculate the oxygen saturation.

3 Environment

This chapter describes the hardware and the software used and implemented for the project. Figure 3.1 shows the different hardware parts and how they interact with each other. Note that the displayed setting is different from the actual used one in the prototype due to wrong hardware orders. The prototype setting is explained in chapter 4.

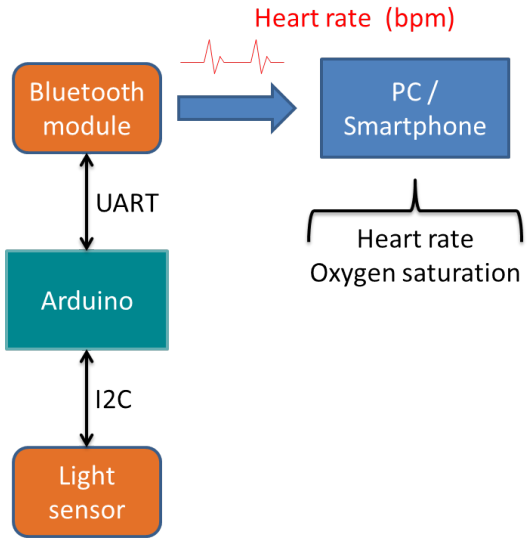

Figure 3.1: Project setup

3.1 Hardware

3.1.1 Light Intensity Sensor

To measure light, the 16bit TSL2561 Luminosity Sensor breakout board from Adafruit is used. It provides the TSL2561 Light-To-Digital Converter which is able to sense full spectrum and IR light with a very high sensitivity (see figure 3.2). The sensor contains a broadband photo-diode (visible and infra-red) and a infra-red photo-diode. Two ADCs convert the analog data to digital data, that can be read via the I^2C bus. The light sensor can be configured with different gain, which changes its sensitivity to light. This is required if the sensor is used in areas with bright or low environment light. A second option to configure is the integration time (13ms, 101ms, 402ms). This configures the resolution of the device, so that the sensor has more time to take samples. With a

integration time of 402ms, the sensor has the complete 16bit resolution. The output of the sensor can be used to calculate the measured SI-Unit lux, which indicates the illuminance [2]. Figure 3.2 shows the light spectrum of the two photo-diodes on the board.

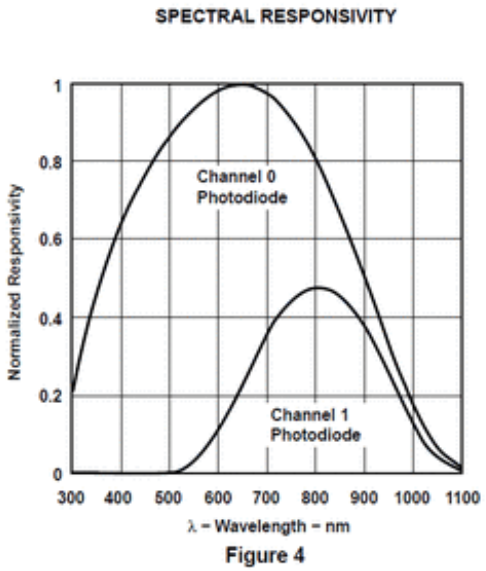

Figure 3.2: TSL-2561 Spectrum

3.1.2 Microcontroller

The data of the sensor is further computed by an Arduino [3]. The Arduino receives the data from the light sensor via the I^2C bus system. Different Arduinos can be used. The only requirements to the Microcontroller are I^2C and UART (for Bluetooth/serial) interfaces to communicate with the light sensor and the PC/Smartphone.

3.1.3 Bluetooth Module

The Bluefruit EZ-Link is a serial link bluetooth module. That means, the Arduino can communicate via UART with the bluetooth module, which handles the wireless transmission to the computer/smartphone. On the computer side, the module (if it was paired successfully) is recognized as an serial port. See also [4].

3.2 Libraries

To create the heart rate monitor, several external libraries were used. This section lists all used external libraries and explains its usage.

3.2.1 FFTW

FFTW is a C library, which offers functions to calculate the discrete Fourier Transform (DFT). It supports the calculation with multiple dimensions, complex or real data and different input sizes. HRM uses FFTW to compute the complex DFT of the light sensors signal and determine the heart rate.

3.2.2 Qt

Qt is a cross-platform C++ library for creating Graphical User Interfaces (GUIs). Additionally, it supports own container classes, networking, database access and a lot of more. For the HRM, Qt is used to create the GUI.

3.2.3 Qwt

Qwt is an add-on for Qt which is able to create 2D and 3D plots. Its advantages are the large amount of features and the high performance. This makes it suitable for technical applications, which need to display a lot of data. Qwt is used to plot the light sensors data and its Fourier Transformed.

3.2.4 QtSerialPort

QtSerialPort is used to access the serial port from the PC. It is an add-on module for Qt4 and Qt5. It allows fast and easy writing and reading to/from the serial ports. Because Qt is used for the Graphical User Interface (GUI), this is the optimal solution for serial port access.

3.2.5 Light Intensity Sensor Driver

To get data from the light sensor, the Adafruit TSL2561 and Adafruit Unified Sensor Driver libraries are used. These libraries do the I^2C communication and the digital to lux calculation. Also they provide functions to set the gain and integration time [5] [6].

3.3 Heart Rate Monitor Software

This section shows all implemented modules which are part of HRM. The implementation details are outlined in chapter 6. The software for the PC-part is implemented in C++. CMake is used as build-tool. The Arduino part is developed with the Arduino IDE. The application is cross-platform capable and supports Qt4 and Qt5. Compilation was tested on Linux and Windows.

3.3.1 Graphical User Interface

The GUI is mainly used for debugging and optimization. The GUIs features are shown in the following list:

- Displays the received data from the Arduino (via serial port)
- Displays the light sensors settings
- Allows setting of the light sensors sample rate
- Displays several diagrams (sensor values, frequency spectrum, ...)
- Displays the signal processing parameter

3.3.2 Serial Interface

The class `Serial` is used to control the access to the serial port. Qts slot and signal mechanism is used, to notify other classes when data is received. The class supports sending and receiving data.

3.3.3 Signal Processing Module

This is the main part of HRM. It is responsible for detecting the heart rate. The class `FFT` executes all signal processing steps explained in chapter 4.3.

3.3.4 Arduino Module

The Arduino software is executed on the Arduino board. It controls the light sensor and sends its data via UART to the PC. It uses the extended Adafruit driver to allow manual timing (using own sample rates). Additionally to that, it is able to parse incoming serial data: The light sensors settings can be queried, also a command to set the sample rate is available.

3.3.5 Extended Light Intensity Sensor Driver

The TSL2561 driver was extended to allow manual timing. Manual timing is required to allow custom sample rates. The standard driver allows only pre-configured sample rates, although the TSL2561 controller does support manual timing. Because the signal processing requires configurable sample rates, the driver was extended to allow this feature.

3.3.6 Overview

Figure 3.3 summarizes the previously outlined software.

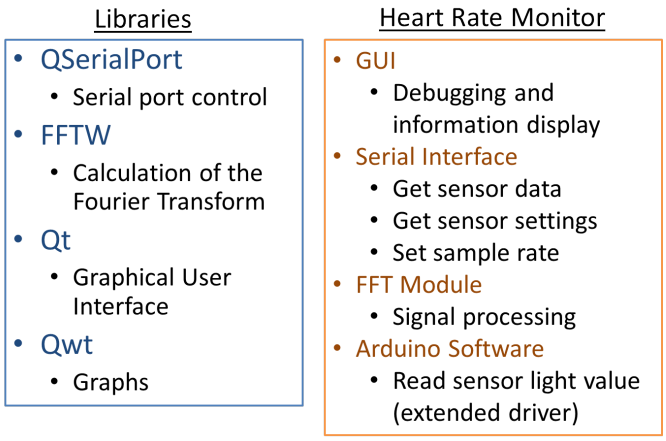

Figure 3.3: HRM Software

4 Heart Rate Monitor

4.1 Solution Approach

Figure 4.1 shows the output signal from the light sensor. The diagram shows the different luminosity values over time (= heart rate) of a patient. It is measured by placing the finger between the light sensor and the LED.

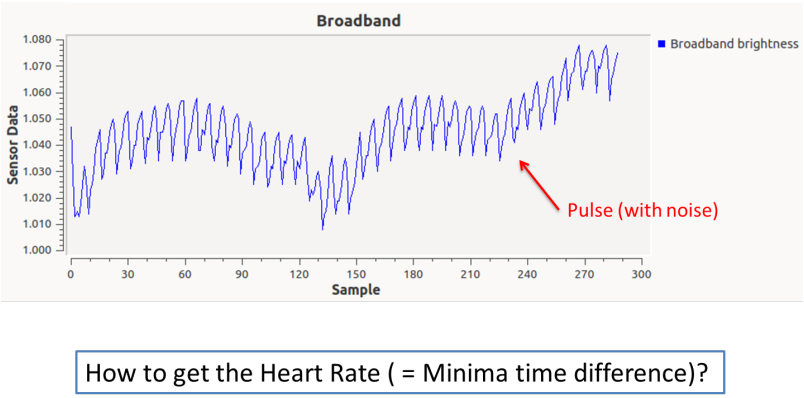

Figure 4.1: Raw light sensor data

To get the exact heart rate, the time between two minima (or maxima) of the discrete sensor values need to be determined. However, it is not trivial to determine the minima, because of the noise in the signals data. The noise comes from different environment settings, the difference in blood volume change of each person and how the finger is placed onto the sensor. These parameter result in a change in the data's y-part.

4.1.1 First Approach

A first naive approach is to determine the grade of each two sample points in the input data and identify if the curve is declining or rising. With this knowledge, the minima could be identified. However, this approach is not optimal. It is impossible to separate different minima in the data and find the correct ones. Due to the noise local minima could be detected which are not required. Introducing a static limit is unsuitable too, because in the difference of the y-axis data in each sample. To conclude, this approach is not suitable for the problem.

4.1.2 Second Approach

As one can see in figure 4.1, the heart rate is represented by a periodic up and down of the input values. These up and down values are distorted by some noise. The time difference between two minimas can also be expressed as the frequency of these periodic curve. Viewed from a signal processing view, these up and downs have the highest contribution of frequencies to the input signal. To get the heart rate, the frequency which contributes most to the input signal (= heart rate) has to be determined. This is a perfect application field for the Fourier Transform. The Fourier Transform is a mathematical method to analyse an input signal and determine the different frequencies which contribute how much to this signal. A Fourier Transform converts an input signal in the time domain to an output signal in the frequency domain (the frequency spectrum). The output signal shows, how much each frequency contributes to the input signal. The basic idea is, to transform the input signal via the Fourier Transform and then determine the signal which contributes most. This means determining the peak of the output frequency signal. Picture 4.2 shows the basic operation principle.

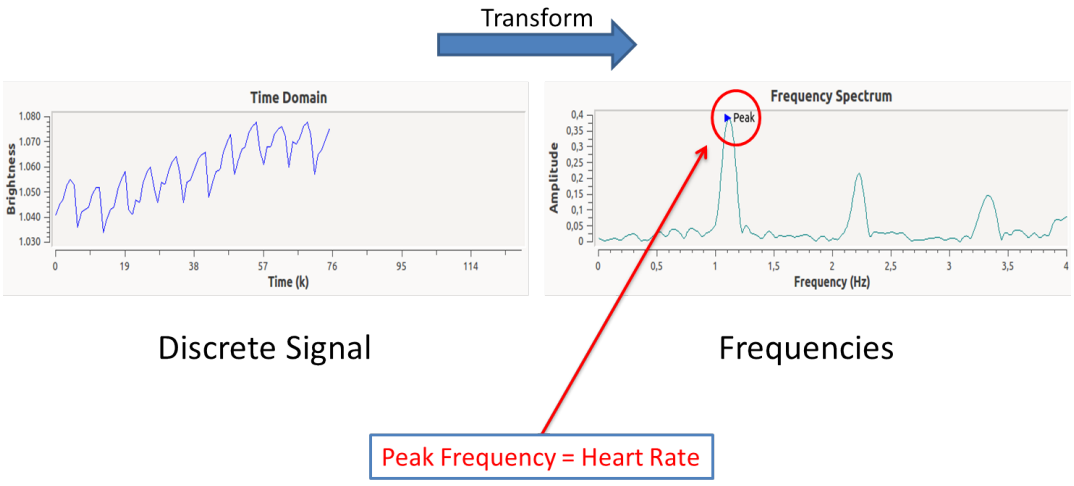

Figure 4.2: Basic Operation principle

4.2 Data Flow

To implement HRM, at first a data flow design has to be created. Because the implementation of the signal processing is not trivial and the debugging capabilities on the Arduino are limited, the first prototype is created on a PC. The data flow between Arduino and light sensor via I^2C can be separated in a control and application part. The control part contains the configuration of the sensor (setting configuration from Arduino to sensor). The application part contains the luminosity values (reading from sensor to Arduino). Like the I^2C communication, the UART communication can be separated in control and application flow, too. For the application part, the Arduino sends the

luminosity values to the PC. The control part consist of setting the sample rate of the sensors data or querying for the sensors settings from the PC. The PC collects the sensors values. When enough samples are collected, it executes the signal processing to identify the heart rate. Figure 4.3 shows the hardware set-up used in the prototype. At the moment, HRM can detect the heart rate. It can be easily extended to measure the oxygen saturation.

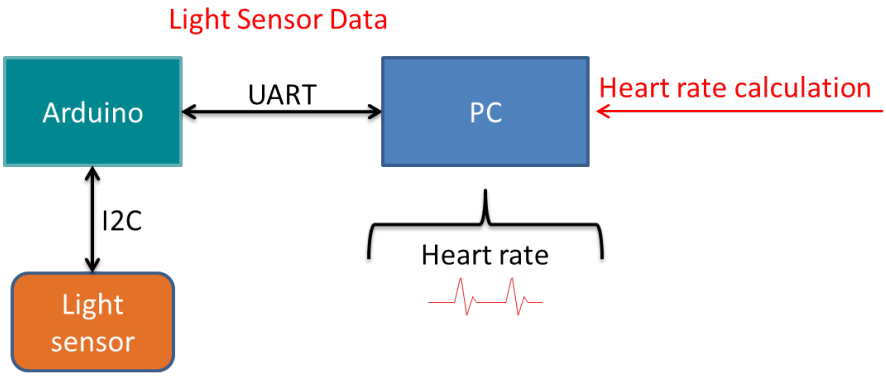

Figure 4.3: Project Setup

4.3 Signal Processing

This section describes the signal processing of the light sensors input values and how to identify based on this data the heart rate. The basic functionality and usage of the signal processing steps are explained. However, due to its complexity and its large scope, the derivation of the mathematical principles are not covered here.

4.3.1 Fourier Transformation

There are a lot of different kinds of Fourier Transforms (FT) for different application fields. First, the correct kind has to be determined.

Synthesis or Analysis

Fourier Transform can be used for analysis (forward FT) and synthesis (inverse FT). Because we want to get the frequency domain signal from the time domain signal, we use the forward transform (analysis).

Figure 4.4: Synthesis or Analysis

Type of FT

There are several different types of the FT, depending on the input signal. The following table explains the types and which FT to use.

Input signal	**FT type**
Continues, aperiodic	Fourier Transform
Continues, periodic	Fourier Series
Discrete, aperiodic	Discrete Time Fourier Transform
Discrete, periodic	Discrete Fourier Transform

Table 4.1: FT types

Because the input signal is an array of discrete values and it is finite, the discrete FT (DFT) is required. The input data is thought of it would be periodic, so the DFT instead of the DTFT is used (The DTFT is only used in theoretical signal analysis and uses infinite input data).

Complex or Real

The formula for the DFT is shown in 4.5. Complex numbers are used. The input are complex discrete values $x[n]$. The output are complex discrete values $X[k]$. N is the number of samples. The iterator k represents the frequencies. Although there is a DFT with real values too, the complex DFT is used. The reason for that is that using this kind of transform is simply more common.

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-j 2\pi kn/N}$$

Figure 4.5: Complex DFT formula

An fast implementation of the DFT is the FFT. The FFTW library is used to compute the FFT of the sensors data. Also the FT is the basic principle to get the heart rate, additional steps need to be made to get a nice frequency spectrum. The following chapters describe the different steps in detail. Figure 4.6 shows the applied functions. The displayed frequency spectra in the following sections are all calculated with enabled zero padding. The result of disabling zero padding is shown in section 4.3.4.

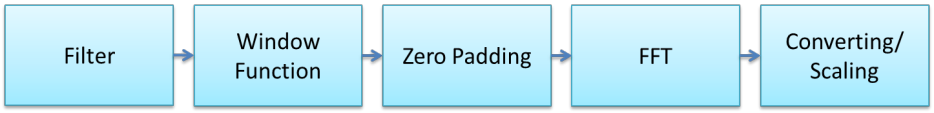

Figure 4.6: Signal Processing Steps

4.3.2 Filter

Without applying the steps mentioned above, the output signal looks like shown in Figure 4.7. There are a lot of high Frequencies and it is difficult to find the correct optimum (Figure 4.2, peak symbol). In the picture, the peak symbol points to the actual optimum. However, it is difficult to detect and not easy to find. The low frequencies are from the noise. To remove these unwanted frequencies, a bandpass filter is applied. To measure the heart rate, only heart rates from 40 - 230bpm are possible. This means, only frequencies from 0.7 - 3.9Hz need to be identified. All other frequencies in the input signal can be removed.

$$\begin{aligned} 40bpm &\approx 0.7Hz \\ 230bpm &\approx 3.9Hz \end{aligned} \tag{4.1}$$

The bandpass filter allows only desired frequencies (a frequency range) and removes the remaining. The filter can be applied to the input signal (convoluting with e.g. butterworth bandpass filter) or to the output signal (multiplication - cutting the unwanted frequencies). When applying the filter before the FFT, it needs some time until it is stabilized. The stabilization time has to be cut off from the signal. Else, this would result in distorted output data. However, because the second option is much more faster and allows to use a perfect filter, this option is used. Using a perfect filter (rectangular function) is much more easier. It is not required to design the filter coefficients. Additionally, multiplying does not requires as much performance as convolution.

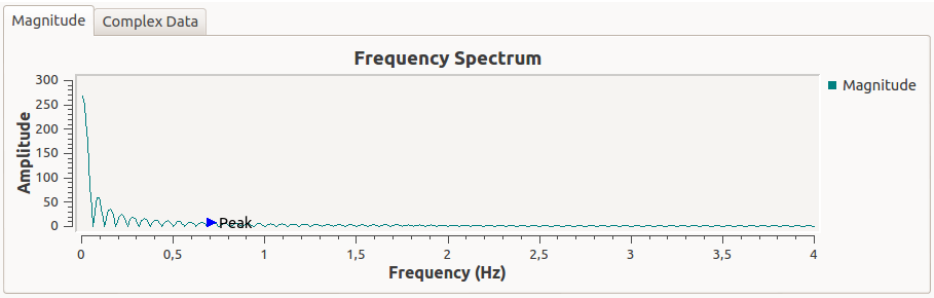

Figure 4.7: Only FFT

The frequency spectrum after applying the filter is shown in figure 4.8. The diagram does look much more better and the peaks are already good to detect.

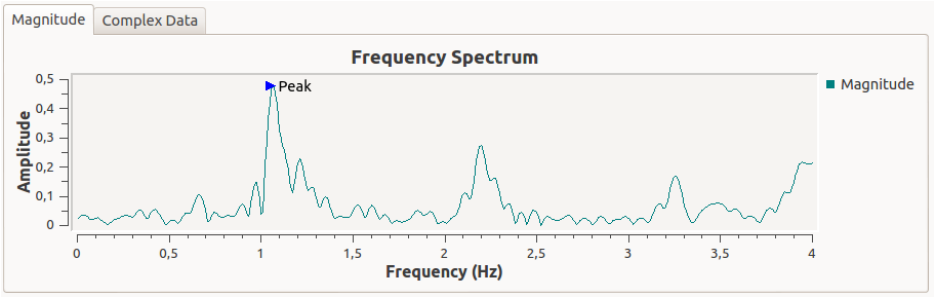

Figure 4.8: With filter

4.3.3 Window Function

Before executing the FFT, a window function has to be applied to reduce the leakage-effect. The leakage effect appears with continues and discrete FT. It happens by using only finite (time limited) data as input signal for the DFT.

Continues

The finite input signal can be viewed as a infinite signal multiplied with a rectangular function to limit it to the sampled interval. Multiplying in the time domain results in convolution in the frequency domain and therefore the output signal is smeared with the transformed rectangular function.

Discrete

When applying the DFT, it is assumed that the input signal is periodic by queueing multiple of these windows at it each other. However, because there are no consistent cuts at the start and end, the window sequences do not fit good to each other. To reduce this effect, a window function needs to be applied. This window function is multiplied with the input signal. It reduces the value at each side of the input data slowly downward to zero, the windows fit perfectly to each other (when they are added to a sequence). With infinite input signal, this effect would not appear. However, infinite long input signals are in the physical world not possible (only theoretical when using the DTFT). The only way to reduce the leakage effect when using the DFT is to synchronize the sample rate with the signals frequency. Here, the signals frequency has to be an integer multiple of the sample interval. This prevents inconsistent cuts between two sequences and allows a periodic continuance in the time domain. However, in most application fields this is not possible.

Used Window Function

There are different window functions to choose from. The parameters which characterize window functions are the main lobe, the sidelobs and the sidelob level. The used func-

tion for HRM is the Hamming-Window. The frequency spectrum with enabled window function is shown in figure 4.9. It is much more smoother and contains less aliasing.

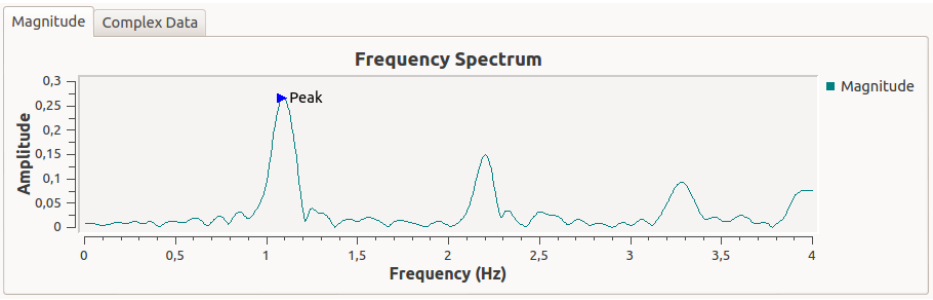

Figure 4.9: With everything

4.3.4 Zero Padding

Before applying the FFT, additional zeros are added to the input signal. This is a trick to interpolate the resulting frequency spectrum and make the possibility higher, that a real maximum is at an output frequency index. With this interpolating technique, the frequency resolution can be increased. Without zero padding, the frequency spectrum would look like displayed in figure 4.10. In this case, it is hard to determine the correct maxima as compared to figure 4.9, because they could be between to indices (due to the poor frequency resolution).

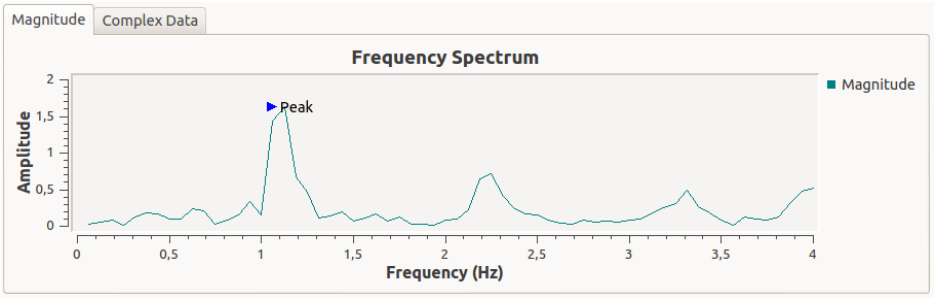

Figure 4.10: Without zero padding

4.3.5 FFT

After using the filter, window function and zero padding, the FFT can be applied to transform the data into the frequency domain. This is done by the FFTW library. The input data is the discrete light data array with N values. Because the discrete complex FT (DFT) works with complex numbers, the imaginary part of each input element is set to 0. The output of the FFT is also an array of N complex numbers.

4.3.6 Converting/Scaling

The resulting frequency domain values need to be scaled, to represent the correct amplitude. This is made by the following code snippet. The values are multiplied with 2, because only the positive frequencies are used, but the complex DFT calculates positive and negative frequencies. Additionally, the values are brought to a more human readable format. The original complex numbers (sin and cos) in rectangular coordinate are converted to polar form with magnitude and phase shift. To apply the scaling, at first the correct values to scale need to be determined: The output format of the FFT is displayed in the following figure 4.11. The first element of the array represents the DC offset (y-axis shift) and is not required. This is because this value changes based on the noise/environment light and is not needed to determine the heart rate. The next $[1, N/2]$ values represent the positive frequencies. The other values the negative frequencies. For this application, only the positive frequencies are used. So only the first half elements $[1, N/2]$ need our attention. Only the magnitude of this data is relevant, to calculate the heart rate. The phase shift is not of interest.

```
1 double scaledAmplReal = 2 * out[i][0] / N;
2 double scaledAmplImag = 2 * out[i][1] / N;
3 // To polar coordinates magnitude
4 double magnitude = (sqrt(pow(scaledAmplReal, 2) + pow(scaledAmplImag,
      2)));
```

Listing 4.1: FFT.c

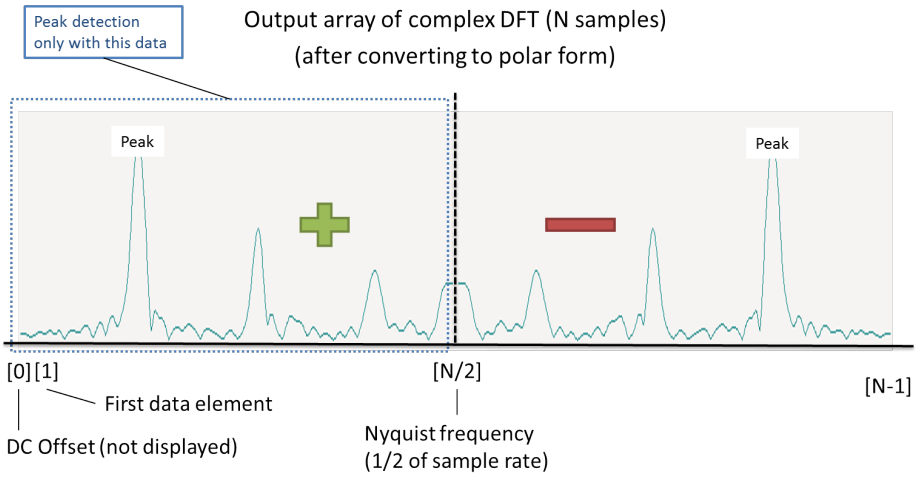

Figure 4.11: Complex DFT output format

Additionally to the values of the y-axis, the x-axis need to be brought in the correct form. These values can be displayed as array indices, as fractions of the sample rate or as frequency. The most human readable format is the frequency representation. To

calculate the available frequencies it is required to know the used sample rate. The frequency for a given array index x is calculated with the following formula.

$$f(x) = f_{sample} * (x/N_{total}) \tag{4.2}$$

N_{total} is the number of all samples including the zero padded ones. f_{sample} is the sample rate in Hz.

4.3.7 Peak Detection

The last step to do is to detect the peak. This is done by iterating through the magnitude values and find the value (x-value/frequency) with the highest amplitude. This x-value needs to be converted into beats per minute (bpm), the standard heart rate unit.

4.3.8 Parameter

Additionally to applying the different signal processing steps, the correct parameter need to be chosen. There are several requirements to the signal processing:

- Small number of samples used for FFT (else it will take long time until enough data is gathered)
- High frequency resolution (0.1bpm)
- Fast calculation (for implementing on smart-phone/uC/...)
- Nice output data in frequency domain which allows easy peak detection

All FFT parameter are saved in the structure `FFT_properties`. The most important parameter are explained in the following sections.

```
1  struct FFT_properties {
2      int numberOfSamples = 0;
3      int zeroPaddingSamples = 0;
4      int totalSamples = 0; // N (numberOfSamples + zeroPaddingSamples)
5      int outputSize = 0; // totalSamples / 2
6      int slidingWindow = 0;
7
8      double sampleInterval = 0.0; // delta x
9      double sampleRate = 0.0; // Hz
10
11     double segmentDuration = 0.0; // ms
12     double frequencyResolution = 0.0; // Hz
13     double frequencyResolutionWithZeroPadding = 0.0; // Hz
14 };
```

Sample Rate

To detect frequencies up to 4Hz, a minimal sample rate of 8Hz (according to the Shannon-Nyquist theorem) is required. This means at least a sample interval of 125ms is needed. The light sensor allows sample intervals up to 15ms.

Frequency Resolution

An optimal value for the number of samples are 128. It allows a good trade-off between frequency resolution and required sample time. However, the frequency resolution is not good enough (3.75bpm). To further improve the resolution, the following steps can be made:

- More samples (unsuitable, because requires more time)
- Zero Padding (interpolating by adding zeros to the end of the input data). However, its only a trick, because it adds not more information to the data.
- Higher sample rate. A higher sample rate results in more frequencies that can be detected. The filter can remove them.

For getting a higher frequency resolution, zero padding is applied as explained in section 4.3.4.

Sliding Window

To get fast new frequency spectra, the FFT is applied to the 128 samples every 5 samples. This is called segment duration/sliding window. It means, that every 5 samples, the FFT of the last 128 samples is calculated and should not be confused with the window function. The segment duration of 5 is trade off between fast results and required computation time. Else, the user would have wait 128 samples (= 16s) until he gets a new frequency spectrum and therefore the heart rate. To implement this, a special buffer (6.1.3) was created, which allows using different segment durations and fast data acquiring. Figure 4.12 shows the sliding window at three different times (t_1, t_2, t_3).

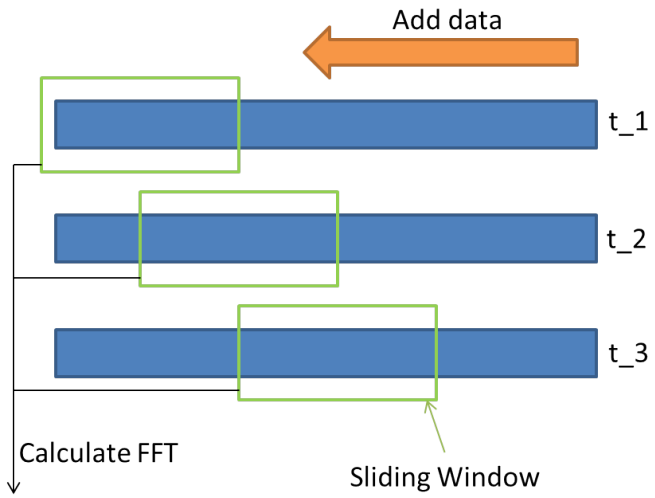

Figure 4.12: Sliding Window

4.4 Results

Figure 4.13 shows the final output signal.

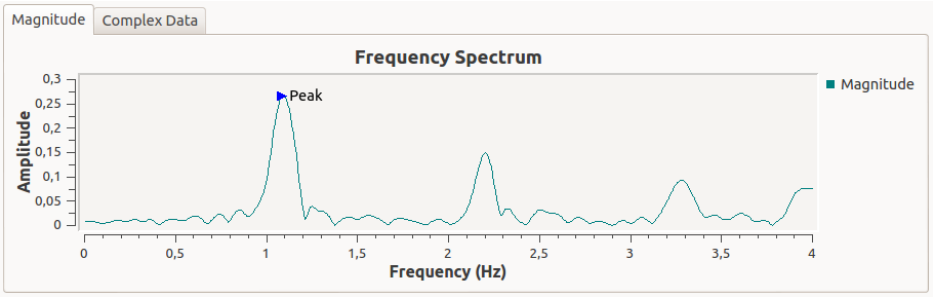

Figure 4.13: Final result

The different peaks are the multiples of the fundamental frequency - the harmonic frequencies. The fundamental frequency is at $x = 1.1$ (1. harmonic), the 2. harmonic is at $2 * x$, the 3. harmonic at $3 * x$, ...

5 Conclusion

5.1 Prototype and Results

The prototype is able to detect the heart rate with a very high precision (0.1bpm to 0.5bpm). Most commercial devices only have a resolution of 1bpm. Tests with commercial devices verified its functionality. Less and cheap hardware is required, to make a heart rate monitoring device. The Fourier Transform is an elegant way, to do the signal processing. It is quit robust against noise. Additionally, HRM is very flexible. The implementation is based on well known and easy to use technology. Also the hardware installation can be adapted to several use cases.

5.2 Improvements

Although HRM works very good, there can be still made some improvements. Based on this prototype, more projects can be realised, to improve the heart rate monitor:

- Porting to an Arduino
  - FFT library is available
  - Performance tests need to be made
- Using a smart-phone for signal processing
- Porting to a smaller uC.
- Adding a Bluetooth module (trivial)
- Add oxygen saturation measurement
  - Easy to implement with the existing knowledge
- Create a wrist band (with better environment light shielding)
- Brighter LED
- Better light sensor (new version is available which is more sensitive)
- Do additional tests with different mounting possiblities (reflection, hand wrist, ...)

Figure 5.1 shows a hardware setup with a Bluetooth module, to allow wireless heart rate detection. As shown in the picture, the heart rate should be calculated on the Arduino. Only the calculated heart rate is sent via Bluetooth to the user.

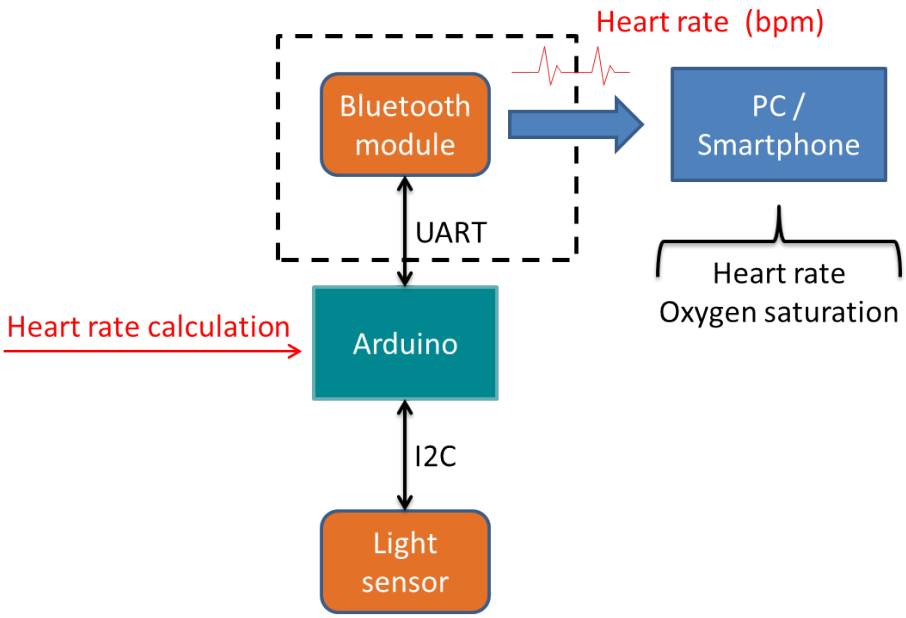

Figure 5.1: Project Setup

6 Implementation Details

This chapter describes the most important implementation details. For further information and API documentation, a doxygen file is available.

6.1 Software

6.1.1 Graphical User Interface

The GUI is used for displaying the different parameter and data. Additionally it calls the signal processing module. Using a GUI simplifies the debugging and optimization process. In this chapter, the usage of the signal processing module from the GUI is covered.

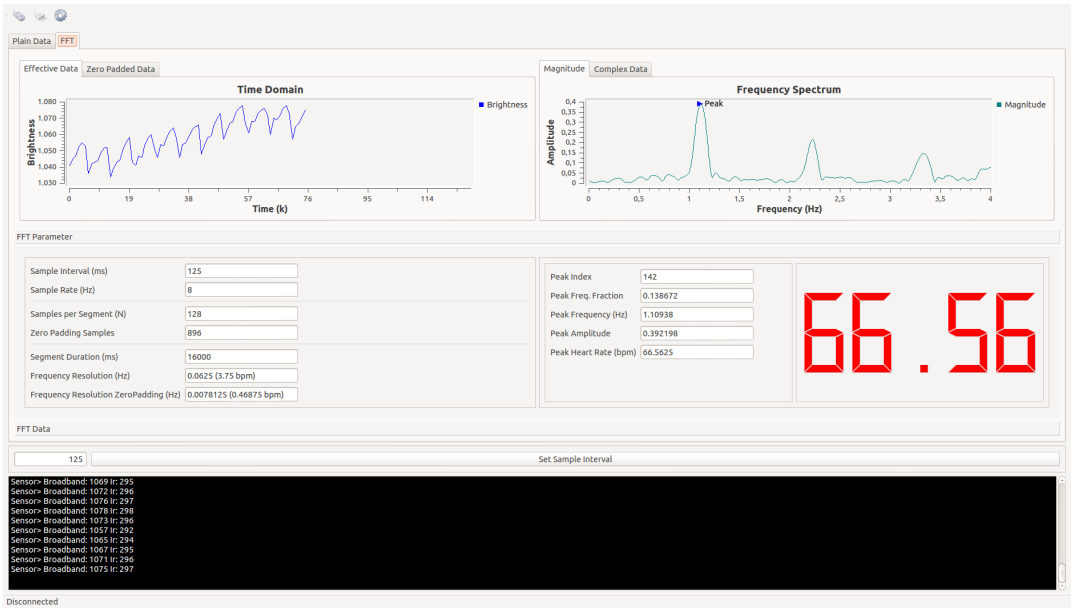

Figure 6.1: GUI

```
1 if (fft.addSample(data.broadband))
2         displayFrequencies();
```

Listing 6.1: MainWindow.cpp

`fft.addSample()` is called in the `receiveSensorData()` slot (when sensor data is received). When it returns non-zero, the DFT of the last sliding window was calculated. The Function `displayFrequencies()` is called:

```
1  double *magnitude = fft.getMagnitude();
2  double *real = fft.getRealPart();
3  double *imaginary = fft.getImaginaryPart();
4
5  FFT_properties properties = fft.getProperties();
6
7  // Max peak
8  int indexMax = fft.getPeak();
9  displayPeak(indexMax, magnitude[indexMax]);
10
11 // Display properties ...
12 // Display output data ...
```

Listing 6.2: MainWindow.cpp

In rows 1-3 the frequency spectrum's magnitude (polar coordinate) and real/imaginary values (rectangular coordinates - for debugging) are queried. After that, the current properties are saved in a `FFT_properties` structure. The peak index of the frequency spectrum is computed with `getPeak()`.

6.1.2 Serial Interface

The class `Serial.h` receives UART data from the Arduino and saves it in the structures `SensorData` and `SensorSettings` for easier usage. It parses the incoming commands and converts it to data structures which are easier to interpret. The following signals are available for receiving plain string data, the light sensors luminosity values or its settings.

```
1  signals:
2      void receiveLine(QString data);
3      void receiveSensorData(SensorData data);
4      void receiveSensorSettings(SensorSettings settings);
```

The sensor settings are saved in the following struct.

```
1  struct SensorSettings {
2      QString sensor;
3      QString id;
4      QString max; // Max luminosity value
5      QString min; // Min luminosity value
6      QString resolution;
7      QString sampleInterval;
8  };
```

The light sensor data (broadband and infra-red) are saved in the `SensorData` structure.

```
1  struct SensorData {
2          uint16_t broadband;
3          uint16_t ir;
4  };
```

Sending data to the Arduino is done with the `sendData(QString string)` function.

Serial Configuration

The following serial port configuration is used:

- Port Name: /dev/ttyACM0
- Baudrate: 9600
- Data Bits: 8
- Parity: No
- Stop Bit: 1
- Flow Control: No

6.1.3 FFTBuffer

The class `FFTBuffer.h` is a wrapper for FFTWs own buffer (`fftw_complex` array). Because the FFTW data buffer is only an array and does not allow more comfortable container operations this wrapper class was written. With this class, data can be added like it would be a queue. Optional, zero padding can be enabled. The following variables are used, to configure the size of the buffer:

```
1 int effectiveSize;
2 int zeroPadSize;
3 // effectiveSize + zeroPadSize = totalSize
4 int windowSize;
```

`effectiveSize` is the number of data samples the buffer can have as a maximum. After `effectiveSize` samples were added, the FFT of this data can be calculated. `zeroPadSize` defines how much additional elements are optionally used for zero padding. `windowSize` is the size of the sliding window explained in 4.3.8. For basic usage the function `add()` is used:

```
1 fftw_complex *add(double p_data);
```

This function adds the parameter `p_data` as real value to the data vector. The imaginary part is set to zero. It returns `nullptr` if the desired maximum capacity is not reached. If it returns non-zero, a pointer to the `fftw_complex` array with valid data is returned. Internally, the data is first copied to a vector container. After `effectiveSize` is reached, the data is copied to the `fftw_complex` array. After copying, zero padding is applied. If the sliding window is enabled, `(effectiveSize - windowSize)` elements are preserved in the data vector. `windowSize` new elements are required until `add()` returns the next array pointer. Figure 6.2 shows the internal work flow.

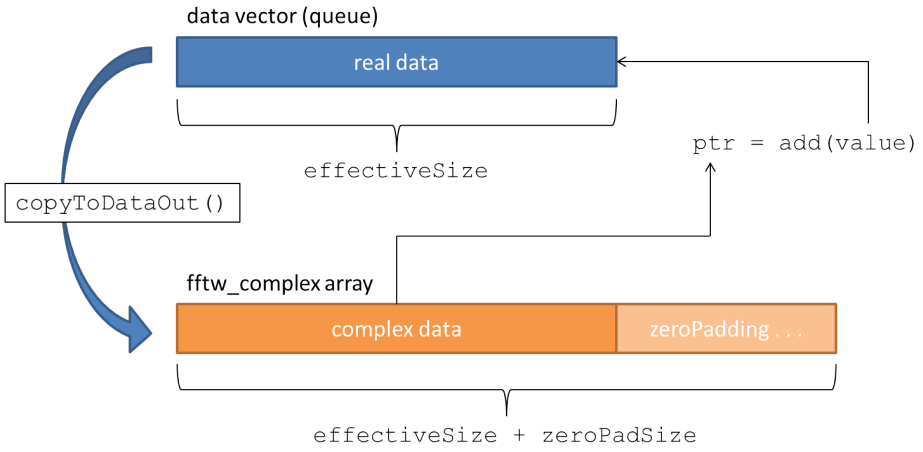

Figure 6.2: FFTBuffer

More functions are supported, to set or get specific elements or get the different sizes.

6.1.4 FFT

The class `FFT` is the main part of HRM and does the signal processing. When enough data was aquired to calculate the DFT, the following steps are executed:

```
1 // Functions for input time domain.
2 filter();
3 windowFunction();
4
5 fftw_execute(plan);
6
7 // Function for output frequency domain.
8 scaleAndConvert();
```

Listing 6.3: FFT.c

`filter()` convolutes the input signal with the butterworth bandpass filter and cuts the stabilization time. `windowFunction()` applies the Hamming-Window. After calcualting the DFT with `fftw_execute(plan)` the resulting frequency spectrum is correctly scaled (to get correct amplitude values) and converted to polar coordinates (`scaleAndConvert()`). As a result of these steps, the output data is available through the following pointers:

```
1 double *outMagnitude;
2 double *outReal;
3 double *outImaginary;
```

Listing 6.4: FFT.c

`outMagnitude` consists of the magnitude part from the converted polar coordinates, `outReal` and `outImaginary` are the real and imaginary parts of the output from the rectangular coordinates. All arrays have $N/2$ elements. The output data is converted in the following way:

```
1  // Complex value to magnitude
2  // (multiply with 2 because of the negative and positive frequencies)
3  double scaledAmplReal = 2 * out[i][0] / N;
4  double scaledAmplImag = 2 * out[i][1] / N;
5  // To polar coordinates magnitude
6  double magnitude = (sqrt(pow(scaledAmplReal, 2) + pow(scaledAmplImag,
       2)));
7
8  outReal[i-1] = scaledAmplReal;
9  outImaginary[i-1] = scaledAmplImag;
10 outMagnitude[i-1] = magnitude;
```

Listing 6.5: FFT.c

The array index of the peak value of the magnitude data to get the heart rate can be requested with the function `int getPeak()`. For calculating the FFT with the function `fftw_execute(plan)` with the FFTW library, at first a plan has to be created. This is done in the `FFT` class constructor. The most important parameter are the first three: Size of the buffer, input buffer and output buffer.

```
1  plan = fftw_plan_dft_1d(buffer->totalSize(), buffer->get(), out,
       FFTW_FORWARD, FFTW_MEASURE | FFTW_PRESERVE_INPUT);
```

Listing 6.6: FFT.c

6.1.5 Arduino Module

The Arduino module software is required to get the light sensors values and send them via UART to the PC. Because this is its main functionality, the software is very small and does not require a lot of performance. The main loop is displayed in the following code snippet.

```
1  void loop()
2  {
3    luminosityManualTiming();
4
5    readData();
6  }
```

The function `readData()` reads incoming UART data and interprets it (for getting sensor settings or setting the sample interval/rate). The function `luminosityManualTiming()` reads the sensor values with the extended manual timing driver functionality:

```
1  void luminosityManualTiming()
2  {
3    uint16_t broadband, ir;
4
5    tsl.beginIntegrationCycle();
6    delay(sampleInterval);
7    tsl.stopIntegrationCycle(&broadband, &ir);
8
9    printLuminosity(broadband, ir);
10 }
```

The sample rate/interval is saved in the variable `sampleInterval`, which can be set from the PC.

6.1.6 Light Intensity Sensor

This section describes the usage of the extended light sensor library. For using predefined integration times, the following code can be used:

```
1 tsl.enableAutoRange(true);
2 tsl.setIntegrationTime(TSL2561_INTEGRATIONTIME_13MS);
3
4 uint16_t broadband, ir;
5 tsl.getLuminosity(&broadband, &ir);
```

Listing 6.7: Adafruit driver

Line 1 sets the gain value to auto range. That means it adapts automatically to the sensors environment. On line 2 the integration time is set to 13ms (low resolution, fast processing). Line 5 queries for the luminosity values of the broadband and ir photodiode. For the use in this project, the sensor driver had to be slightly modified to allow manual integration times. The original driver allows only to set three different static integration times (13ms, 101ms, 402ms). Specific integration time are required to allow the desired sample frequencies. The extended driver provides functions to start and stop the integration of the light values, shown in the following code extract.

```
1 /* Manual timing control */
2 void beginIntegrationCycle();
3 void stopIntegrationCycle(uint16_t *broadband, uint16_t *ir);
```

Listing 6.8: Adafruit_TSL2561_U.h

6.2 Wiring

Figure 6.3 shows the wired luminosity sensor and the red LED. Only the actually parts required for the prototype are displayed. A picture of LED and sensor is displayed in figure 6.4. The complete prototype installation is shown in 6.5.

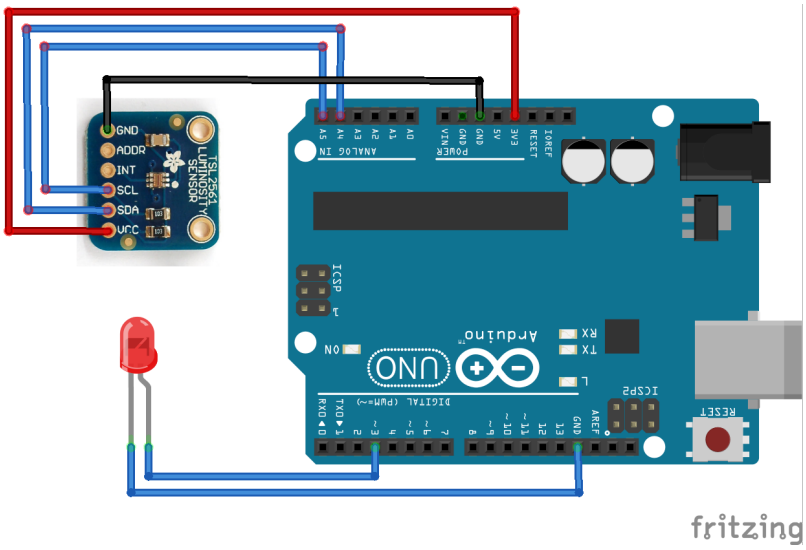

Figure 6.3: Wiring

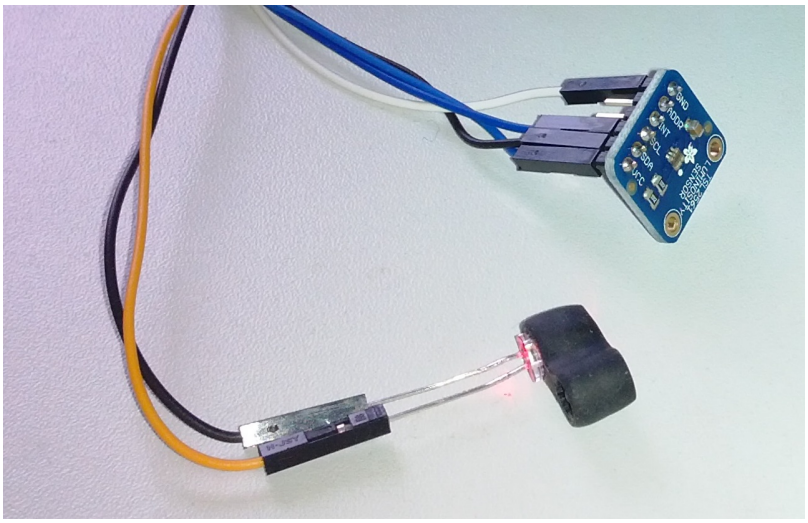

Figure 6.4: LED and sensor

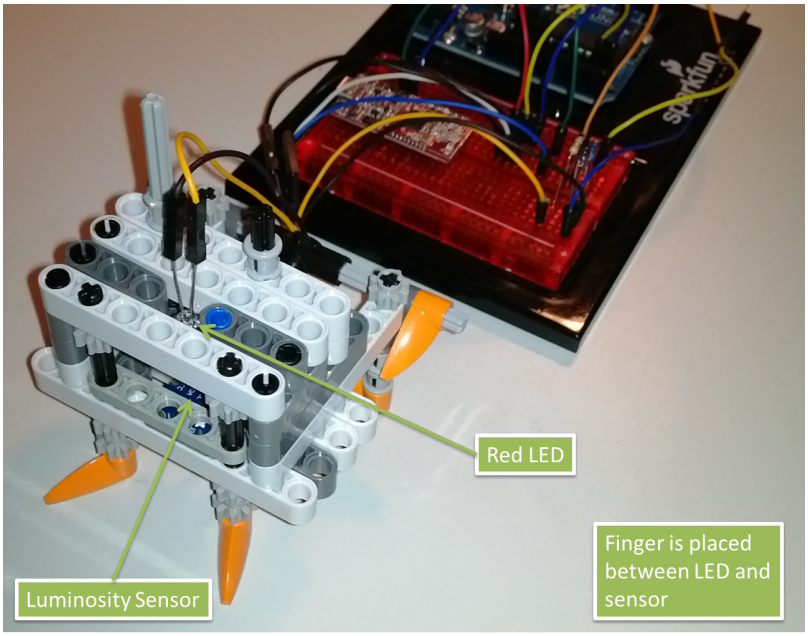

Figure 6.5: Prototype

6.3 Source Code

The source code and the documentation are available at Github (https://github.com/UbiquitousComputingSS14). It is licensed under the GPLv3 license.

Bibliography

[1] Pulse Oximetry, Dr. Chloe Borton, http://www.patient.co.uk/doctor/pulse-oximetry, 30.05.2014

[2] TSL2561, https://learn.adafruit.com/tsl2561/overview, 30.05.2014

[3] Arduino Board, http://www.adafruit.com/product/50, 31.05.2014

[4] Bluefruit EZ-Link, https://learn.adafruit.com/introducing-bluefruit-ez-link/overview, 31.05.2014

[5] TSL 2561 Library, https://github.com/adafruit/Adafruit_TSL2561, 31.05.2014

[6] Adafruit Unified Sensor Driver library, https://github.com/adafruit/Adafruit_Sensor, 31.05.2014

[7] Steven W. Smith, Ph.D. ,http://www.dspguide.com/, 21.07.2014

[8] Isaac Amidror, Springer, Mastering the Discrete Fourier Transform in One, Two or Several Dimensions - Pitfalls and Artifacts, 21.07.2014

[9] André Neubauer, Springer, DFT – Diskrete Fourier-Transformation, 21.07.2014

[10] Alfred Mertins, Springer, Signaltheorie, 21.07.2014

[11] http://www.thefouriertransform.com/series/fourier.php

[12] https://ccrma.stanford.edu/~jos/mdft/

[13] http://paulbourke.net/miscellaneous/dft/

[14] http://www.ignaciomellado.es/blog/Measuring-heart-rate-with-a-smartphone-camera

[15] http://www-users.cs.york.ac.uk/~fisher/mkfilter/trad.html