Non-Invasive Monitoring and Prevention of RBD Episodes

Bachelor Graduation Thesis

TU Delft

Momo Medical

Non-Invasive Monitoring and Prevention of RBD Episodes

by

Student Name	Student Number
K.B. Dzhumageldyev	5274230
J.H. Kruize	5261961
T.R.J. Schram	5353175

TU Delft Supervisor:	Prof.dr. Paddy French
Momo Medical Supervisor:	Ir. Thomas Bakker
Project Duration:	April, 2023 - June, 2023
Faculty:	Faculty of Electrical Engineering, Mathematics & Computer Science, Delft
Cover:	Storyset Illustrations

TU Delft

Abstract

The goal of this project was to develop a sensor system that detects Rapid Eye Movement Sleep Behaviour Disorder (RBD) episodes in an early stage and brings the patient to a lighter sleep stage at the episode onset. This project was completed in close collaboration with Momo Medical, a fast-growing start-up and developer of the BedSense - a device placed under mattresses that tracks the bed posture and restlessness of residents in nursing homes. To detect RBD episodes, the BedSense was used in combination with a sock that houses the biosensors EDA, PPG and an accelerometer. The sock also contains a vibrator module, that can bring the patient to a lighter sleep stage in case of an episode.

The outcomes of this project are promising, data has been recorded from both non-RBD test subjects and an RBD patient, and the vibration module has been tested.

The results of this project were integrated with another project seeking to develop an algorithm to detect RBD episodes in an early stage. This project used data from the Momo BedSense and from the sock that has been created in this project.

Preface

Rapid Eye Movement Sleep Behavior Disorder (RBD) is a neurological parasomnia disorder caused by the absence of muscle paralysis during REM sleep, leading to individuals acting out their dreams and potentially causing harm to themselves and others. The goal of this project was to design a non-invasive and user-friendly system capable of detecting episodes caused by RBD and bringing patients to a lighter sleep stage.

In order to achieve this goal, two subgroups were formed; one was responsible for developing a hardware system, and another focused on designing a software system. This thesis focuses on the design process, method, and results of the hardware subgroup.

The goal of the hardware subgroup was to design and implement a hardware system capable of measuring biosignals related to RBD episodes in an early stage and bringing the patient to a lighter sleep stage in case of an episode. The designed hardware system is complementary to the Momo BedSense. On the other hand, the goal of the software subgroup was to develop an algorithm to detect a RBD episode in an early stage, based on the data from the BedSense and the hardware system that is designed in this thesis.

We would like to express our gratitude to the Momo Medical team: Thomas Bakker, Danny Eldering, and Karen van der Werff. We are particularly grateful for Thomas Bakker's input and guidance in helping us realise a successful project. We would also like to extend our thanks to our supervisor, Paddy French, for his mentorship and guidance.
A special thanks goes to Krishnan Kandiyoor, Jasper Post, and Pepijn Kremers, the members of the software subgroup. It was a pleasure working with you to make this project a success.
Lastly, our sincerest thanks to Bert Verzijl, a patient diagnosed with RBD who graciously agreed to collaborate with us. His invaluable insights and goodwill have been of great help in the success of this project.

Delft, June 2023
Kerim Dzhumageldyev, Jan Kruize, and Tijn Schram

Contents

1 Introduction

In this chapter, important concepts regarding Rapid Eye Movement Sleep Behaviour Disorder that are applied in the project will be explained. After that, the goal of the entire project will be defined and the task division of the subgroups will be presented. Lastly, the structure of this thesis will be briefly described.

1.1. Rapid Eye Movement Sleep Behaviour Disorder

Rapid Eye Movement (REM) Sleep Behavior Disorder (RBD) is a neurological parasomnia disorder that affects muscle atonia during REM sleep [1]. To gain a better understanding of RBD the sleep stages of humans will briefly be explained.

1.1.1. Sleep Stages

Regular sleep consists of four stages: N1, N2, N3, and REM [2]. The N1, N2, and N3 stages are considered non-rapid eye movement (NREM) sleep, and roughly 75% of sleep is spent in these stages. During a typical 8 hours of sleep, there are roughly 4 to 5 sleep cycles, where a sleep cycle follows the following sleep stage progression: N1, N2, N3, REM. One sleep cycle typically lasts around 90 to 110 minutes, with the majority of the time spent in the N2 stage.

The first sleep stage is 'N1', commonly referred to as 'light sleep'. During this stage, there is some light muscle tone in the skeletal muscles and breathing tends to occur at regular intervals [2]. Eye movement, heart rate, and body temperature all decrease. This sleep stage lasts around 1 to 5 minutes, accounting for approximately 5% of the total sleep time [2]. As the night progresses, an uninterrupted sleeper may spend even less time in this stage [3]. The next stages are N2 and N3, which are deep sleep, and usually, it is quite difficult to wake people up in these stages [2]. They account for 45% and 25% of total sleep respectively [2]. In these stages, the body is the most relaxed. When a person is woken up in one of these stages, they often feel groggy and they usually have diminished mental performance for 30 minutes to an hour [2].

The last stage and the focus of this research is the REM stage. This is the stage where dreams occur and brain waves are similar to those of a wakeful state. However, all skeletal muscles, except for the eyes and breathing muscles, are atonic and without movement [2]. In fact, this is the reason this sleep stage is called rapid eye movement, as the eyes may move much more during this stage compared to the other stages. This stage initially lasts 10 minutes, but it gets longer in subsequent sleep cycles. After this stage, the cycle starts again [2]. Figure 1.1 illustrates an average night of sleep.

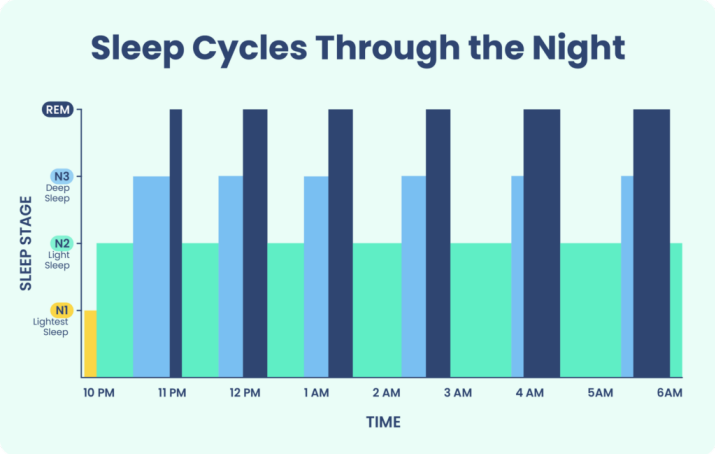

Figure 1.1: An average night's sleep with the sleep stages. [3]

1.1.2. REM Sleep Behaviour Disorder

People suffering from RBD do not experience any kind of muscle paralysis during the REM sleep stage. This leads to people acting out their dreams, potentially resulting in dangerous situations causing them to hurt themselves or their bed partners. The muscle movements of RBD patients during REM sleep consist of jerky and rapid movements in contrast to a wakened state [4]. Injuries that have been documented include lacerations, dislocations, and hair pulling among many others [5]. The frequency of episodes is highly variable, sometimes only occurring once every 2 weeks, but at other times occurring multiple times a night for consecutive nights [5]. On top of that, RBD may be a precursor to neurodegenerative diseases, like Parkinson's and Alzheimer's Disease [6]. The time between RBD diagnosis and the onset of such diseases is usually 5 to 15 years [4].

1.1.3. Diagnosis

There are currently three different standards used to diagnose RBD (see Appendix B). Diagnosis is primarily done using a polysomnography (PSG), often accompanied by video recordings to visually verify the results. A PSG is a sleep study used to diagnose many illnesses and disorders and consists of an extensive monitoring system that records electrical activity in the brain, muscles, retina, respiration, and heart rate among other things [7]. The electrical activity is measured using electroencephalogram (EEG), electromyography (EMG), and electrooculogram (EOG) techniques. With the information from these sensors, the sleep stages can be determined, and combined with other data gathered during the study, a diagnosis of RBD can be made.

Traditionally, it was thought that there was a gender disparity regarding the occurrence of RBD [6] [5]. However, a recent group study found that this was not the case [8]. Instead, males tend to experience more issues from RBD because they usually have more aggressive dreams. This leads to males seeking out medical help more often than females. Additionally, the same study found that approximately 1% of the general population has RBD [8]. The average age of patients suffering from RBD is typically around 50 to 60 years old [1] [9].

1.1.4. Treatment for RBD

There are currently no treatments specific for RBD, but the symptoms can be controlled by medication [10]. Two commonly used medicines to suppress RBD symptoms are clonazepam and melatonin [11], although the effectiveness of these medicines is based upon case series, small clinical trials, and expert consensus [1]. These medicines may not work for every patient and may introduce unwanted side effects. Another way to prevent injuries due to nightly movement is to use restraints or employ safety measures such as a sleeping bag [10]. A simple albeit crude solution, it is the easiest and most cost-effective way to alleviate some of the struggles of RBD.

1.2. State-of-the-Art Analysis

A lot of research has been conducted on the diagnosis of RBD. For instance, one study explored diagnosing RBD based on wrist actigraphy alone [12]. This method is less intrusive and easier to use compared to polysomnography. The outcomes of this research are supported by other studies [13]. Another less intrusive approach to diagnosing RBD involves measuring the muscle activity [14]. These electromyography (EMG) signals are fed into a machine-learning algorithm with a random forest classifier. This algorithm can diagnose RBD with an accuracy of 92%. EMG is also a good measure for RBD diagnosis without a machine learning algorithm [15].
EMG is not the only suitable biosignal to diagnose patients with RBD. The electrical activity of the heart and the electrical activity of the retina are also promising measures for the diagnosing of RBD [16].
Another biosignal worth considering is electrodermal activity (EDA). An EDA sensor measures the skin conductivity and can be used to detect sleep stages and convulsive seizures [17].

Less research has been done on the real-time detection of RBD episodes. Some studies have focused on detecting episodes at a later stage when the patient has already left the bed [18]. In this research, a pressure sensor is used to detect whether a patient has left the bed. When the pressure sensor detects a sudden drop in pressure, indicating that the patient left the bed, a voice recording is played. This is a voice recording of a family member that tells the patient to go back to bed. In addition to the pressure sensor, one patient used a bed exit monitor that operates with a tethering cord. When the tethering cord is pulled, the voice recording will play too. While this research is a step in the right direction, it only measures the episodes at a later stage. The findings of this research demonstrated a decrease in the number of episodes after using the system for some time.

A proven method for real time-time monitoring of RBD patients is the utilisation of radio-frequency signals [19]. This research enables continuous spatial monitoring of RBD patients. However, it may be too late to intervene and wake the patient if they are already moving excessively. Nevertheless, this approach is non-intrusive.

To ensure that the sensor data is only used during REM sleep, a REM sleep detector could be implemented. During REM sleep, the eyes move rapidly. This can be detected by measuring the movement of the eyes. The size of the pupil can be measured even with closed eyelids [20]. This research could be used to develop a pupil tracker that works with closed eyelids. It is important to note that this approach may be quite intrusive and could potentially have long-term harmful effects on the eyes.

1.3. Goal of the Project

As mentioned previously, there are currently no specific treatments for RBD episodes. The State-of-the-Art Analysis showed that there are well-functioning novel methods to diagnose patients and detect episodes. However, these episodes can only be detected in a late stage, when the patients already experience significant restlessness.

Therefore, the overall goal of this project is to design a complete system capable of detecting upcoming episodes caused by RBD at an early stage and preventing them. The system to be designed must be non-invasive, non-intrusive, and user-friendly.

To achieve the aforementioned goal, two subgroups were formed; one was responsible for developing a hardware system, and the other focused on designing a software system. Figure 1.2 illustrates the interconnection of the subsystems. The objective of the hardware group was to design and build a device capable of gathering biomedical signals and bringing the patient to a lighter sleep stage.

This thesis focuses on the design process, methodology, and results related to the goal of the hardware subgroup, which was: **To design and implement a hardware system to measure signals related to RBD episodes in an early stage, that can bring the patient to a lighter sleep stage in case of an episode.**

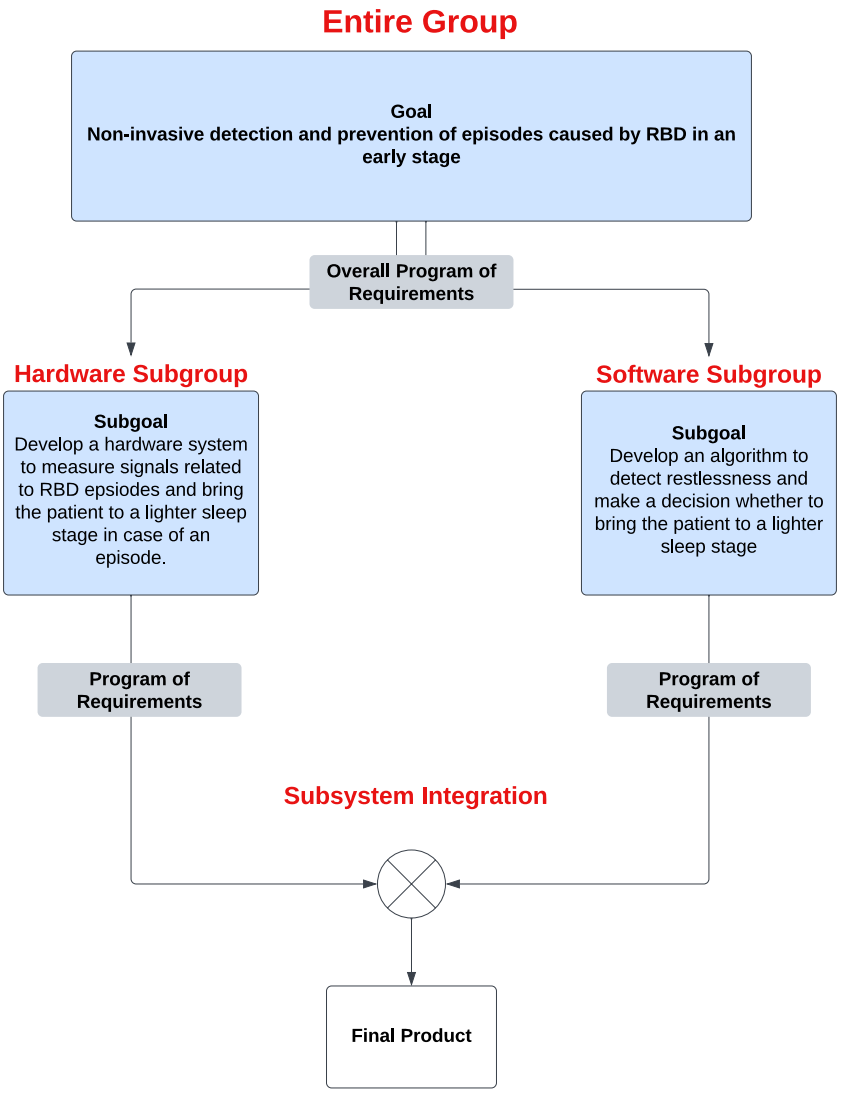

Figure 1.2: Overview of the project and subsystem goals

1.4. Structure of Thesis

This thesis is structured as follows: Chapter 2 presents the Program of Requirements for the entire system and specifically for the hardware group. In Chapter 3 the possible sensors are explained and eventually, a choice will be made for a sensor combination. Chapter 4 will elaborate on the reading of the sensor data by the ESP32 microcontroller. Chapter 5 explores different waking methods and a decision will be made on what waking method will be used in the prototype. Chapter 6 will describe the process of designing and manufacturing the prototype. Chapter 7 discusses the collection and testing of data on the prototype. Finally, Chapter 8, presents a conclusion and a discussion, along with suggestions for future work.

2

Program of Requirements

The Program of Requirements describes the requirements for the system to be designed. These requirements are categorised into two sets: requirements for the entire system and requirements specifically for the hardware component of the system. These requirements are further divided into two subsets: functional and non-functional requirements. Functional requirements outline the desired functionalities and capabilities of the product, while the non-functional requirements describe the general properties of the system, such as quality.

2.1. Requirements for the Entire System

The functional requirements for the entire system are defined as follows:

1.1 The system must be applicable to multiple patients.
1.2 The system should work continuously for at least 10 hours.
1.3 The system must facilitate real-time episode detection.
1.4 The time elapsed between an RBD episode onset and bringing a patient to a lighter sleep stage should be less than 15 seconds.

The non-functional requirements are defined as follows:

2.1 The system should be non-invasive.
2.2 The system should be stand-alone. So it should work without any external products or applications.
2.3 The system should be user-friendly. Patients should be able to use the system without having any prerequisite knowledge.
2.4 The system should be wireless.

The safety requirements describe how safe the system should be:

3.1 The system must not cause harm to the patient.

2.2. Requirements for the Hardware Group

The functional requirements that apply specifically to the hardware system are formulated as follows:

A.1 The system should measure signals related to an RBD episode in an early stage.
A.2 The measured signals should be complementary to the signals measured by the BedSense.
A.3 The system should be battery-powered, and the battery should be able to supply the system with power for at least 10 hours.
A.4 The system should be wireless.
A.5 The system should get the patient out of REM sleep within two seconds after episode detection.

A.6 The system should communicate with the software system on the Momo BedSense via Bluetooth Low Energy.

A.7 The system should bring the patient effectively to a lighter sleep stage. This means that the patient should not be fully woken up, but only moved out of REM sleep.

A.8 The system should wake the patient in a targeted manner (waking only the patient and not potential bed partners).

The non-functional requirements that apply to the hardware system are defined as follows:

B.1 The system should be non-invasive.

B.2 The system should be user-friendly.

B.3 The system should be comfortable to wear.

B.4 The system must not cause harm to the patient.

B.5 The code should be well-crafted and structured in a manner that allows for easy future expansion.

3

Sensors

This chapter concerns the choices of the sensors. The sensors have to record the biosignals most relevant to detecting RBD episodes. Therefore, biosignals related to RBD will be discussed. Moreover, a decision will be made regarding the appropriate sensors to use for this project.

3.1. Possible Sensors

Momo BedSense

The sensor system that was used in the project, along with other sensors, is the Momo BedSense, shown in Figure 3.1. The BedSense is a collection of sensors developed and produced by Momo Medical and commonly employed in nursing homes to provide a detailed overview of the patient's sleep patterns. It consists of six piezoelectric sensors, four force-sensing resistors, and a single accelerometer. The piezoelectric sensors are sampled at 120 Hz, while the accelerometer and force-sensing resistors are sampled at 10 Hz. The BedSense is placed beneath the mattress at chest height. Proprietary data analysis on the BedSense data showed that movements of the arm are clearly distinguishable. Leg movements, however, are not distinguishable from this data [21]. As RBD episodes do consist of leg movements [22], relying solely on the BedSense would not suffice for effective detection of RBD episodes.

Figure 3.1: Momo BedSense [23]

Electromyography (EMG)
Electromyography (EMG) is a physiological signal that captures muscle activity by detecting the electrical signals generated during muscle contraction. EMG signals have been used as a valuable method for detecting RBD episodes. Several studies have reported the utilization of EMG activity to diagnose patients with RBD, as the patient's muscles exhibit movement during an RBD episode, which is detected by the EMG sensor [24]. However, an EMG sensor requires electrodes with gel, which makes the measurements more intrusive, uncomfortable, and non-user-friendly. Consequently, it does not align with requirements B.1, B.2, and B.3.

Electrodermal Activity (EDA)
Electrodermal activity (EDA), also known as galvanic skin response (GSR), measures changes in the conductance of the skin, caused by changes in sweat gland activity. EDA has been strongly associated with emotional arousal, as emotional responses trigger changes in eccrine sweat gland activity [25]. Therefore, if a patient experiences a stressful dream before an RBD episode, there would be a change in emotional response, which can be detected by the EDA sensor. An advantage of the EDA sensor is its ability to provide data indicating the potential occurrence of an RBD episode before it happens. However, an EDA sensor alone cannot adequately measure an RBD episode and needs to be used in combination with other sensors. It can only provide an indication of a change in emotional response, which may be attributed to a dream alone.

Electroencephalogram (EEG)
An EEG sensor senses the electrical activity of the brain. This is achieved by positioning electrodes on the scalp to detect the electric signals originating from the brain. EEG is a biosignal strongly linked to RBD episodes [26]. However, there are several issues when it comes to recording this signal. Firstly, the signal is complex and requires extensive filtering before use. Secondly, EEG measurements involve placing electrodes on the scalp using electrode gel, which can be uncomfortable to sleep with. As a result, the EEG sensor does not conform with requirement B.3.

Electrocardiogram (ECG)
An electrocardiogram sensor consists of electrodes placed on specific locations on the skin to measure the electrical activity of the heart. Preliminary studies have indicated a lower heart rate associated with movement in sleep during an RBD episode [27], indicating that ECG could be a sensible sensor choice. However, ECG is typically measured on the chest and the electrodes need to be applied with electrode gel. This may cause discomfort for the person sleeping, thereby not meeting requirement B.3.

Radio-frequency (RF)
A non-intrusive way to possibly detect RBD episodes is using radio-frequency [19]. In this approach, an RF signal is continuously transmitted to the patient. In normal sleeping conditions, when there is no movement, the signals are mostly reflected back to the receiver without distortion. However, during an RBD episode when the patient begins to move, the reflected signals become distorted. In principle, this non-intrusive approach holds promise. However, it does not provide additional information beyond the BedSense data. Therefore it does not fulfil requirement A.2.

Photoplethysmogram (PPG)
A PPG sensor is used to measure variations in blood volume within the bloodstream. The sensor operates by emitting red light using an LED which penetrates human tissue and blood vessels. The reflected light is then detected to measure the bloodstream [28]. This process is demonstrated in Figure 3.2. Changes in blood pressure can help infer changes in heart rate. In addition to measuring heart rate using red light ($\lambda \approx 660$ nm), the PPG sensor can measure oxygen saturation using infrared light ($\lambda \approx 880$ nm).

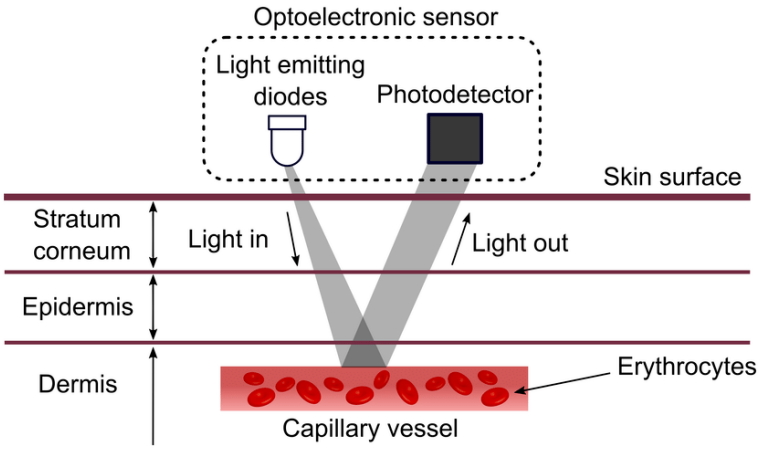

Figure 3.2: PPG sensor [28]

The PPG sensor is effective in measuring an RBD episode due to changes in heart rate, which occur during such episodes. During REM sleep, the heart is more likely to fluctuate [27]. Additionally, as mentioned before, preliminary studies have indicated a decreased heart rate associated with movement during sleep in an RBD episode [27].

Accelerometer (ACC)

An accelerometer is a sensor that measures acceleration in one or more axes. The way it operates varies depending on the specific accelerometer. However, it is an effective way to measure sudden movement in a specific location of the body. During an RBD episode, the patient exhibits erratic and significant movement in multiple body parts. Supporting this, a study reported that RBD episodes could be detected by monitoring leg movement [29] [26].

Laser

Another non-intrusive method of measuring movements during sleep is through the use of a laser. In this method, a laser beam is projected across the room, and a sensor is placed at the other end to detect any interruptions in the light. For example, if a person moves their arm and interrupts the beam, the system would detect a potential RBD episode. It is crucial to choose the appropriate height for the sensor so that any movement above this threshold height would be considered a potential RBD episode. However, this sensor also raises some concerns. For instance, distinguishing between normal sleep movements and an RBD episode may be challenging. It had to be decided whether measurements in the Z direction were sufficient, or that additional sensors in the X and Y directions were required to improve detection accuracy. Due to this ambiguity, the sensor may potentially diagnose the RBD episode at a later stage, failing to fulfil requirement A.1.

Camera

A camera can also be an effective sensor for detecting RBD episodes. This method involves recording the patient during sleep and analyzing their movements to determine if they are experiencing an RBD episode. However, there are some limitations to this approach. One of the main challenges is the time it takes to process the recorded images and run them through an algorithm to make a decision. This delay can be significant and may result in a situation where the patient has an RBD episode and gets hurt before the system can detect and alert about the episode. This implies that the sensor would not fulfil requirement A.5. Therefore, it is important to consider the speed and efficiency of the algorithm used to process the recorded images. This is to minimize the delay and ensure timely detection of potential RBD episodes. Nevertheless, a positive aspect of this sensor is, similar to the previous sensors, that it is non-intrusive.

Infrared/Electrooculography (EOG) Eye Tracker

RBD only occurs during REM sleep, so an effective way to detect an RBD episode is to monitor if the patient is in REM sleep. This can be done by tracking eye movement, as REM stands for Rapid Eye Movement. Tracking eye movement during sleep, however, can be challenging since the patient's eyelids are closed. Nevertheless, using infrared or electrooculography (EOG) technology, it is possible to track eye movement even when the patient is asleep. This could be implemented by using a sleeping mask with the sensors attached. Infrared eye trackers shine infrared light close to the eye to detect the position of the pupil [30]. On the other hand, an EOG sensor measures the biosignals of the eye [31]. However, both sensors would not meet requirements B.3 and B.4. Requirement B.3 would not be met because shining an infrared signal into the eye could potentially be dangerous. Moreover, requirement B.4 would not be met because sleeping with a mask that contains electronics could be uncomfortable.

Sensor Comparison

To compare the sensors and choose the optimal set of sensors, the requirements for sensors will be used. The requirements are described in the Program of Requirements. In Table 3.1, a summary is provided for each sensor to determine if they meet the requirements. This table only lists the requirements that are relevant to sensor choice.

Table 3.1: Sensor comparison with the appropriate requirements

	Req. A.1	**Req. A.2**	**Req. A.5**	**Req. B.1**	**Req. B.3**	**Req. B.4**
EMG	X	X	X	X		X
EDA	X	X	X	X	X	X
EEG	X	X	X	X		X
ECG	X	X	X	X		X
RF	X		X	X	X	X
PPG	X	X	X	X	X	X
Accelerometer	X	X	X	X	X	X
Laser		X	X	X	X	
Camera	X	X		X	X	X
Infrared/EOG	X	X	X	X		

From Table 3.1, it can be concluded that the best sensors to use are the EDA sensor, PPG sensor, and accelerometer.

3.2. Sensor Choices

From the sensor comparison, it was concluded that the sensors that an EDA sensor, a PPG sensor and an accelerometer will be used. The following exact sensors were chosen for this project:

- EDA sensor: Grove GSR
- PPG sensor: Analog Devices MAX30102
- Accelerometer: STMicroelectronics LIS3DH

The Grove GSR sensor was chosen for the EDA sensor. There was not much choice in EDA sensors, the only other choice was the CJMCU-6701 sensor. The availability of this sensor is worse and the Grove GSR sensor is more commonly used with more documentation available, making it a better choice for prototyping. It is also smaller than the CJMCU-6701, resulting in improved comfortability.

The MAX30102 PPG sensor was chosen as it came with both red and infrared light, enabling to measure heart rate and oxygen saturation alternately. Another advantage of this sensor is that it comes with a built-in buffer and integrated sample averaging.

The LIS3DH accelerometer was chosen for its selectable scale of ±2g/±4g/±8g/±16g and for its built-in buffer. More detailed specifications of all the sensors can be found in Appendix C.

4

Sensor Readout

This chapter will describe how the data of the PPG sensor (MAX30102), EDA sensor, and accelerometer (LIS3DH) are read out.

4.1. I^2C Communication

In this section, a short overview of the I^2C serial communication protocol will be given. I^2C uses two wires consisting of a data line (SDA) and a clock line (SCL). It is a multi-master multi-slave protocol with the possibility to have up to 128 slaves (using an 8-bit format) and an unlimited amount of masters. The least significant bit of this address specifies a read/write (R/W) operation. When the master initializes a connection with a slave, it sends a start condition by pulling down the SDA. After a short delay, the SCL line will be pulled down as well. Next, the master sends a frame consisting of the slave address to which the master wants to write, together with an R/W bit that specifies the operation. After the slave receives its own address it sends an ACK signal by pulling down the SDA line for one clock cycle. Depending on the type of operation, the master sends or receives data from the slave. The recipient of the data sends an ACK after receiving a frame. Following that, the master decides to stop the communication and free the serial bus. It sends a stop condition by pulling up the SCL line and after a short delay also pulling up the SDA line. Figure 4.1 provides a visual representation of a message being sent in the I^2C protocol.

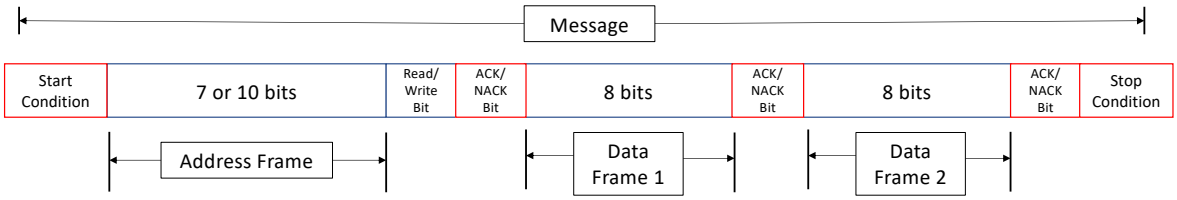

Figure 4.1: I^2C Protocol

4.2. Sensor Structures

Both the MAX30102 [32] and the LIS3DH [33] sensor independently sample their respective biosignals to ensure accurate sample timings and reduce the host processor's workload. The analog output of the EDA sensor is sampled using the ESP32's built-in 12-bit analog-to-digital converter (ADC). The microcontroller makes use of its two built-in I^2C controllers to communicate with the MAX30102 and LIS3DH sensors. To start the data collection with these sensors, they first have to be programmed by writing to specific registers specified in their datasheets. After the initial programming, the sensors start to independently sample and fill their buffers, waiting for the microcontroller to read their data. Both sensor buffers are used as a first-in, first-out buffer (FIFO), in which for every new sample an old

sample is overwritten in a cyclic order.

The MAX30102 and the LIS3DH sensors need to be communicated with in a specific way to program the sensors and make sure the sensors operate in the way that is desired. Appendix D includes a comprehensive explanation of the procedure employed and the specific registers that have been written to in order to ensure accurate operation.

4.3. Sampling

This section will describe the frequencies at which the sensors should be sampled. Section 4.3.1 describes the ideal sampling frequencies, whereas Section 4.3.2 describes the sampling frequencies that are used for data collection.

4.3.1. Ideal

The frequency spectrum of a heart rate is about 24 - 240 bpm, which is relatively wide considering the average heart rate is between 60 - 100 bpm [34]. To accurately capture all possible heart rates and remove all other noise, the PPG data should at least be sampled at 8 Hz after filtering out high-frequency noise with a low-pass filter with a cut-off frequency of 4 Hz.

Sampling the accelerometer at 50 Hz results in a maximum measurable frequency of 25 Hz, using Nyquist sampling theorem. It was concluded that this sampling rate was capable of extracting small rapid movements of the leg

The frequency spectra of electrodermal activity signals are not clearly defined, however, most research shows that it is between 0 - 2 Hz [35] [36]. Using the Nyquist sampling theorem, the sampling frequency should be chosen to be 4 Hz, while using a low-pass filter with a cut-off frequency of 2 Hz before sampling. This ensures that high-frequency noise is filtered out and aliasing is avoided.

4.3.2. Data Collection

For data collection, higher sampling rates than the theoretical minimum are used to ensure that there are no overlooked frequency components being discarded.
The sampling rates chosen for data collection are as follows:

- PPG: 50 Hz
- Accelerometer: 50 Hz
- EDA: 25 Hz

When using the sensor system together with the episode detection algorithm, the sampling frequencies should approach the ideal sampling frequencies to limit the processing workload.

4.4. Implementation

The best way to program the ESP32 is to use Espressif's official development framework (ESP-IDF). This would give the most control over the hardware. However, due to the critical need to design an early-stage prototype for data collection purposes, the decision was made to utilise the Arduino framework instead. This framework has access to a wide range of built-in functionalities and libraries to sufficiently control the hardware of the ESP32. This framework encapsulates a lot of functionality of the ESP-IDF, but it adds an extra layer of abstraction. It is important to note that for future iterations it is wise to switch to ESP-IDF before expanding the system further. This is to keep the number of layers of abstraction as low as possible and ensure overall efficiency.

Using the Arduino IDE, the code for the ESP32 is written in C++. In the following subsections, functional descriptions of the written code will be given.

General structure

A package has been created to have maximal control over the LIS3DH and MAX30102 settings, as well as timings and communication. The package consists of a single base class and 3 of its derived

classes, one for each sensor type used in this project. Specialized member functions have been created for each of the derived classes, to collect their corresponding sensor data in a well-structured and organised way. All the code is dependent on precise timing and is based on the system run-time timer, which consists of a 32-bit counter that increments every millisecond. This means that about every 49 days, the counter rolls over and resets. Upon starting the system the sensors are programmed and the system counter starts to count. For every loop in the program, the total run time is checked and the difference is calculated between the last time the sensor information was pulled and the current time. Upon reaching the desired difference new information is requested using the function `getData()` and the current time is stored. The sensor system makes use of 32-sample buffers for the PPGs and the accelerometer, while only a 16-sample buffer for the EDA sensor is used. This is the case since the PPGs and the accelerometer sample frequencies are twice as high as the sample rate of the EDA sensor. Once the buffers are full, the samples in the buffer are stored and ready to be processed. The buffers are freed and can be overwritten with new data. The buffer size of both of the PPG sensors is 256 bytes, while the accelerometer and the EDA buffers are 192 and 32 bytes respectively. The exact sizes of the buffers are rather arbitrary since every size which follows the following requirements suffices:

1. The total size of each buffer divided by its sample frequency should be the same for each sensor.
2. The time it takes to fill all the buffers should not take more than 5 seconds, since having a larger buffer results in a longer response time.
3. The total buffer size should not take too much space in memory since the ESP32 has a limited memory size. The memory size of the used ESP32 is in the order of hundreds of kilobytes. However, this could change due to potential changes after expanding the program.

Considering the current settings, the limit of the buffer size is set due to the maximum output delay of 5 seconds. This results in a buffer size of 250 samples for the PPGs and the accelerometer and in 125 samples for the EDA sensor.

MAX30102 read-out

Every polling period the sensor readout begins by a read operation, requesting the pointer to the start of the FIFO buffer over the I^2C bus. This pointer is being compared to the pointer to the end of the FIFO buffer, which is being updated internally by the ESP32. From this information, the total number of unread samples is calculated. This value is compared to the current level of the ESP32's buffer and it is then decided how many samples are to be extracted from the FIFO buffer. When the number of new samples in the FIFO buffer is less than the space left in the ESP32 buffer, all the data in the FIFO buffer is extracted. When there is limited space left in the buffer of the ESP32, it only requests the number of samples necessary in order to completely fill the buffer. After deciding the number of samples to request, the pointers are updated accordingly and the ESP32's buffer is filled with the new data. This process is implemented in the function `MAX30102::getData()`, which can be found in Appendix F.3. More detailed information about sensor readouts and the exact procedure can be found in Appendix D.

LIS3DH read-out

Every polling period the sensor readout begins with a read operation requesting FIFO_level over the I^2C bus. FIFO_level represents the total number of unread samples in the FIFO buffer. When the total number of unread samples is lower than the space left in the buffer of the ESP32, all the data from the FIFO buffer is extracted. When there are more unread samples in the FIFO buffer than there is space in the ESP32 buffer, the ESP32 only requests the number of samples to fill its buffer completely. The implementation of this is done by the function `LIS3DH::getData()`, which can be found in Appendix F.5.

EDA read-out

Sampling of the EDA sensor is done by tracking the time between each sample and using the built-in 12-bit ADC of the ESP32. This samples every 40 ms and saves the sample in the ESP32's buffer. This is implemented using the function `EDA::getData()`, which can be found in Appendix F.7.

Storage

Since sampling is done asynchronously with respect to the other sensor modules, not all the buffers for the different sensors are filled at the same time. Saving the contents of the ESP32's corresponding sensor buffer in memory at the moment it is full prevents the overwriting of unread samples. This will also ensure that there is no stalling of the new incoming data since the sensor buffer can freely refill with new data without having read the old data in the buffer.

Diagram 4.2 illustrates the data flow between the data buffers in the sensor system for the PPG and accelerometer.

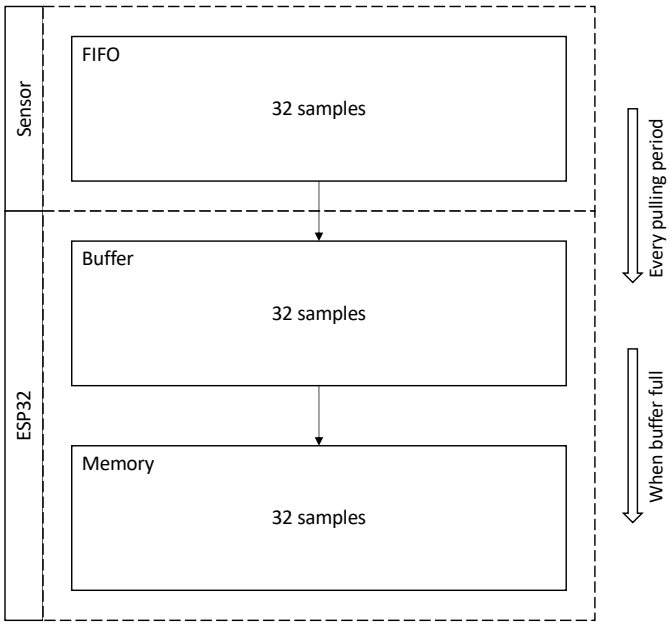

Figure 4.2: Samples in memory

Every time the ESP32 requests new data from the FIFO buffer of the accelerometer and PPG sensors, which is done every polling period, several samples are stored in the ESP32 buffer. Once the ESP32 buffer is completely filled, the contents of this buffer get copied to memory, and new data can flow from the FIFO buffer to the ESP32 buffer. Since the EDA sensor does not have a FIFO buffer, the ESP32 samples a single EDA sample every polling period. After this, the same process follows as for data collection of the PPG sensor and accelerometer.
The source code can be found in Appendix D.

5

Waking Methods

A crucial part of this project is waking the patient when an RBD episode is detected. There are various methods to accomplish this, and the following chapter will explore them. At the end of this chapter, a waking method will be selected for this project.

5.1. Possible Waking Options

Sound

One of the most widely used ways to wake a person up is through sound. This could be an alarm that would go off when an RBD episode is detected. Furthermore, it is not necessary to wake up the patient completely but just move them from REM sleep to another lighter stage of sleep. It is to be investigated how long the sound would need to be played, and at what volume. Additionally, using sound as a waking option does not meet the requirements for effective and targeted awakening (requirements A.7 and A.8). The reason is that the sound can potentially wake up the bed partner alongside the patient, which is not ideal. A solution to the issue would be to use earbuds, yet this could be uncomfortable for the patient. Another disadvantage is that people who suffer from RBD are likely to be older and may have worsened hearing due to their age. Nevertheless, certain frequencies can still effectively wake up older people [37]. However, the age of the patient and their hearing would need to be considered when designing such an alarm.

Light

Light can be an effective way to wake up the patient during an episode. There are existing products already that wake up people using light systems. This technology uses dawn simulation, also called wake-up lights. They slowly simulate sunrise in order to gently wake a person up. Similar technology has also been incorporated into a sleeping mask, which would fulfil the requirement of targeted awakening (requirement A.8). However, this would take too long. If the system incorporated light as a waking method, it would need to be immediate and bright. Even though this would work, such a solution would most likely be undesirable since it would not gently wake the patient out of REM sleep. Instead, it would wake the patient entirely. This is not ideal since according to the requirements the patient has to be brought to a lighter sleep stage, and not woken up completely (requirement A.7).

Heat

A more novel way to wake a patient up would be through the use of heat. If the sensor system would be located on the body of the patient, it could wake them up. However, there are numerous concerns about the reliability and safety of such a method to wake the patient. Sleep arousal with heat stimuli (shifting from a deeper sleep stage to a lighter sleep stage) worked in below 50% of the cases [38]. This does not meet the requirement of effective awakening (requirement A.7). Moreover, using such a system could potentially cause harm to the patient or could burn the patient and his surroundings. This does not meet the requirement that the system should be safe to use (requirement B.3).

Vibration
Another method of waking that is becoming increasingly popular is vibrating. This is caused by the introduction of smartwatches that can be worn comfortably during the night. Patients do not have to be bothered with sound anymore, but a vibration on the wrist can be enough to wake the patient up. The method of vibration complies with all the requirements that are set in the Program of Requirements.

Electric Shocks
What could also be a potential method to wake the patient is the use of electric shocks. However, similar to waking by heat, this method suffers from setbacks. Waking the patients by shocking them might be too effective, they will be awoken immediately, and their sleep will be disturbed. Safety is a factor that should be taken into account very seriously in case of using a shocking device.

5.2. Waking Choice
Similarly to Chapter 3, a comparison between the waking methods has been done and is summarised in Table 5.1. This table only lists the requirements that are of interest when choosing the waking method.

Table 5.1: Waking method comparison with the appropriate requirements

	Req. A.5	**Req. A.7**	**Req. A.8**	**Req. B.1**	**Req. B.3**	**Req. B.4**
Sound	X	X	X*	X	X*	X*
Light	X	X	X*	X	X*	X*
Heat	X			X		
Vibration	X	X	X	X	X	X
Electric Shock	X		X	X		

Where a ”*” means that this waking system can be either more intrusive or less targeted.
From Table 5.1, it can be concluded that sound, light, and vibration are good methods to wake the patient. However, light and sound have an asterisk on some of the requirements. For example, the light system could either be a light that shines on both partners or a system that is in a mask concentrated on the patient. Therefore, the choice of the waking system will be the vibration module. This is further supported by the fact that our sensor system will not be placed at the head, meaning that there is a need for wires or for an extra battery-powered system. In addition, having spoken to an RBD patient that uses the vibrating alarm clock in a smartwatch, it was concluded to use a vibrating module to wake the patient in case of an episode.

The chosen vibration module is the Adafruit 1201 mini vibrating disk. This is a small module that measures 10 mm in diameter and is 2.7 mm thick. It is able to operate in the voltage range from 2 V to 5 V, where increasing the voltage increases the rotational speed of the module. These characteristics make it an ideal module for use in the system at hand; it is small and can directly operate at the GPIO output of the ESP32.

6 Prototyping

This chapter will describe all the design choices and the implementation of the prototyping process, including investigating the placement of the system, prototyping how the system comes together, and lastly the power unit for the system.

6.1. Placement

Two positions for the placement of the sensor system were identified: at the wrist or at the foot. The reason is that these are spots where accurate biosignal measurement can be achieved, and are more likely to be comfortable since it should not bother the patient during sleep. This section will elaborate on the placement of the sensor system.

As the Program of Requirements indicates, the sensor system should be complementary to the Momo BedSense (requirement A.2). Proprietary data analysis on the BedSense data showed that movements of the arm are clearly distinguishable in the data recorded from the BedSense. Leg movements, however, are not distinguishable from this data [21]. As RBD episodes do consist of leg movements [22], it would be good to measure leg movements with the sensor system. If the PPG and EDA sensors did not measure an episode, the accelerometer would be the last resort to detect the episode. To implement this, the accelerometer could be placed somewhere on the leg.

PPG sensors are usually placed at the fingertip of the index finger or at the earlobe [39]. EDA sensors are normally placed on the index finger and the middle finger [40]. As the movements of the legs should be measured, it would be practical to place the PPG and the EDA sensors also somewhere on the leg or foot.

Potential placements of EDA and PPG sensors on the foot are shown in Figure 6.1 [40] and in Figure 6.2 [41], respectively.

Figure 6.1: EDA sensor placement [40]

Testing was further done on all places that Figure 6.2 indicates, and the best signal-to-noise ratio was indeed obtained at location 1. The placement of the PPG sensor was therefore chosen to be location 1 of Figure 6.2. Testing showed that a slight deviation in the sensor placement results in a large signal drop, so two MAX30102 modules are used and placed beside each other in the final prototype. This ensures a reliable measurement of blood volume and blood oxygen saturation.

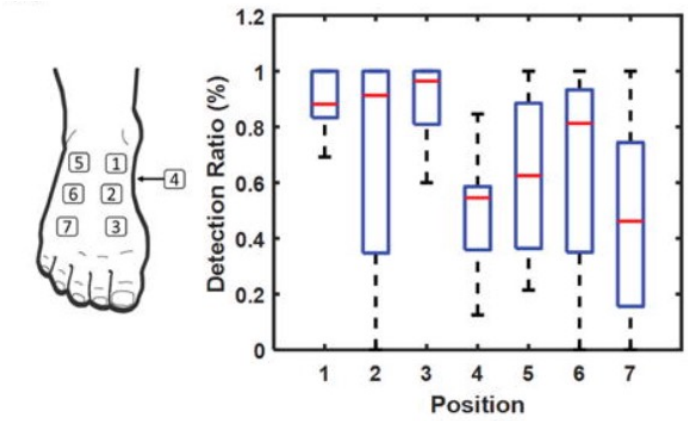

Figure 6.2: PPG sensor placement [41]

As described before, literature showed that PPG and EDA data can be measured accurately on the foot, and the Momo BedSense does not accurately measure leg movement. Hence, it was decided to create a sock that contains the PPG and EDA sensors, the accelerometer, the vibrator module, and the ESP32.

6.2. Sock Design

The components mentioned above had to be placed within the sock. The initial idea was to implement all the sensors and the ESP32 in a normal sock. After some preliminary testing with the PPG and EDA sensors, it was found that for the sensors to work accurately, they had to sit tight around the body. They should, however, not be too constrictive, to comply with requirement B.3. Therefore, a so-called compression sock was used. This is a sock that applies constant pressure to the lower legs. This sock was folded in half to make it easy to put on and improve comfort.

6.3. ESP32 Housing

To fit the ESP32 comfortably in the sock, a design of a housing box was made. This was then printed using a 3D printer with PLE filament, which resulted in the box in Figure 6.3. The circuit boards of the

EDA sensor and the accelerometer are placed inside this box too.

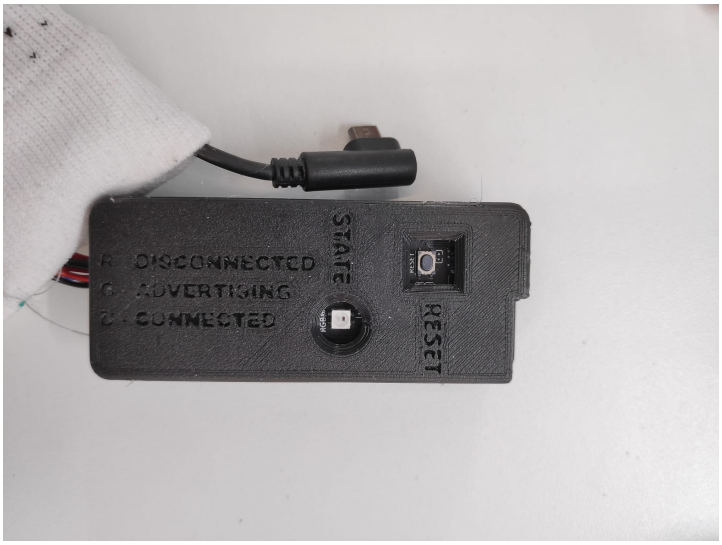

Figure 6.3: The 3D-printed box to house the ESP32, the EDA sensor and the accelerometer

6.4. Power Unit

Designing a power unit is a crucial part of the project to ensure the sock works wirelessly. Before designing the power unit, a power estimation needs to be made to know what kind of battery would be needed.

To make such an estimation the maximum power will be derived for all the sensors. For the accelerometer (LIS3DH) the voltage input is 1.7 V to 3.6 V, while the current use at the highest operating mode is 185 μA [33]. Therefore, the total energy consumption for LIS3D for 10 hours (requirement A.3) is 0.007 Wh. Similarly, for the PPG sensor (MAX30102) such a calculation can be made. With an input voltage of 3.3 V and the current at the highest operating mode being 50 mA [32], for 10 hours, this results in an energy consumption of 1.65 Wh. However, since there are 2 PPG sensors the energy consumption is doubled, resulting in 3.3 Wh. The EDA sensor in the highest operating mode results in a 10 mA current consumption at an input voltage of 3.3 V/5 V [42]. This means that for 10 hours, the sensor will consume 0.5Wh. All the sensors together consume 3.807 Wh in 10 hours.

Besides calculating the power for the sensors, the energy consumption of the ESP32 and the vibrator module have to be taken into account too. The ESP32 average current drawn is 130 mA with a voltage input of 5 V. Therefore, the energy consumption for 10 hours is 6.5 Wh.

The energy consumption of the vibration module is negligible, as it is only powered for a few seconds during an entire night. The vibration module draws 100 mA of current at a voltage of 5 V. The total energy consumption of the system is 10.307 Wh for 10 hours.

As a result, for an input voltage of 5 V, the battery would need 2061.4 mAh to provide the system with power for 10 hours. A standard Li-Po battery could cover that, so in theory, a power circuit could be built to accommodate that energy demand. However, for prototyping in this case it was decided to not implement a power circuit, but use a 5 V power bank. The above-mentioned approximation is of course a worst-case estimate and changes can be made at later stages of development to significantly improve the power consumption. Some design suggestions can be found in section 8.

The power bank that was chosen had a capacity of 5000 mAh. The reason for choosing a power bank with such capacity is that it would cover requirement A.3 more than enough. This means that if the prototype was given to the patient and they forgot to recharge it, the system could still be used. Furthermore, the round shape of the battery meant it could quite easily fit inside the sock and be comfortable around the patient's foot.

The energy consumption for 10 hours for all the components is summarised in Table 6.1.

Table 6.1: Energy consumption for 10 hours

Component	**Energy consumption for 10 hours (Wh)**
Accelerometer	0.007
2 PPGs	3.3
EDA	0.5
ESP32	6.5
Vibrator module	≈ 0
Total	**10.3**

6.5. Sock Prototype

Once everything was ready to be implemented in the sock, the sock, sensors, ESP32, and vibration module were put together. This leads to the system shown in Figure 6.4.

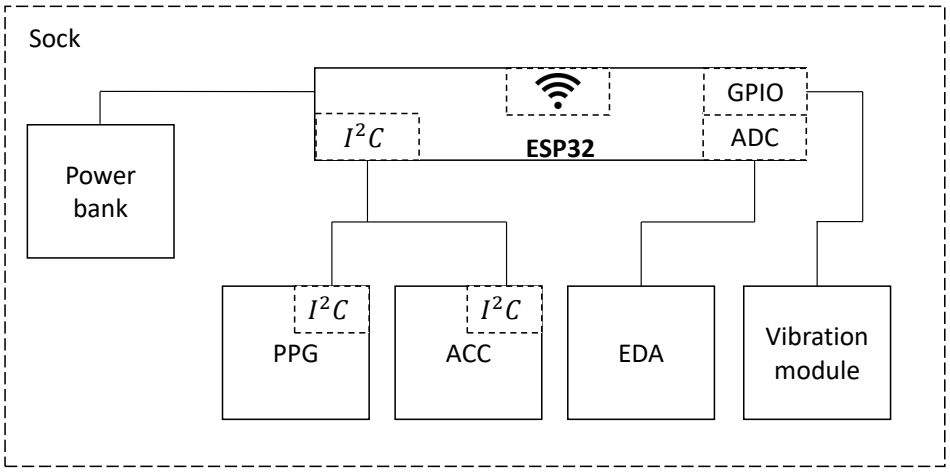

Figure 6.4: Prototype system

The final prototype is shown in Figures 6.5 and 6.6.

Figure 6.5: The sock

Figure 6.6: The sock attached to the power bank

Improvements Made

After performing initial tests with the socks, the PPG sensors showed a signal indicating a heart rate of around 80 BPM. Initially, it was thought that this was fine, but it remained at 80 BPM after doing some jumping jacks, while it was expected to increase. Investigating this problem, it turned out to be the case that the two PPG sensors interfered with each other, resulting in a heart rate artefact of 80 BPM. The solution to this was to switch off one PPG sensor, and only use one PPG sensor at a time.

7

Data Collection and Testing

An important part of the project is to collect data from patients. This collected data is used by the software subgroup to develop an algorithm.

The methodology for data collection is as follows: the sock is powered by the power bank and sends all the data wirelessly to a PC. The PC then stores the data in a CSV file. This methodology is different than that of the desired final product, as the final product will not send a lot of data, but will process it locally. The ESP32, however, does not have enough storage capacity to store all the sensor data for one night. This is why the choice was made to send all the sensor data to a PC and store it there.

7.1. Communication

Bluetooth Low Energy was the first choice for the communication protocol between the ESP32 and the PC as a first step in meeting requirement A.6. This protocol uses less power than classic Bluetooth, but in return has a lower data rate. A more detailed comparison can be found in Table 7.1. The ESP32 microcontrollers on the Momo BedSense also use Bluetooth Low Energy. As it is the goal to communicate with the BedSense, the choice was made to implement wireless data communication using Bluetooth Low Energy (BLE). For the final product, the ESP32 should only receive restlessness data from BedSense which should fit in a single byte.

The following table compares some functional characteristics of both Bluetooth Classic and BLE [43]:

Table 7.1: Comparison between BLC and BLE [43]

	Bluetooth Classic	**Bluetooth Low Energy**
Channel scheme	79 channels	40 channels
Output power	100 mW	10 mW
Latency	100 ms	6 ms
Time to send data	100 ms	3 ms
Raw data rate	1 Mb/s - 3 Mb/s	1 Mb/s
Throughput	0.7 Mb/s - 2.1 Mb/s	0.27 Mb/s
Max range	≈ 100 m	≈ 100 m

7.1.1. BLE: Sock and the BedSense

The sock will act as a BLE client which controls the communication, while the BedSense will act as a server which sends data to the client when requested. To communicate between the Sock and the BedSense, a connection needs to be established. When the server is ready to connect, it broadcasts

packets over a wide frequency band to let clients know that they are ready to connect. An advertising packet contains for example a service UUID, device name, and target address. A service is a collection of characteristics that can hold data and a descriptor that provides additional information about the data. In the most basic setting, the BLE server (BedSense) will have a single service with a single characteristic containing restlessness information. When the client receives an advertising packet, it can request a connection with the server and ask the server to notify it when there is new data available in a specific characteristic. Once the client connects with the server, the server will send the client an update whenever the server has new data ready.

7.1.2. WiFi: Sock and the PC

Since the throughput of BLE is too low for the amount of raw data that is gathered, it was decided to use WiFi to collect measurement data of the sensors. Specifically, the UDP protocol was used, since it prioritizes speed and efficiency, contrary to the TCP protocol which prioritizes reliability. For our purposes, however, both methods would suffice.

Upon starting data collecting, the sock and PC initiate a connection with a specified network and request an IP using the DNS protocol. After connecting to the local network, the PC sends a start packet containing the IP address of the laptop to the sock, using the IP of the sock obtained at the connection request. When the sock receives the start packet from the PC, it stores the PC's IP and starts collecting data from the sensors. When new data is available in the sock, it sends a packet containing the sensor data to the PC. Upon the arrival of the packet, the PC sends a packet back again containing its own IP. The continuous sending of packets ensures a stable connection between the sock and PC, even when the PC or sock changes the IP address. The composition of the data section of the UDP packets is shown in Figure 7.1.

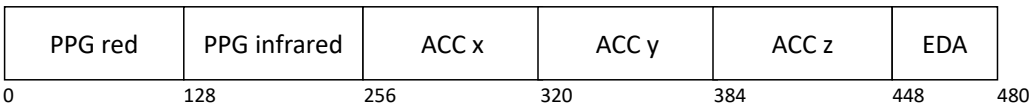

Figure 7.1: Packet overview with the number of bytes

7.2. Data Collection and Testing

Data collection was first performed on the members of the hardware subgroup before an attempt was made to collect data on an RBD patient. Figure 7.2 shows the entire data collection setup. The raw collected data was sent to the PC, where it can be processed by the software subgroup.

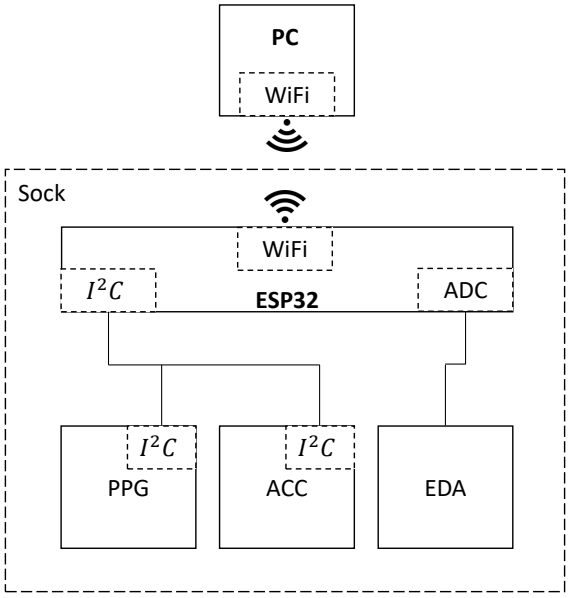

Figure 7.2: Data collection setup

The system was comfortable to wear according to the member of the hardware group. However, on the first night, there were interruptions in the data collection. It seemed that these were interruptions in the power supply from the power bank. After restarting the system the data collection worked again for some time. On the second night, the power bank was replaced by a 5 V mobile phone charger. Once again, there was an interruption in the data collection. This ruled out power supply problems as the cause. The issue turned out to be that the PC changed its IP address. As a result, the ESP32 was sending sensor data to the old IP address of the laptop, while the laptop had a new IP address. After this problem had been identified, it was solved by adjusting the source code.

Figure 7.3 shows the data of the accelerometer in x-, y-, and z-direction during sleep. It can be concluded from this figure that the person changed sides at around 7750 seconds after bedtime, and moved a bit at around 7468 seconds. A close-up of this peak can be found in Figure E.4 in Appendix E.

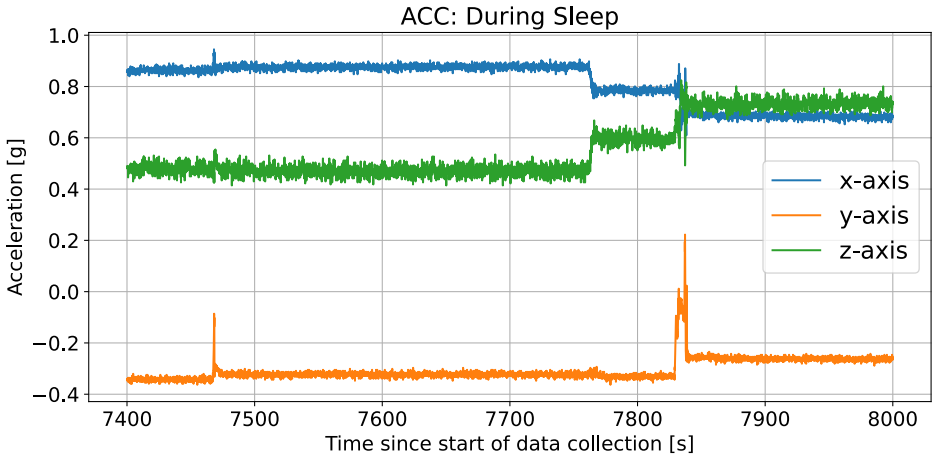

Figure 7.3: Data from the accelerometer while sleeping

Figure 7.4 shows the raw data from the PPG sensor. The heart rate can be determined to be around 60 BPM.

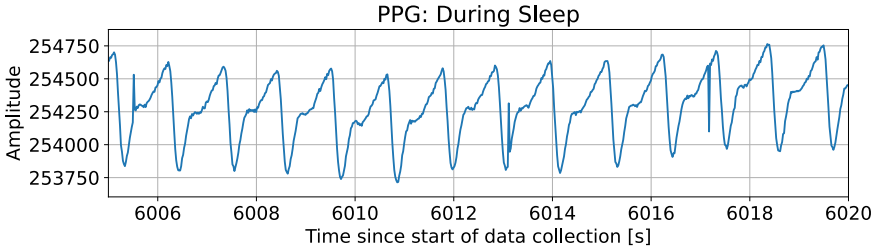

Figure 7.4: Data from the PPG sensor while sleeping

Figure 7.5 shows the measured EDA signal. There is a lot of high-frequency noise present in the signal. The peaks and troughs in this figure are the phasic components of the EDA signal, which is associated with short-term events that are invoked by environmental stimuli, such as sound or smell [44].

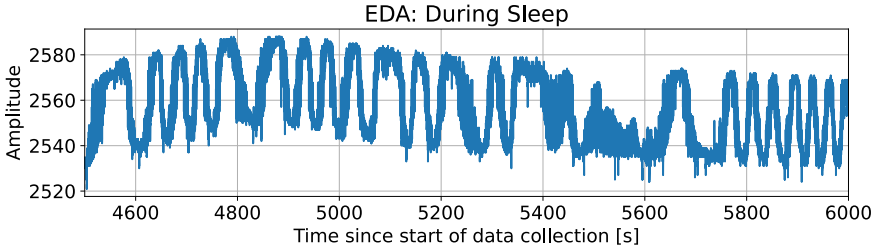

Figure 7.5: Data from the EDA sensor while sleeping

Data collection while moving

The quality of EDA and PPG signal is quite sensitive to the exact placement of the sensor, especially for the PPG sensor. Moreover, if the sensor is not tightly secured on the body or if there is movement, the sensor data may no longer display the heart rate. This subsection will explore the influence of a moving foot on the sensor output.

Figure 7.6 shows the accelerometer data when raising the foot. Around $t = 103.5$ seconds, the test subject started to raise their foot.

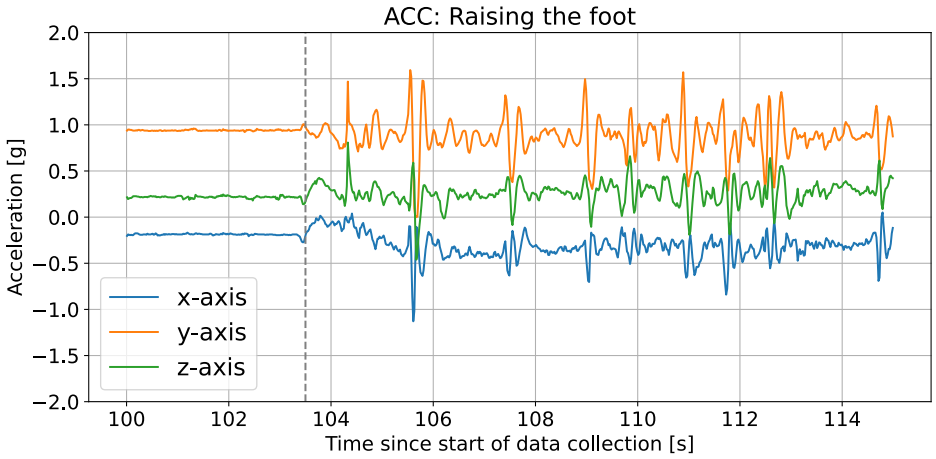

Figure 7.6: Data from the accelerometer while raising the foot

Figure 7.7 shows the PPG data when raising the foot. From this figure, it can be concluded that the desired signal is not present anymore. Heart rate extraction from this signal is not possible.

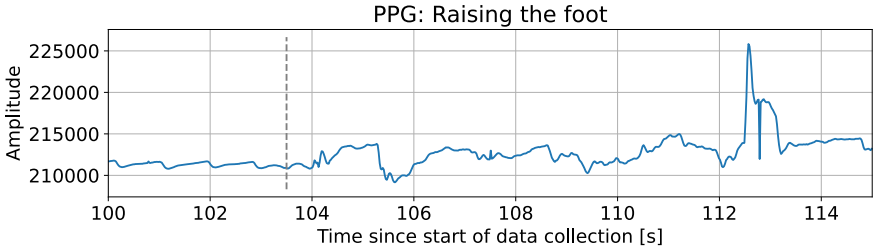

Figure 7.7: Data from the PPG sensor while raising the foot

Figure 7.8 shows the EDA data when raising the foot. It can again be concluded that the desired signal is not present in this data.

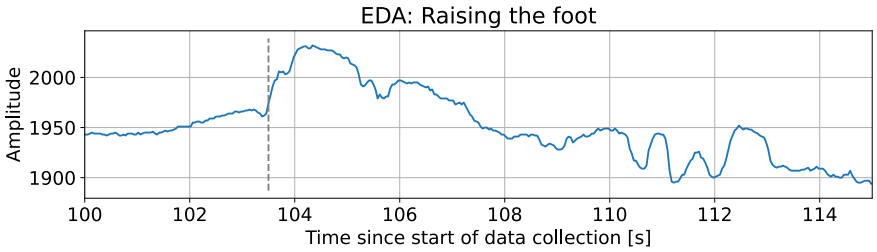

Figure 7.8: Data from the EDA sensor while raising the foot

Furthermore, Appendix E includes the results of data collection while moving the toes. These results again show heavily distorted EDA and PPG signals. This is however not an issue. The system should detect an RBD episode in an early stage, and the movement of the legs is already a late stage of an episode. The data that has been collected before the leg movement is more important than the information on the leg movement.

RBD Patient Data Collection

Data has been collected on an RBD patient. These results are elaborated on in Appendix A.

Vibration module testing

The vibration module has been tested on the members of the hardware subgroup, and the vibration module seems to vibrate hard enough to wake someone.

8

Conclusion and Discussion

8.1. Discussion

To collect data, the sock system was first applied to a member of the hardware group. While the hardware group member is not an RBD patient, data was still obtained. This data is useful since it further supports the proof of concept. Firstly, as mentioned before, Figure 7.3 shows the data of the accelerometer in the x-, y-, and z-direction. This graph shows the hardware group member moving in the night during sleep. This means that if the RBD patient were to move repeatedly and rapidly, the software group would be able to detect a potential episode.

Moreover, Figure 7.4 shows the raw data from the PPG sensor. From the graph, it is possible to extract the heart rate by observing peak-to-peak changes. Furthermore, a low-frequency amplitude change can be observed from the raw data of the PPG sensors. However, this is possibly due to the change in pressure between the sensor and the foot, which causes a change in the intensity of the reflected light. This can be filtered out, where the remaining data results in the heart rate. Changes in heart rate could indicate a possible early onset episode, thus fulfilling requirement A.1.

Furthermore, for the EDA sensor, the collected data can be seen in Figure 7.5. As explained, the peaks in the graph are associated with short-term events that are invoked by environmental stimuli. This means that if the patient experiences a stressful dream, those stimuli could be recognized by the peaks. This would allow for possible early episode detection and would fulfil requirement A.1.

The output of the PPG and EDA sensors are heavily distorted by moving the foot, as can be seen in Figure 7.7 and 7.8, respectively. This does not cause a problem, as the goal is to detect an episode in an early stage, and leg movement is not considered to be an early stage in this research. This behavior does therefore not conflict with requirements A.1 and A.2.

The vibration module that was attached can sufficiently vibrate since the power of vibration is more than comparable to a smartwatch. However, experiments have not been performed to assess if the vibration module can lift someone from REM sleep into a lighter sleep stage.

Lastly, data was obtained from an RBD patient, as shown in Appendix A. There was a minor RBD event during the data collection, and data from this has been recorded.

8.2. Conclusion

In conclusion, the goal of this project was partly achieved, since a proof of concept for non-invasive monitoring and prevention of RBD was demonstrated. This was accomplished by first selecting the sensors that met requirements A.1, A.2, A.7, and A.8, which are EDA, PPG, and accelerometer. To extract data from these sensors, the ESP32 microcontroller was selected, for which the I^2C protocol was used to read out the PPG and accelerometer, while for the EDA sensor the ADC of the ESP32 was used. To complete the system, a battery module and vibration module were chosen to wake

up the patient, in accordance with requirements A.4, A.5, and A.7. With the various components, it was established that the system would fit inside a sock, making it comfortable and non-invasive (requirements B.1 and B.3), and complementary to the BedSense (requirement A.2). Using this sock system, a WiFi communication protocol was set up to transmit the information from the sensors to the computer for processing, meeting requirement A.4. No Bluetooth Low Energy connection was established with the Momo BedSense, and requirement A.6 is therefore not met. Testing this data firstly on non-RBD patients allowed for measurements from which RBD episodes could potentially be detected. Data was collected from the RBD patient. This was only data collection and the algorithm on the sock was not running and hence, requirement A.5 has not been fulfilled. Therefore, more research and data collection needs to be done with the sock to potentially further develop it for use on patients. Moreover, the absence of testing the waking module on the patient is also the reason for the goal not being fully fulfilled. Nevertheless, the sock system theoretically has shown capabilities of being a non-invasive solution for patients suffering from RBD.

8.3. Future Work

The results of this designed hardware system seem promising, the sensor data is read out and wirelessly stored on a PC where the software group can process the data. A fully developed final product, however, would need to have some improvements.

Data collection

Firstly, more data collection has to be performed on an RBD patient. To fully test the system, the vibration module should be tested for bringing a patient to a lighter sleep stage. Moreover, for future data collection, it would be better to store the data on an SD card, as the data collection via WiFi can be a hassle to set up correctly.

EDA sensor

It would be wise to implement an EDA sensor that comes with a built-in sampler, as the PPG sensor and the accelerometer do. Currently, the EDA sensor outputs an analog signal, that the analog-to-digital converter of the ESP32 converts into a digital signal. Using an EDA sensor with a built-in sampler would decrease the computational power that is needed in the microcontroller. It would leave more computational power for the algorithm developed by the software subgroup. It would also improve the flexibility and the readability of the code, as the sensor would be read out using the I^2C communication protocol. This protocol is currently already used to read out the PPG sensor and the accelerometer. Replacing the EDA sensor, which is sampled by the ESP32 ADC, with a sensor having its own built-in sampler, eliminates the noise of the ESP32 ADC.

Automatic heartbeat screening

Another improvement that could be made is to implement an automatic screening for a good heartbeat signal. The PPG sensors should be placed at a quite specific spot to obtain a proper PPG signal. An algorithm could be implemented that automatically determines whether the PPG sensor is placed at the correct spot or not. When the first PPG sensor is not at the right position, it switches to the second PPG sensor and when that sensor also does not measure a correct PPG signal, it can give an indication to the user to move the sock, by for instance turning on an LED. This would make the system more user-friendly.

Sock Improvements

The PPG and EDA sensors in the current prototype are still able to move a bit. This results in distorted signals when moving the foot. This could be improved by having a tighter sock, where the sensors are not free to move anymore. This could however result in a less comfortable system.

Battery system

Currently, the system is powered by a power bank that is quite large and makes the sock not as comfortable to wear as without the power bank. This could be improved by implementing a rechargeable battery with just a bit more capacity than needed to run the system for an entire night. Together with this battery, a battery circuit can be implemented. This circuit then takes care of charging the battery with an appropriate voltage and could implement adaptive charging, which will prolong the lifetime of the battery.

Space efficiency
In the current prototype, there is an ESP32 development board, which is not very space efficient. There is a lot of functionality on the ESP32 development board that is not being used currently. The current prototype also uses off-the-shelf sensor boards. When only the processor of the ESP32 is used and the sensor circuits are placed, routed, and printed together on one single printed circuit board, this will decrease the used space and make the entire system smaller and more comfortable to wear.

Hygiene
The present prototype is not washable, meaning that it is not very hygienic to wear for multiple nights. It would be ideal to manufacture two socks, in which one set of sensors can be placed. After a few nights, this set of sensors can be placed in the other sock and the other sock can be washed.

Communication with the BedSense
For the sensor system to work effectively together with the data collected from the Momo BedSense, communication needs to be established between the sensor system and the BedSense. This communication will be based on the Bluetooth Low Energy communication protocol. The BedSense will send data about restlessness to the sensor system. The sensor system will then combine the restlessness data from the BedSense with its own data and make a decision about waking the patient. The sensor system can, of course, also make a decision independent of the BedSense when this is needed.

Dynamic scaling
Currently, the PPG and accelerometer sensors have static scaling, however on-the-run sensor scaling, and sensitivity settings can be implemented. To implement dynamic scaling, at the start, the measurement scale should be as low as possible. When the sensor reading reaches its maximum value, the scale should be lowered by one stage until the signal is lower than the maximum value. To implement dynamic sensitivity for the PPG, a feedback loop could be implemented. When starting a measurement the PPG sensitivity should be as low as possible. When the data is processed and the heart rate is not found, the sensitivity should increase by a stage until a heart rate can be detected. When the peak-to-peak detection is above a certain rate the sensitivity can be lowered again. It is important to limit the sensitivity since it is directly proportional to the power consumption of the PPG sensor, so limiting it to a minimum ensures efficient design.

Synchronous sampling
The PPG sensor and accelerometer are asynchronously sampled. Both sensors have their own clock and sample independently of each other. This is not optimal since the sensor buffers can fill up at slightly different times. Especially for the PPG sensors, asynchronous sampling is a problem since this can cause light interference between multiple sensors. The light of one sensor can be sensed by another sensor creating noise. This issue would be solved by using a single clock to synchronize all sensors or using a dedicated signal to notify the sensors when to sample new data.

References

[1] C. H. Schenck, B. Högl, and A. Videnovic, *Rapid-Eye-Movement Sleep Behavior Disorder*, 1st ed. Cham: Springer, 2019.

[2] A. K. Patel, V. Reddy, K. R. Shumway, and J. F. Araujo, *Physiology, Sleep Stages*. StatPearls Publishing, 2022.

[3] E. Suni and N. Vyas, *Stages of sleep: What happens in a sleep cycle*, May 2023. [Online]. Available: `https://www.sleepfoundation.org/stages-of-sleep`.

[4] I. Arnulf, "REM sleep behavior disorder: Motor manifestations and pathophysiology," *Movement Disorders*, vol. 27, no. 6, pp. 677–689, May 2012.

[5] "REM sleep behavior disorder," *Clinical Neurophysiology*, vol. 111, S136–S140, 2000, Sleep and Epilepsy Supplement, ISSN: 1388-2457.

[6] J. F. Gagnon *et al.*, "REM sleep behavior disorder and REM sleep without atonia in Parkinson's disease," *Neurology*, vol. 59, no. 4, pp. 585–589, 2002.

[7] J. V. Rundo and R. Downey, "Chapter 25 - Polysomnography," in *Clinical Neurophysiology: Basis and Technical Aspects*, ser. Handbook of Clinical Neurology, K. H. Levin and P. Chauvel, Eds., vol. 160, Elsevier, 2019, pp. 381–392.

[8] E. K. St Louis and B. F. Boeve, "REM Sleep Behavior Disorder: Diagnosis, Clinical Implications, and Future Directions," *Mayo Clinic Proceedings*, vol. 92, no. 11, pp. 1723–1736, 2017.

[9] B. F. Boeve, "REM sleep behavior disorder," *Annals of the New York Academy of Sciences*, vol. 1184, no. 1, pp. 15–54, 2010.

[10] R. N. Aurora *et al.*, "Best Practice Guide for the Treatment of REM Sleep Behavior Disorder (RBD)," *Journal of Clinical Sleep Medicine*, vol. 06, no. 01, pp. 85–95, 2010.

[11] A. Roguski, D. Rayment, A. L. Whone, M. W. Jones, and M. Rolinski, "A Neurologist's Guide to REM Sleep Behavior Disorder," *Frontiers in Neurology*, vol. 11, Jul. 2020.

[12] F. Raschellà, S. Scafa, A. Puiatti, E. M. Moraud, and P.-L. Ratti, "RBDAct: Home screening of REM sleep behaviour disorder based on wrist actigraphy in Parkinson's patients," *Annals of Neurology*, Jan. 2022.

[13] M. Louter, J. B. A. M. Arends, B. R. Bloem, and S. Overeem, "Actigraphy as a diagnostic aid for REM sleep behavior disorder in Parkinson's disease," *BMC Neurol*, vol. 14, no. 76, Apr. 2014.

[14] N. Cooray *et al.*, "Enabling Automated REM Sleep Behaviour Disorder Detection," *The 40th International Conference of the IEEE, EMBC*, Jul. 2018.

[15] A. B. Neikrug and S. Ancoli-Israel, "Diagnostic tools for REM sleep behavior disorder," *Sleep Medicine Reviews*, vol. 16, no. 5, pp. 415–429, Oct. 2012.

[16] N. Cooray *et al.*, "Proof of concept: Screening for REM sleep behaviour disorder with a minimal set of sensors," *Clinical Neurophysiology*, vol. 132, no. 4, pp. 904–913, Apr. 2021.

[17] K. T. Johnson and R. W. Picard, "Advancing Neuroscience through Wearable Devices," *Neuron*, vol. 108, no. 1, pp. 8–12, Oct. 2020.

[18] M. J. Howell, P. A. Arneson, and C. H. Schenck, "A Novel Therapy for REM Sleep Behavior Disorder (RBD)," *Journal of Clinical Sleep Medicine*, vol. 7, no. 6, pp. 639–644, Dec. 2011.

[19] X. Yang *et al.*, "Monitoring of Patients Suffering From REM Sleep Behavior Disorder," *IEEE Journal of Electromagnetics, RF and Microwaves in Medicine and Biology*, vol. 2, no. 2, pp. 138–143, Jun. 2018.

[20] Y. Farraj *et al.*, "Measuring pupil size and light response through closed eyelids," *Biomedical Optics Express*, vol. 12, no. 10, pp. 6485–6495, Oct. 2021.

[21] K. Kandiyoor, P. Kremers, and J. R. Post, “Real-Time Detection of Restlessness Caused by Rapid Eye Movement Sleep Behaviour Disorder,” 2023, The thesis of the software subgroup.

[22] M. Manconi *et al.*, “Time structure analysis of leg movements during sleep in REM sleep behavior disorder,” *Sleep*, vol. 30, no. 12, pp. 1779–1785, Dec. 2007.

[23] Zorggroep Oude en Nieuwe Land. “Slimme sensor verbetert de nachtrust bij bewoners ’t Kompas.” (2019), [Online]. Available: `https://zorggroep-onl.nl/slimme-sensor-verbetert-de-nachtrust-bij-bewoners-t-kompas` (visited on 06/01/2023).

[24] B. Frauscher *et al.*, “Normative EMG Values during REM Sleep for the Diagnosis of REM Sleep Behavior Disorder,” *Sleep*, vol. 35, 6 Jun. 2012.

[25] R. Zangróniz, A. Martínez-Rodrigo, J. M. Pastor, M. T. López, and A. Fernández-Caballero, “Electrodermal Activity Sensor for Classification of Calm/Distress Condition,” *Sensors*, vol. 17, 10 Oct. 2017.

[26] M. L. Fantini, M. Michaud, N. Gosselin, G. Lavigne, and J. Montplaisir, “Periodic leg movements in REM sleep behavior disorder and related autonomic and EEG activation,” *Neurology*, vol. 59, pp. 1889–1894, 12 Dec. 2002.

[27] P. A. Lanfranchi, L. Fradette, J. F. Gagnon, R. Colombo, and J. Montplaisir, “Cardiac Autonomic Regulation During Sleep in Idiopathic REM Sleep Behavior Disorder,” *Sleep*, vol. 30, 8 Aug. 2007.

[28] J. de Moraes *et al.*, “Advances in Photopletysmography Signal Analysis for Biomedical Applications,” *Sensors*, vol. 18, Jun. 2018.

[29] M. Manconi *et al.*, “Time structure analysis of leg movements during sleep in REM sleep behavior disorder,” *Sleep*, vol. 30, pp. 1779–1785, 12 Dec. 2007.

[30] Z. Zhu and Q. Ji, “Robust real-time eye detection and tracking under variable lighting conditions and various face orientations,” *Computer Vision and Image Understanding*, vol. 98, pp. 124–154, 1 Apr. 2005.

[31] I. S. Gopal and G. G. Haddad, “Automatic detection of eye movements in REM sleep using the electrooculogram,” *The American journal of physiology*, vol. 241, pp. 217–221, 3 1981.

[32] *MAX30102 datasheet*, Maxim Integrated, Oct. 2018. [Online]. Available: `https://www.analog.com/media/en/technical-documentation/data-sheets/max30102.pdf`.

[33] *LIS3DH datasheet*, STMicroelectronics, Dec. 2016. [Online]. Available: `https://www.st.com/resource/en/datasheet/cd00274221.pdf`.

[34] British Heart Foundation. “What is a normal pulse rate?” (2021), [Online]. Available: `https://www.bhf.org.uk/informationsupport/heart-matters-magazine/medical/ask-the-experts/pulse-rate` (visited on 05/24/2023).

[35] H. F. Posada-Quintero and K. H. Chon, “Innovations in Electrodermal Activity Data Collection and Signal Processing: A Systematic Review,” *Sensors*, vol. 20, no. 2, 2020, ISSN: 1424-8220.

[36] A. Greco *et al.*, “Acute Stress State Classification Based on Electrodermal Activity Modeling,” *IEEE Transactions on Affective Computing*, vol. 14, no. 1, pp. 788–799, 2023.

[37] D. Bruck and I. Thomas, “Comparison of the Effectiveness of Different Fire Notification Signals in Sleeping Older Adults,” *Fire Technology*, vol. 44, Aug. 2007.

[38] G. Lavigne *et al.*, “Sleep arousal response to experimental thermal stimulation during sleep in human subjects free of pain and sleep problems,” *PAIN®*, vol. 84, pp. 283–290, 2-3 Feb. 2000.

[39] D. Castaneda, A. Esparza, M. Ghamari, C. Soltanpur, and H. Nazeran, “A review on wearable photoplethysmography sensors and their potential future applications in health care,” *International Journal of Biosensors and Bioelectronics*, vol. 4, Aug. 2018.

[40] M.-B. Hossain, Y. Kong, H. F. Posada-Quintero, and K. H. Chon, “Comparison of Electrodermal Activity from Multiple Body Locations Based on Standard EDA Indices’ Quality and Robustness against Motion Artifact,” *Sensors*, vol. 22, May 2022.

[41] A. M. Carek and O. T. Inan, “Robust Sensing of Distal Pulse Waveforms on a Modified Weighing Scale for Ubiquitous Pulse Transit Time Measurement,” *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, Aug. 2017.

[42] *GSR Sensor datasheet*, Grove, Oct. 2018. [Online]. Available: https://wiki.seeedstudio.com/Grove-GSR_Sensor/.

[43] Bestwireless, *Bluetooth Low Energy*, https://bestwirelessbluetoothheadphones.com/wiki/bluetooth-low-energy/, Accessed June 8, 2023.

[44] H. F. Posada-Quintero and K. H. Chon, "Innovations in Electrodermal Activity Data Collection and Signal Processing: A Systematic Review," *Sensors*, vol. 20, Jan. 2020.

A

Data Collection on RBD Patient

Measurements have been taken with the sock on an RBD patient. This appendix consists of the resulting sock and BedSense measurements. The period from 02:25 to 02:40 will be discussed, because the patient noticed some slight arm movements at 02:29 and leg movements at 02:35. The PE measurements seen in Figure A.1 show vibrations around 02:29, which can be explained as the arm movements. The vibrations after 02:35 could be caused by the leg movements. The FSR data, shown in Figure A.2, indicates a fluctuating pressure around 02:36.

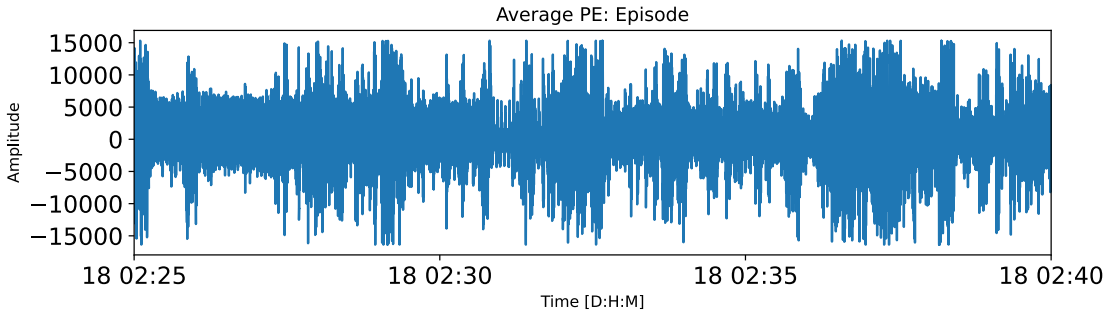

Figure A.1: PE data from the BedSense

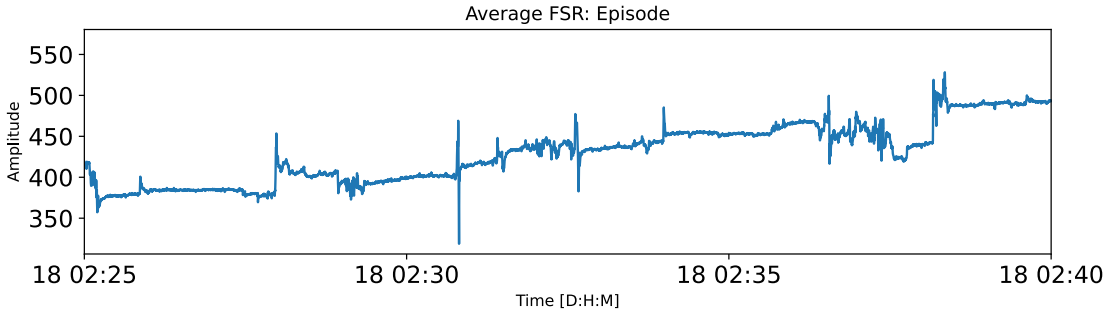

Figure A.2: FSR data from the BedSense

Looking at the accelerometer data from the sock in Figure A.3, both the leg and arm movements can be seen. There is a small peak around 02:29, concurrent with the arm movements. After 02:35 a bigger peak occurs, which shows leg movement. When comparing the accelerometer data with the BedSense data seen in Figures A.1 and A.2, it can be concluded that the sock does indeed provide more information on leg movement.

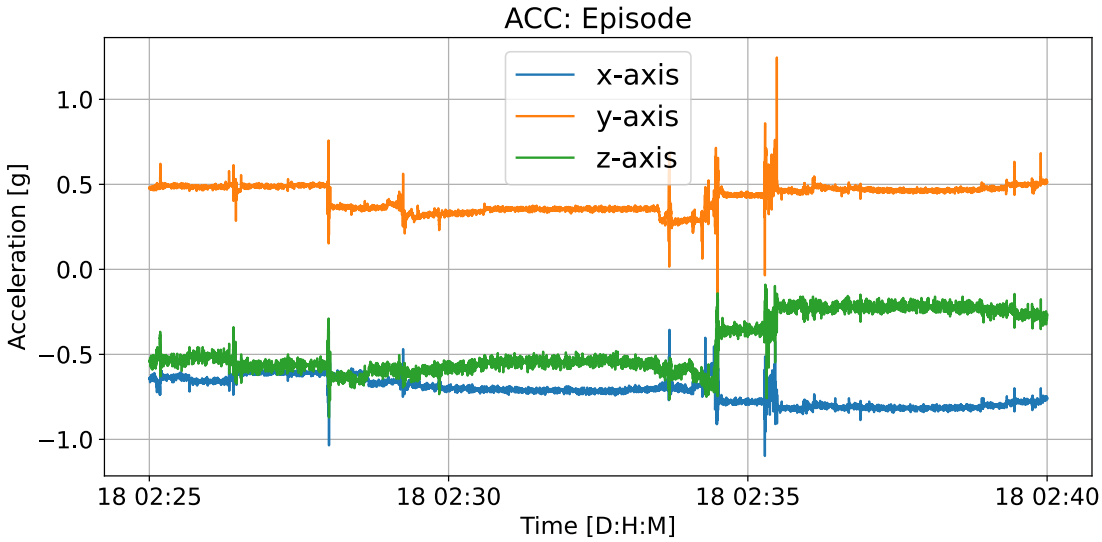

Figure A.3: Data from the accelerometer

In Figure A.4, the raw PPG data is presented. This has been processed and turned into an estimated heart rate, shown in Figure A.5. It shows no abnormalities around the time of the arm movements, but at the same time of the leg movement the heart rate does seem to be higher for a while. After the leg movements the estimated heart rate reaches around 140 bpm, this is likely to be a false outlier.

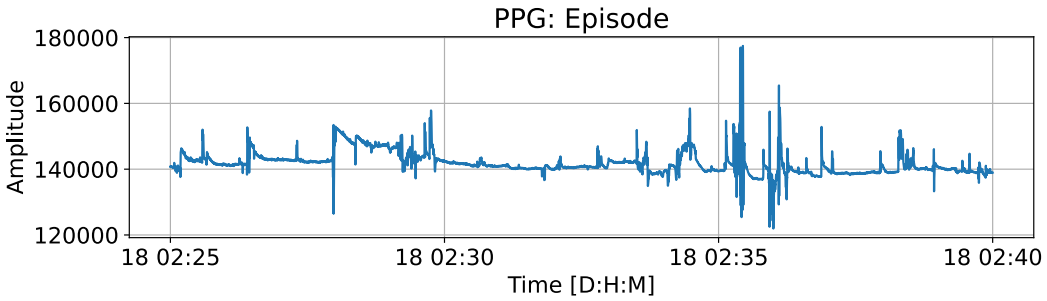

Figure A.4: Data from the PPG sensor

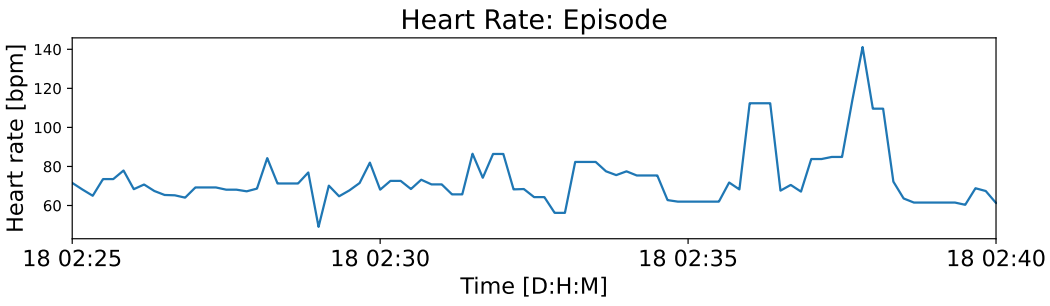

Figure A.5: Extracted heart rate

The EDA data can be seen in Figure A.6. At the same time of the leg movement the value of the raw EDA data drops suddenly, this could be caused by the sensors being moved. But after EDA values gradually start increasing, which could be an indicator of stress.

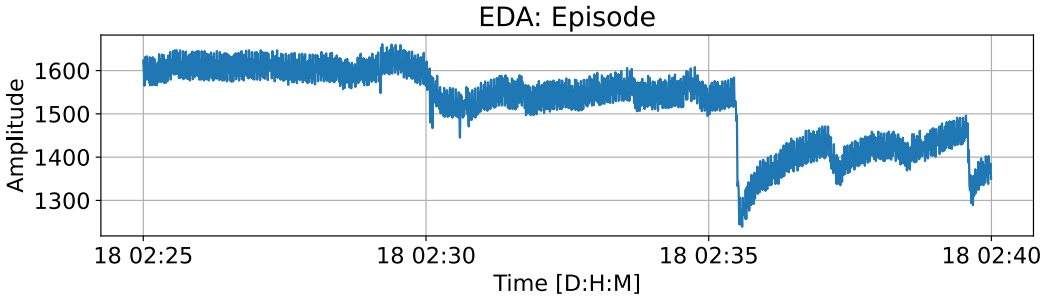

Figure A.6: Data from the EDA sensor

B

RBD Diagnostic Criteria

B.1. International Classification of Sleep Disorders, Third Edition

Table B.1: *International Classification of Sleep Disorders, Third Edition* (ICSD-3). REM Sleep Behavior Disorder Diagnostic Criteria [1].

Criteria A-D must be met	
A.	Repeated episodes of sleep-related vocalization and/or complex motor behaviors
B.	These behaviors are documented by polysomnography to occur during REM sleep or, based on clinical history of dream enactment, are presumed to occur during REM sleep
C.	Polysomnographic recording demonstrates REM sleep without atonia (RWA)
D.	The disturbance is not better explained by another sleep disorder, mental disorder, medication or substance use

B.2. The American Academy of Sleep Medicine Manual for the Scoring of Sleep and Associated Events

Table B.2: The American Academy of Sleep Medicine (AASM) Manual for the Scoring of Sleep and Associated Events. Scoring Polysomnographic Features of REM Sleep Behavior Disorder (RBD) [1].

1.	Score in accordance with the following definitions
Sustained muscle activity (tonic activity) in REM sleep: An epoch of REM sleep with at least 50% of the duration of the epoch having a chin EMG amplitude greater than the minimum amplitude demonstrated in NREM sleep	
Excessive transient muscle activity (phasic activity) in REM sleep: In a 30 s epoch of REM sleep divided into ten sequential 3 s mini-epochs, at least five (50%) of the mini-epochs contain bursts of transient muscle activity. In RBD, excessive transient muscle activity bursts are 0.1-5.0 s in duration and at least four times as high in amplitude as the background EMG activity	
2.	The polysomnographic characteristics of RBD are characterized by EITHER or BOTH of the following features:
(a)	Sustained muscle activity in REM sleep in the chin EMG
(b)	Excessive transient muscle activity during REM in the chin or limb EMG

B.3. Diagnostic and Statistical Manual of Mental Disorders

Table B.3: *Diagnostic and Statistical Manual of Mental Disorders, Fifth Edition* (DSM-5). Rapid Eye Movement sleep Behavior Disorder Diagnostic Criteria (Only the Core Features are Reported) [1].

A.	Repeated episodes of arousal during sleep associated with vocalization and/or complex motor behaviors
B.	These behaviors arise during rapid eye movement (REM) sleep and therefore usually occur more than 90 min after sleep onset, are more frequent during the later portions of the sleep period and uncommonly occur during daytime naps
C.	Upon awakening from these periods, the individual is completely awake, alert and not confused or disoriented
D.	Either of the following: 1. REM sleep without atonia in polysomnographic recording 2. A history suggestive of REM sleep behavior disorder and established synucleinopathy diagnosis (e.g. Parkinson's disease, multiple system atrophy)

C

Sensor Characteristics

From the sensor comparison, it was concluded that the following sensors will be used: EDA sensor, PPG sensor, and an accelerometer.

EDA Sensor

The exact choice of sensor for EDA was chosen to be Grove (galvanic skin response) GSR. It measures the resistance of the skin which has a strong correlation to emotional stress end can cause stimulation to the sympathetic nervous system which is made visible by low-varying AC components of the sweat response. The sensor has the following key specifications [42]:

- Operating voltage 3.3V/5V
- Analog Output Signal
- 2 Electrodes

PPG Sensor

For the PPG sensor, the MAX30102 from Analog Devices was chosen. The sensor has the following key specifications [32]:

- Programmable Sample Rate and LED Current for Power Savings
- Low-Power Heart-Rate Monitor (< 1mW)
- Embedded 32 levels of 18-bit data output FIFO
- Integrated sample averaging
- 18 bit ADCs
- 2 LED channels: 1 x red-light & 1x IR-light
- Dynamic scaling capabilities

The MAX30102 module is able to measure both pulse oximetry and heart rate. It mainly consists of LED and photodetectors. The sensor can be read out using an I^2C interface.

Accelerometer

The accelerometer that was chosen is the STMicroelectronics LIS3DH. The sensor has the following key specifications [33]:

- ±2g/±4g/±8g/±16g dynamically selectable scale
- Ultra-low-power mode consumption down to 2 µA
- Embedded 32 levels of 16-bit data output FIFO
- 8 to 12-bit ADCs

The LIS3DH is a low-power three-axis linear accelerometer with a digital I^2C serial interface standard output. The LIS3DH has dynamically user-selectable full scales of ±2g/±4g/±8g/±16g and is capable of measuring accelerations with output data rates from 1 Hz to 5.3 kHz. It has an integrated 32-level first-in, first-out (FIFO) buffer allowing to store data in order to limit intervention by the host processor.

The following operating settings are used for this sensor:

- Sample frequency: 50 Hz
- Operating range: $\pm$ 2 g
- Sample size: 12 bit
- Resolution: 1 mg/bit

D

Sensor Readout

D.1. MAX30102

Programming

To control how the MAX30102 operates, it is necessary to program it by writing to a number of registers that store the operating settings of the device. Each register of the MAX30102 consists of an 8-bit word in memory. To write to those registers a series of I^2C operations need to be performed. These operations are specified later in this section. The registers written to at restart are listed in Table D.1.

Table D.1: Register settings for MAX30102

Name	**Address**	**Data**	**Default**
FIFO_CONFIG	0x08	0x70	0x00
MODE_CONFIG	0x09	0x03	0x00
SPO2_CONFIG	0x0A	0x4F	0x00
LED1_PA	0x0C	0x5A	0x00
LED2_PA	0x0D	0x5A	0x00

FIFO_CONFIG

- Sets sample averaging to 8 samples.
- Enable FIFO.
- Enables FIFO rollover.

MODE_CONFIG

- Enables SPO2 mode (enable both the IR-LED and RED-LED).

SPO2_CONFIG

- Sets ADC scale to 8192 nA.
- Sets sample rate to 400 Hz.
- Sets LED pulse with to 411 μs (18 bit resolution).

LED1_PA

- Sets RED-LED current level to 18 mA.

LED2_PA
- Sets IR-LED current level to 18 mA.

Register writing
To write to a specific register of the MAX30102, the microcontroller (master) performs the following operations. The master establishes a connection by sending a start condition, followed by the slave address of the MAX30102 (slave) appended with a 1 to specify a write operation (0b10101110 | 0b0). Upon obtaining its slave address, the MAX30102 (slave) emits an ACK signal. Consequently, the master sends a byte frame that consists of the registered address to write to and waits again for an ACK from the slave. After receiving the ACK from the slave, the master sends the data to be stored in the register. Once again, the slave sends an ACK upon receiving the new register data and the master responds with a stop condition, after which the bus is freed.

Register reading
To read the contents of specific registers of the MAX30102, the microcontroller (master) performs the following operations. The master establishes a connection by sending a start condition followed by the slave address of the MAX30102 (slave), appended with a 1 to specify a write operation (0b10101110 | 0b0). Following the receipt of its slave address, the MAX30102 (slave) emits an ACK signal. Next, the master sends a byte frame which consists of the register address to read from, and waits again for an ACK from the slave. Upon receiving the ACK from the slave, the master sends a restart condition followed by the slave address together with a read bit (0b10101110 | 0b1), to specify a read operation. Following this, the slave sends an ACK signal followed by the contents of the register specified. After the master receives the data from the slave, and the master decides to end the communication, it sends a NACK followed by a stop condition. However, by sending an ACK instead of a NACK, the slave sends the contents of the next register in memory to the master. The slave continues this process until the master sends a NACK followed by a stop condition after receiving the data.

FIFO-readout
The MAX30102 has a built-in FIFO (first-in-first-out) buffer. It has space for 32 samples per LED channel. When a new sample is ready, the sample is transferred to the buffer at the location of the value specified in the FIFO_WR register (0x04) and is automatically incremented by 1. Reading from the FIFO buffer is done by reading register FIFO_DATA (0x07) 6 times using the multi-read mode, retrieving 6 bytes consisting of the 3-byte data for each LED channel. Once a sample is read, the value in the FIFO_RD register (0x06) is automatically incremented by one. This sets the data in register FIFO_DATA to be the value located at the location of the FIFO write pointer in FIFO_WR. Once the write pointer reaches the end of the buffer, which is at address 31, it rolls back to 0 and overwrites the old data. The structure of the FIFO buffer is shown in Figure D.1.
It is important that the FIFO is read before it overwrites unread data to prevent data loss. This is done by at least reading all the data in the buffer with a frequency f_{read}, which is specified in Equation D.1.

$$f_{read} > \frac{f_s}{fifo\_size} \approx 1.56Hz \tag{D.1}$$

Where f_{read} is the frequency of reading all the data stored in the FIFO buffer, f_s is the frequency to which new data is generated and $fifo\_size$ is the size of the FIFO buffer.

D.2. LIS3DH
Programming
To control how the LIS3DH operates it is necessary to program it by writing to a number of registers that store the operating settings of the device. Like the MAX30102, each register of the LIS3DH consists of an 8-bit word in memory. To write to those registers, a series of I^2C operations need to be performed which are specified later in this chapter. The registers written to at restart are listed in Table D.2.

CTRL_REG1
- Sets sample frequency to 50 Hz.

Table D.2: Register settings for LIS3DH

Name	**Address**	**Data**	**Default**
CTRL_REG1_MASK	0x20	0x47	0x00
CTRL_REG4	0x23	0x08	0x00
CTRL_REG5	0x24	0x40	0x00
FIFO_CTRL_REG	0x2E	0x94	0x00

- Disables low power mode.
- Enable x, y, z data collection.

CTRL_REG4
- Measuring scale is $\pm 2g$.
- Enable High resolution (12-bit).

CTRL_REG5
- Enable FIFO.

FIFO_CTRL_REG
- FIFO mode (stream).

Register Writing

To write to a specific register for the LIS3DH the microcontroller (master) performs the following operations. The master establishes a connection by sending a start condition followed by the slave address of the LIS3DH (slave) append with a 1 to specify a write operation (0b00110010 | 0b0). Following the slave's receipt of its slave address, it sends an ACK signal. Next, the master sends the byte frame which consists of the register address to write to and waits again for an ACK from the slave. After the slave sends the ACK, the master sends the data to be stored in the register. Furthermore, the slave sends again an ACK upon receiving the new register data and the master responds with a stop condition to end free the bus.

Register Reading

To read the contents of specific registers for the LIS3DH the microcontroller (master) performs the following operations. The master establishes a connection by sending a start condition followed by the slave address of the LIS3DH (slave) appended with a 1 to specify a write operation (0b00110010 | 0b0). Subsequently, the slave receives its slave address and it sends an ACK signal. Next, the master sends the byte frame which consists of the register address to read from. The address is appended with a bit that enables or disables address auto-increment mode (EN/DIS) and waits again for an ACK from the slave. Subsequently the slave sends the ACK, the master sends a restart condition followed by the slave address together with a read bit (0b00110010 | 0b1), to specify a read operation. The slave sends an ACK signal followed by the contents of the register specified. After the master receives the data from the slave it sends an ACK or NACK. When the master decides to end the communication it sends a NACK followed by a stop condition. However, if address auto-increment mode was enabled and an ACK was sent instead of a NACK, the slave sends the contents of the next register in memory to the master until the master sends a NACK followed by a stop condition after receiving the data. When instead burst read mode is disabled, the slave will not auto-increment the register and resends its contents until the master initiates a stop sequence. After receiving the ACK from the slave, the master sends a restart condition, followed by the slave address and a read bit (0b00110010 followed by 0b1), to specify a read operation.

FIFO-readout

The LIS3DH also has a FIFO buffer of 32 samples. Each sample consists of 3x16-bit values corresponding to the measured acceleration in each of the axis. When a new sample is ready, the sample is transferred to the buffer and the data in FSS[4:0], located in register FIFO_SRC_REG (0x2f) is automatically incremented by one. FSS[4:0] resembles the number of unread samples inside the buffer. Reading from the FIFO is done by enabling address auto-increment mode, and then reading 6 bytes starting from address 0x28 and ending at address 0x2D. This results in reading the 2 bytes of acceleration data for each of the axes: once a sample is read, the value of FSS[4:0] is automatically decremented by one. The next oldest sample in the FIFO is then forwarded to the registers 0x28 to 0x2D, such that the next sample can be read. The structure of the FIFO buffer is shown in Figure D.2. Equation D.1 holds for reading out the FIFO data.

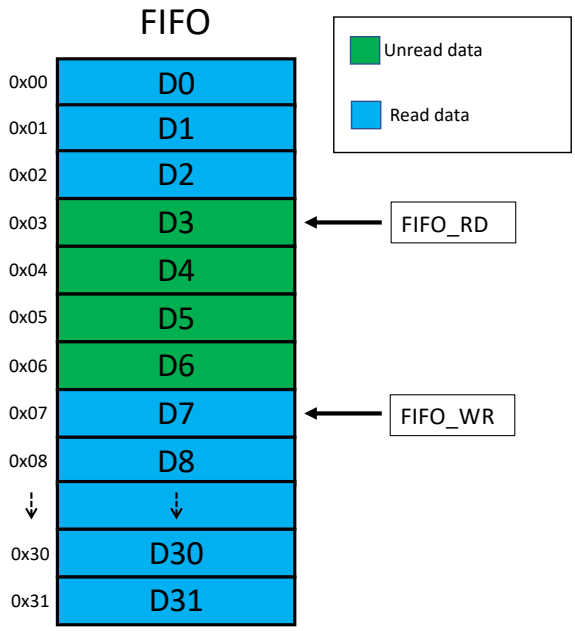

Figure D.1: Structure of the MAX30102 FIFO buffer

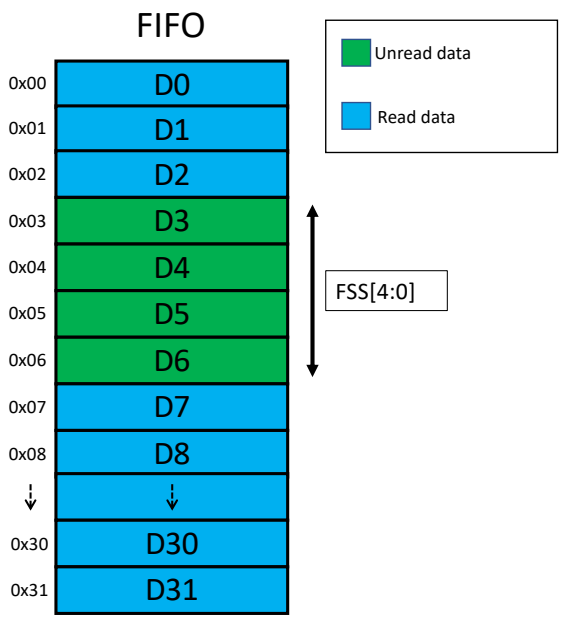

Figure D.2: Structure of the LIS3DH FIFO buffer

E

Results

Figure E.1 shows the accelerometer data when moving the toes.

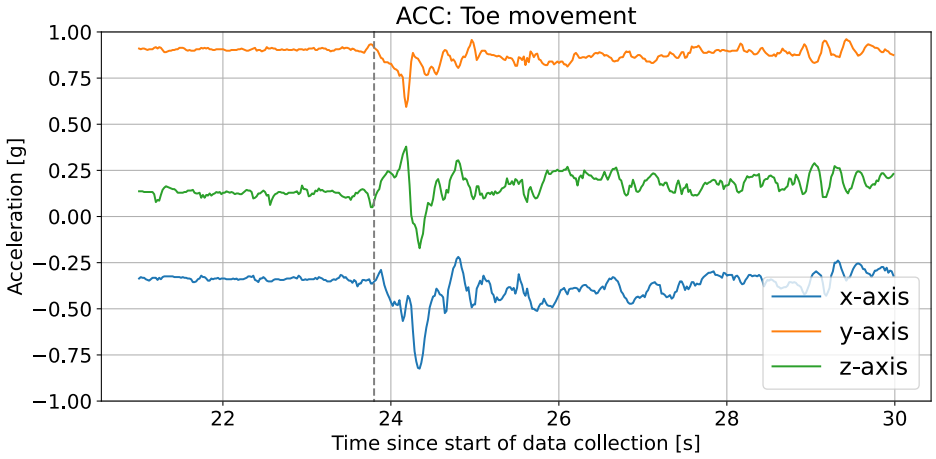

Figure E.1: Data from the accelerometer while moving the toes

Figure E.2 shows the PPG data when moving the toes.

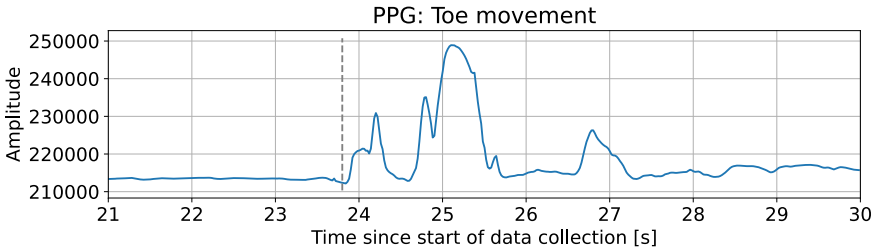

Figure E.2: Data from the PPG sensor while moving the toes

Figure E.3 shows the EDA data when moving the toes.

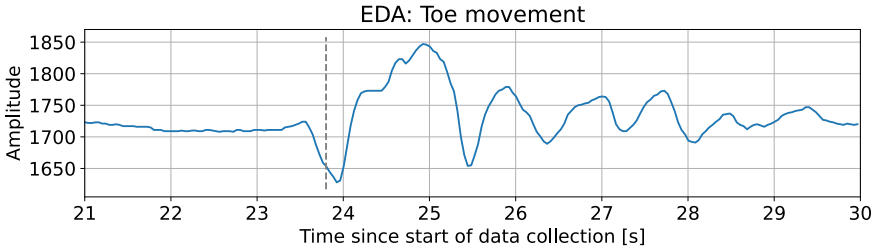

Figure E.3: Data from the EDA sensor while moving the toes

Figure E.4 shows a close-up of the peak discussed in Section 7.2.

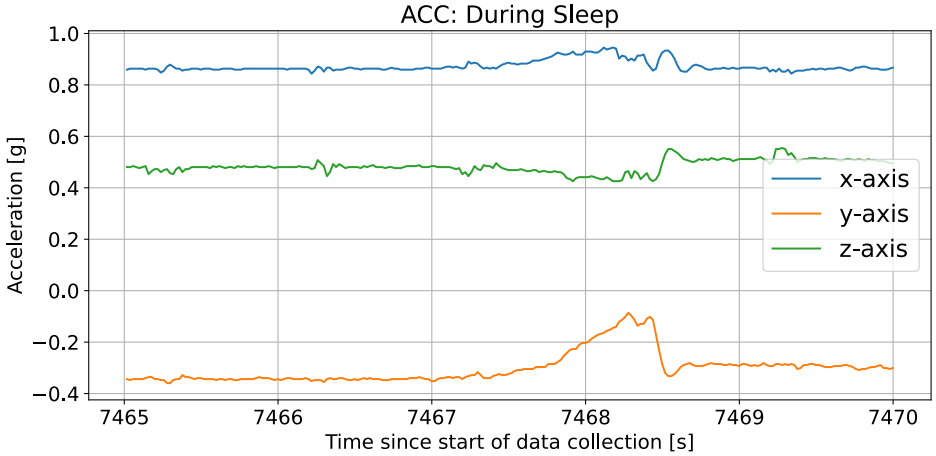

Figure E.4: Close-up of peak

F

Source Code

F.1. server_UDP.ino

Code for data collection using WiFi.

```
#include <WebServer.h>
#include <WiFi.h>
#include <WiFiUdp.h>
#include "LIS3DHTR.h"
#include "MAX30102.h"
#include "EDA.h"

//Connection status
#define DISCONNECTED 0
#define ADVERTISING 1
#define CONNECTED 2

//Wifi login
const char* ssid = "1234";
const char* password = "5678";

//I2c slave addresses
uint8_t acc_addr = 0b0011001;
uint8_t ppg_addr = 0x57;

//Specify pin connections
int eda_pin = 14;
int sda1 = 21;
int sda2 = 8;
int scl1 = 12;
int scl2 = 9;
int vibrator 37;

//Specifi sensor buffer sizes
const int ppg_buffer_size = 32;
const int acc_buffer_size = 32;
const int eda_buffer_size = 16;

//new data request timings
int ppg1_ppull = 40;
int ppg2_ppull = 40;
int acc_ppull = 40;
int eda_ppull = 40;

//state initializations
int wifi_state = DISCONNECTED;
bool ppg1_full = false;
bool ppg2_full = false;
bool acc_full = false;
bool eda_full = false;

//creating sensor instances
```

```
48 EDA eda(eda_pin, eda_buffer_size, eda_ppull, true);
49 MAX30102 ppg1(ppg_addr, Wire, sda1, scl1, ppg_buffer_size, ppg1_ppull, true);
50 MAX30102 ppg2(ppg_addr, Wire1, sda2, scl2, ppg_buffer_size, ppg2_ppull, false);
51 LIS3DHTR acc(acc_addr, Wire, sda1, scl1, acc_buffer_size, acc_ppull, true);
52
53 //creating Wifi instance
54 WiFiUDP Udp;
55 unsigned int localUdpPort = 4210;
56 char incomingPacket[16];
57 int packetSize;
58 uint8_t* UDPpacket;
59
60 void connectToWifi() {
61   // Wait for connection
62   neopixelWrite(RGB_BUILTIN, 0, 50, 0);
63   int status = WL_IDLE_STATUS;
64   while (WiFi.status() != WL_CONNECTED) {
65     delay(500);
66     Serial.print(".");
67   }
68   Serial.println("Connected to wifi");
69   Udp.begin(localUdpPort);
70   Serial.printf("Now listening at IP %s, UDP port %d\n", WiFi.localIP().toString().c_str(),
       localUdpPort);
71
72   //wait for client connection
73   bool readPacket = false;
74   while (!readPacket) {
75     int packetSize = Udp.parsePacket();
76     if (packetSize) {
77       // receive incoming UDP packets
78       Serial.printf("Received %d bytes from %s, port %d\n", packetSize, Udp.remoteIP().
           toString().c_str(), Udp.remotePort());
79       int len = Udp.read(incomingPacket, 16);
80       if (len > 0) {
81         incomingPacket[len] = 0;
82       }
83       neopixelWrite(RGB_BUILTIN, 0, 0, 50);
84       Serial.printf("UDP packet contents: %s\n", incomingPacket);
85       readPacket = true;
86     }
87   }
88 }
89
90 //addapt to new ip
91 void checkToReconnect() {
92   bool readPacket = false;
93     int packetSize = Udp.parsePacket();
94     if (packetSize) {
95       // receive incoming UDP packets
96       Serial.printf("Received %d bytes from %s, port %d\n", packetSize, Udp.remoteIP().
           toString().c_str(), Udp.remotePort());
97       int len = Udp.read(incomingPacket, 16);
98       if (len > 0) {
99         incomingPacket[len] = 0;
100      }
101    }
102  }
103
104  void setup() {
105    Serial.begin(115200);
106    WiFi.begin(ssid, password);
107    Wire.begin(sda1, scl1);
108    Serial.println("");
109
110    packetSize = 0;
111    packetSize += ppg1.getByteArraySize();
112    packetSize += acc.getByteArraySize();
113    packetSize += eda.getByteArraySize();
114
115    //pair with pc
```

```
116     connectToWifi();
117
118     //programm sensors
119     ppg1.programm(false);
120     ppg2.programm(false);
121     acc.programm(false);
122   }
123
124   //This function gathers all the sensor information and sends it to an specified ip
125   void sendUDPData() {
126     UDPpacket = new uint8_t[packetSize];
127     int offset = 0;
128     memcpy(UDPpacket + offset, ppg1.toBytes(), ppg1.getByteArraySize());
129     offset += ppg1.getByteArraySize();
130     memcpy(UDPpacket + offset, acc.toBytes(), acc.getByteArraySize());
131     offset += acc.getByteArraySize();
132     memcpy(UDPpacket + offset, eda.toBytes(), eda.getByteArraySize());
133
134     uint8_t* ptr = ppg1.toBytes();
135     Serial.println();
136     Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
137     Udp.write(UDPpacket, packetSize);
138     Udp.endPacket();
139     delete[] UDPpacket;
140   }
141
142   void loop() {
143     switch (WiFi.status()) {
144       case WL_CONNECTED:
145         checkToReconnect();
146         ppg1_full = ppg1.bufferFull(false);
147         acc_full = acc.bufferFull(false);
148         eda_full = eda.bufferFull(false);
149         if (acc_full && ppg1_full && eda_full) {
150           sendUDPData();
151           eda.resetNew();
152           ppg1.resetNew();
153           acc.resetNew();
154         }
155         break;
156
157       case !WL_CONNECTED:
158         Serial.println("wifi connection: disconnected, restarting connection protocol");
159         neopixelWrite(RGB_BUILTIN, 50, 0, 0);
160         connectToWifi();
161         break;
162     }
163   }
```

F.2. server_Serial.ino

Code for data collection using the serial communication port.

```
#define SERIAL_TX_BUFFER_SIZE 2000
#include "LIS3DHTR.h"
#include "MAX30102.h"
#include "EDA.h"

//Connection status
#define DISCONNECTED 0
#define ADVERTISING 1
#define CONNECTED 2

//I2c slave addresses
uint8_t acc_addr = 0b0011001;
uint8_t ppg_addr = 0x57;

//Specify pin connections
int eda_pin = 14;
int sda1 = 21;
int sda2 = 8;
int scl1 = 12;
int scl2 = 9;
int vibrator = 37;
bool vibrator_on = 0;

//Specifi sensor buffer sizes
const int ppg_buffer_size = 32;
const int acc_buffer_size = 32;
const int eda_buffer_size = 16;

//new data request timings
int ppg1_ppull = 40;
int ppg2_ppull = 40;
int acc_ppull = 40;
int eda_ppull = 40;

//state initializations
int wifi_state = DISCONNECTED;
bool ppg1_full = false;
bool ppg2_full = false;
bool acc_full = false;
bool eda_full = false;

//creating sensor instances
EDA eda(eda_pin, eda_buffer_size, eda_ppull, true);
MAX30102 ppg1(ppg_addr, Wire, sda1, scl1, ppg_buffer_size, ppg1_ppull, true);
MAX30102 ppg2(ppg_addr, Wire1, sda2, scl2, ppg_buffer_size, ppg2_ppull, false);
LIS3DHTR acc(acc_addr, Wire, sda1, scl1, acc_buffer_size, acc_ppull, true);
int packetSize;
uint8_t* packet;

void setup() {
  Serial.begin(115200);
  packetSize = 0;
  packetSize += ppg1.getByteArraySize();
  packetSize += acc.getByteArraySize();
  packetSize += eda.getByteArraySize();

  //vibrator pinselect
  pinMode(vibrator, OUTPUT);

  //programm sensors
  ppg1.programm(false);
  ppg2.programm(false);
  acc.programm(false);
  neopixelWrite(RGB_BUILTIN, 0, 0, 50);
}

//This function gathers all the sensor information and sends it to an specified ip
```

```
68 void sendData() {
69   packet = new uint8_t[packetSize];
70   int offset = 0;
71   memcpy(packet + offset, ppg1.toBytes(), ppg1.getByteArraySize());
72   offset += ppg1.getByteArraySize();
73   memcpy(packet + offset, acc.toBytes(), acc.getByteArraySize());
74   offset += acc.getByteArraySize();
75   memcpy(packet + offset, eda.toBytes(), eda.getByteArraySize());
76 
77   Serial.write(packet, packetSize);
78   Serial.flush();
79   delete[] packet;
80 }
81 
82 void vibrateToggle() {
83   digitalWrite(vibrator, vibrator_on);
84   vibrator_on = !vibrator_on;
85 }
86 
87 void loop() {
88   ppg1_full = ppg1.bufferFull(false);
89   acc_full = acc.bufferFull(false);
90   eda_full = eda.bufferFull(false);
91 
92   if (acc_full && ppg1_full && eda_full) {
93     sendData();
94     eda.resetNew();
95     ppg1.resetNew();
96     acc.resetNew();
97   }
98 }
```

F.3. MAX30102.h

Header file for the MAX30102 class.

```
#ifndef _MAX30102_H
#define _MAX30102_H

#include <stdint.h>
#include <Wire.h>
#include <HardwareSerial.h>
#include <string.h>
#include "sensormonitor.h"

class MAX30102 : public sensor {
public:
  MAX30102(uint8_t addr, TwoWire& _I2c, int sda, int scl, int buffer_size, int ppulltime,
      bool enable);
  void IN_STATUS1_MASK();
  void IN_STATUS2_MASK();
  void INT_EN1_MASK();
  void INT_EN2_MASK();
  void FIFO_WR_PTR_MASK();
  void OVERFLOW_COUNTER_MASK();
  void FIFO_RD_PTR_MASK();
  void FIFO_DATA_MASK();
  void FIFO_CONFIG_MASK();
  void MODE_CONFIG_MASK();
  void SPO2_CONFIG_MASK();
  void LED1_PA_MASK();
  void LED2_PA_MASK();
  void MULTI_LED_CONTROL_REG1_MASK();
  void MULTI_LED_CONTROL_REG2_MASK();
  int32_t* getRedDataPtr();
  int32_t* getIrDataPtr();
  void writeRegister(uint8_t reg, uint8_t data);
  uint8_t readRegister(uint8_t reg);
  void getData();
  void initiateRegisters();
  uint8_t* toBytes();
  void storeData();
  void printData();
  void begin();

private:
  int32_t* red_data;
  int32_t* ir_data;
  int32_t* red_data_stored;
  int32_t* ir_data_stored;
  uint8_t addr;
  uint8_t fifo_rd;
  uint8_t fifo_wr;
  TwoWire* _I2c;
  int sda;
  int scl;
};
#endif
```

F.4. MAX30102.cpp

Member functions of class MAX30102.

```
#include "MAX30102.h"
#include "sensormonitor.h"

//constructor of MAX30102
MAX30102::MAX30102(uint8_t addr, TwoWire& _I2c, int sda, int scl, int buffer_size, int
    ppulltime, bool enable)
  : sensor(buffer_size, ppulltime, enable),
    addr(addr),
    sda(sda),
    scl(scl),
    fifo_rd(0),
    fifo_wr(0),
    _I2c(&_I2c) {

  dataByteArray_size = buffer_size * 8;
  red_data = new int32_t[buffer_size];
  ir_data = new int32_t[buffer_size];
  red_data_stored = new int32_t[buffer_size];
  ir_data_stored = new int32_t[buffer_size];
  dataByteArray = new uint8_t[dataByteArray_size];
};

//Initiates I2c object and specifies pins
void MAX30102::begin() {
  _I2c->begin(sda, scl);
}

//prints the sensor data to the serial port
void MAX30102::printData() {
  for (int i = 0; i < buffer_size; i++) {
    Serial.println(red_data_stored[i]);
    //Serial.print(" ");
    //Serial.println(ir_data_stored[i]);
  }
}

//executes sensor settings
void MAX30102::initiateRegisters() {
  num_register = 15;
  register_addr = new uint8_t[num_register];
  register_data = new uint8_t[num_register];
  register_couter = 0;
  IN_STATUS1_MASK();
  IN_STATUS2_MASK();
  INT_EN1_MASK();
  INT_EN2_MASK();
  FIFO_WR_PTR_MASK();
  OVERFLOW_COUNTER_MASK();
  FIFO_RD_PTR_MASK();
  FIFO_DATA_MASK();
  FIFO_CONFIG_MASK();
  MODE_CONFIG_MASK();
  SPO2_CONFIG_MASK();
  LED1_PA_MASK();
  LED2_PA_MASK();
  MULTI_LED_CONTROL_REG1_MASK();
  MULTI_LED_CONTROL_REG2_MASK();
}

//returns ptr to red_ data
int32_t* MAX30102::getRedDataPtr() {
  return red_data_stored;
}

//returns ptr to ir_ data
int32_t* MAX30102::getIrDataPtr() {
  return ir_data_stored;
```

```
}

//function for writing a single byte to a specifier address
void MAX30102::writeRegister(uint8_t reg, uint8_t data) {
  _I2c->beginTransmission(addr);
  _I2c->write(reg);
  _I2c->write(data);
  _I2c->endTransmission(true);
}

//function for reading a single byte to a specifier address
uint8_t MAX30102::readRegister(uint8_t reg) {
  uint8_t data;
  _I2c->beginTransmission(addr);
  _I2c->write(reg);
  _I2c->endTransmission(false);

  _I2c->requestFrom((int)addr, 1);
  while (_I2c->available()) {
    data = _I2c->read();
  }
  _I2c->endTransmission(true);
  return data;
}

//function for wextracting data from the MAX30102's FIFO
void MAX30102::getData() {
  int get_num;
  //retreive fifo write ponter
  fifo_wr = MAX30102::readRegister(0x04);

  //Specify address to read from
  _I2c->beginTransmission(addr);
  _I2c->write(0x07);
  _I2c->endTransmission(false);

  //calculate the number of new samples to retreive
  if (fifo_wr > fifo_rd) {
    get_num = fifo_wr - fifo_rd;
  } else {
    get_num = buffer_size - fifo_rd + fifo_wr;
  }

  //dont overflow the buffer
  if (buffer_level + get_num > buffer_size - 1) {
    get_num = buffer_size - buffer_level;
  }
  fifo_rd = (get_num + fifo_rd) % buffer_size;

  //request the number of samples
  for (int j = 0; j < get_num; j++) {
    uint8_t temp[6];
    _I2c->requestFrom((int)addr, 6);
    int i = 0;
    while (_I2c->available() && i < 6) {
      temp[i++] = _I2c->read();
    }
    red_data[buffer_level] = ((temp[0] << 16) + (temp[1] << 8) + temp[2]) & 0x3ffff;
    ir_data[buffer_level++] = ((temp[3] << 16) + (temp[4] << 8) + temp[5]) & 0x3ffff;
    _I2c->endTransmission(false);
  }
  _I2c->endTransmission(true);
}

//copys data array to a byte array
uint8_t* MAX30102::toBytes() {
  int index = 0;
  memcpy(dataByteArray + index, red_data_stored, buffer_size * 4);
  index += buffer_size * 4;
  memcpy(dataByteArray + index, ir_data_stored, buffer_size * 4);
  return dataByteArray;
```

```
}

//saves a copy of the data buffer
void MAX30102::storeData() {
  memcpy(red_data_stored, red_data, buffer_size * 4);
  memcpy(ir_data_stored, ir_data, buffer_size * 4);
}

//The following function specify the resgister settings to apply when programming
void MAX30102::IN_STATUS1_MASK() {
  uint8_t mask = 0;
  register_addr[register_couter] = 0x00;
  register_data[register_couter++] = mask;
}

void MAX30102::IN_STATUS2_MASK() {
  uint8_t mask = 0;
  register_addr[register_couter] = 0x01;
  register_data[register_couter++] = mask;
}

void MAX30102::INT_EN1_MASK() {
  uint8_t mask = 0;
  register_addr[register_couter] = 0x02;
  register_data[register_couter++] = mask;
}

void MAX30102::INT_EN2_MASK() {
  uint8_t mask = 0;
  register_addr[register_couter] = 0x03;
  register_data[register_couter++] = mask;
}

void MAX30102::FIFO_WR_PTR_MASK() {
  uint8_t mask = 0;
  register_addr[register_couter] = 0x04;
  register_data[register_couter++] = mask;
}

void MAX30102::OVERFLOW_COUNTER_MASK() {
  uint8_t mask = 0;
  register_addr[register_couter] = 0x05;
  register_data[register_couter++] = mask;
}

void MAX30102::FIFO_RD_PTR_MASK() {
  uint8_t mask = 0;
  register_addr[register_couter] = 0x06;
  register_data[register_couter++] = mask;
}

void MAX30102::FIFO_DATA_MASK() {
  uint8_t mask = 0;
  register_addr[register_couter] = 0x07;
  register_data[register_couter++] = mask;
}

void MAX30102::FIFO_CONFIG_MASK() {
  int samples_averaged = 8;
  int rollover_en = 1;

  uint8_t FIFO_ROLLOVER_EN = (uint8_t)rollover_en << 4;
  uint8_t SMP_AVE;
  switch (samples_averaged) {
    case 1:
      SMP_AVE = 0b000 << 5;
      break;
    case 2:
      SMP_AVE = 0b001 << 5;
      break;
    case 4:
```

```
209       SMP_AVE = 0b010 << 5;
210       break;
211     case 8:
212       SMP_AVE = 0b011 << 5;
213       break;
214     case 16:
215       SMP_AVE = 0b100 << 5;
216       break;
217     case 32:
218       SMP_AVE = 0b111 << 5;
219       break;
220   }
221   uint8_t mask = 0;
222   register_addr[register_couter] = 0x08;
223   register_data[register_couter++] = mask | SMP_AVE | FIFO_ROLLOVER_EN;
224 }
225
226 void MAX30102::MODE_CONFIG_MASK() {
227   int mode = 1;
228   if (!enable) {
229     mode = 3;
230   }
231
232   uint8_t MODE;
233   switch (mode) {
234     case 0:  //hr
235       MODE = 0b010 << 0;
236       break;
237     case 1:  //spo2
238       MODE = 0b011 << 0;
239       break;
240     case 2:  //multi
241       MODE = 0b111 << 0;
242       break;
243     case 3:  //disable
244       MODE = 0b000 << 0;
245       break;
246   }
247
248   uint8_t mask = 0;
249   register_addr[register_couter] = 0x09;
250   register_data[register_couter++] = mask | MODE;
251 }
252
253 void MAX30102::SPO2_CONFIG_MASK() {
254   int scale = 8192;
255   int fsample = 400;
256   int adc_resolution = 18;
257
258   uint8_t SPO2_ADC_RGE;
259   uint8_t SPO2_SR;
260   uint8_t LED_PW;
261   switch (scale) {
262     case 2048:
263       SPO2_ADC_RGE = 0b00 << 5;
264       break;
265     case 4096:
266       SPO2_ADC_RGE = 0b01 << 5;
267       break;
268     case 8192:
269       SPO2_ADC_RGE = 0b10 << 5;
270       break;
271     case 16384:
272       SPO2_ADC_RGE = 0b11 << 5;
273       break;
274   }
275
276   switch (fsample) {
277     case 50:
278       SPO2_SR = 0b000 << 2;
279       break;
```

```
280     case 100:
281       SPO2_SR = 0b001 << 2;
282       break;
283     case 200:
284       SPO2_SR = 0b010 << 2;
285       break;
286     case 400:
287       SPO2_SR = 0b011 << 2;
288       break;
289     case 800:
290       SPO2_SR = 0b100 << 2;
291       break;
292     case 1000:
293       SPO2_SR = 0b101 << 2;
294       break;
295     case 1600:
296       SPO2_SR = 0b110 << 2;
297       break;
298     case 3200:
299       SPO2_SR = 0b111 << 2;
300       break;
301   }
302
303   switch (adc_resolution) {
304     case 15:
305       LED_PW = 0b00 << 0;
306       break;
307     case 16:
308       LED_PW = 0b01 << 0;
309       break;
310     case 17:
311       LED_PW = 0b10 << 0;
312       break;
313     case 18:
314       LED_PW = 0b11 << 0;
315       break;
316   }
317   uint8_t mask = 0;
318   register_addr[register_couter] = 0x0A;
319   register_data[register_couter++] = mask | SPO2_ADC_RGE | SPO2_SR | LED_PW;
320 }
321
322 void MAX30102::LED1_PA_MASK() {
323   float led_current = 18;
324   uint8_t LED_PA = led_current / 0.2;
325   uint8_t mask = 0;
326   register_addr[register_couter] = 0x0C;
327   register_data[register_couter++] = mask | LED_PA;
328 }
329
330 void MAX30102::LED2_PA_MASK() {
331   float led_current = 18;
332   uint8_t LED_PA = led_current / 0.2;
333   uint8_t mask = 0;
334   register_addr[register_couter] = 0x0D;
335   register_data[register_couter++] = mask | LED_PA;
336 }
337
338 void MAX30102::MULTI_LED_CONTROL_REG1_MASK() {
339   uint8_t mask = 0;
340   register_addr[register_couter] = 0x11;
341   register_data[register_couter++] = mask;
342 }
343
344 void MAX30102::MULTI_LED_CONTROL_REG2_MASK() {
345   uint8_t mask = 0;
346   register_addr[register_couter] = 0x12;
347   register_data[register_couter++] = mask;
348 }
```

F.5. LIS3DHTR.h

Header file for the LIS3DHTR class.

```
#ifndef _LIS3DHTR_H
#define _LIS3DHTR_H

#include <stdint.h>
#include <Wire.h>
#include <HardwareSerial.h>
#include <string.h>
#include "sensormonitor.h"

class LIS3DHTR : public sensor {
public:
  LIS3DHTR(uint8_t addr, TwoWire& _I2c, int sda, int scl, int buffer_size, int ppulltime,
      bool enable);
  void TEMP_CFG_REG_MAKS();
  void CTRL_REG1_MASK();
  void CTRL_REG2_MASK();
  void CTRL_REG3_MASK();
  void CTRL_REG4_MASK();
  void CTRL_REG5_MASK();
  void CTRL_REG6_MASK();
  void REVERENCE_MASK();
  void FIFO_CTRL_REG_MASK();
  void INT1_CFG_MASK();
  void NT1_THS_MASK();
  void INT1_DURATION_MASK();
  void CLICK_CFG_MASK();
  void CLICK_THS_MASK();
  void TIME_LIMIT_MASK();
  void TIME_LATENCY_MASK();
  void TIME_WINDOW_MASK();

  int16_t* getxDataPtr();
  int16_t* getyDataPtr();
  int16_t* getzDataPtr();
  void writeRegister(uint8_t reg, uint8_t data);
  uint8_t readRegister(uint8_t reg);
  void getData();
  uint8_t* toBytes();
  void storeData();
  void initiateRegisters();
  void printData();
  void begin();
private:
  int16_t* x_data;
  int16_t* y_data;
  int16_t* z_data;
  int16_t* x_data_stored;
  int16_t* y_data_stored;
  int16_t* z_data_stored;
  bool x_en, y_en, z_en;
  uint8_t addr;
  TwoWire* _I2c;
  int sda;
  int scl;
};
#endif
```

F.6. LIS3DHTR.cpp

Member functions of class LIS3DHTR.

```
#include "LIS3DHTR.h"
#include "sensormonitor.h"

//constructor of LIS3DHTR
LIS3DHTR::LIS3DHTR(uint8_t addr, TwoWire& _I2c, int sda, int scl, int buffer_size, int
    ppulltime, bool enable)
  : sensor(buffer_size, ppulltime, enable),
    sda(sda),
    scl(scl),
    addr(addr),
    _I2c(&_I2c) {

  dataByteArray_size = buffer_size * 6;
  x_data = new int16_t[buffer_size];
  y_data = new int16_t[buffer_size];
  z_data = new int16_t[buffer_size];
  x_data_stored = new int16_t[buffer_size];
  y_data_stored = new int16_t[buffer_size];
  z_data_stored = new int16_t[buffer_size];
  dataByteArray = new uint8_t[dataByteArray_size];
}

//Initiates I2c object and specifies pins
void LIS3DHTR::begin() {
  _I2c->begin(sda, scl);
}

//prints the sensor data to the serial port
void LIS3DHTR::printData() {
  for (int i = 0; i < buffer_size; i++) {
    Serial.print(x_data_stored[i]);
    Serial.print(" ");
    Serial.print(y_data_stored[i]);
    Serial.print(" ");
    Serial.println(z_data_stored[i]);
  }
}

//executes sensor settings
void LIS3DHTR::initiateRegisters() {
  num_register = 17;
  register_addr = new uint8_t[num_register];
  register_data = new uint8_t[num_register];
  register_couter = 0;
  TEMP_CFG_REG_MAKS();
  CTRL_REG1_MASK();
  CTRL_REG2_MASK();
  CTRL_REG3_MASK();
  CTRL_REG4_MASK();
  CTRL_REG5_MASK();
  CTRL_REG6_MASK();
  REVERENCE_MASK();
  FIFO_CTRL_REG_MASK();
  INT1_CFG_MASK();
  NT1_THS_MASK();
  INT1_DURATION_MASK();
  CLICK_CFG_MASK();
  CLICK_THS_MASK();
  TIME_LIMIT_MASK();
  TIME_LATENCY_MASK();
  TIME_WINDOW_MASK();
}

//returns ptr to x_ data
int16_t* LIS3DHTR::getxDataPtr() {
  return x_data_stored;
}
```

```
67
68 //returns ptr to y_ data
69 int16_t* LIS3DHTR::getyDataPtr() {
70   return y_data_stored;
71 }
72
73 //returns ptr to z_ data
74 int16_t* LIS3DHTR::getzDataPtr() {
75   return z_data_stored;
76 }
77
78 //function for writing a single byte to a specifier address
79 void LIS3DHTR::writeRegister(uint8_t reg, uint8_t data) {
80   _I2c->beginTransmission(addr);
81   _I2c->write(reg);
82   _I2c->write(data);
83   _I2c->endTransmission(true);
84 }
85
86 //function for reading a single byte to a specifier address
87 uint8_t LIS3DHTR::readRegister(uint8_t reg) {
88   uint8_t data;
89   _I2c->beginTransmission(addr);
90   _I2c->write(reg | 0x80);
91   _I2c->endTransmission(false);
92
93   _I2c->requestFrom((int)addr, 1);
94   while (_I2c->available()) {
95     data = _I2c->read();
96   }
97   _I2c->endTransmission(true);
98   return data;
99 }
100
101 //function for wextracting data from the LIS3DHTR's FIFO
102 void LIS3DHTR::getData() {
103   int get_num;
104   fifo_level = this->readRegister(0x2f) & 0xf;
105   _I2c->beginTransmission(addr);
106   _I2c->write(0x28 | 0x80);
107   _I2c->endTransmission(false);
108
109   //calculate the number of new samples to retreive
110   if (fifo_level > buffer_size - buffer_level) {
111     get_num = buffer_size - buffer_level;
112   } else {
113     get_num = fifo_level;
114   }
115
116   //request the number of samples
117   if (x_en && y_en && z_en) {
118     for (int j = 0; j < get_num; j++) {
119       uint8_t temp[6];
120       _I2c->requestFrom((int)addr, 6);
121       int i = 0;
122       while (_I2c->available() && i < 6) {
123         temp[i++] = _I2c->read();
124       }
125
126       x_data[buffer_level] = (int16_t)((temp[1] << 8) | temp[0]);
127       y_data[buffer_level] = (int16_t)((temp[3] << 8) | temp[2]);
128       z_data[buffer_level++] = (int16_t)((temp[5] << 8) | temp[4]);
129       _I2c->endTransmission(false);
130     }
131   }
132   _I2c->endTransmission(true);
133 }
134
135 //copys data array to a byte array
136 uint8_t* LIS3DHTR::toBytes() {
137   int index = 0;
```

```
138   memcpy(dataByteArray + index, x_data_stored, buffer_size * 2);
139   index += buffer_size * 2;
140   memcpy(dataByteArray + index, y_data_stored, buffer_size * 2);
141   index += buffer_size * 2;
142   memcpy(dataByteArray + index, z_data_stored, buffer_size * 2);
143   return dataByteArray;
144 }
145
146 //saves a copy of the data buffer
147 void LIS3DHTR::storeData() {
148   memcpy(x_data_stored, x_data, buffer_size * 2);
149   memcpy(y_data_stored, y_data, buffer_size * 2);
150   memcpy(z_data_stored, z_data, buffer_size * 2);
151 }
152
153 //The following function specify the resgister settings to apply when programming
154 void LIS3DHTR::TEMP_CFG_REG_MAKS() {
155   int adc_en = 0;
156   int temp_en = 0;
157
158   uint8_t ADC_PD = adc_en << 7;
159   uint8_t TEMP_EN = temp_en << 6;
160   uint8_t mask = 0;
161
162   register_addr[register_couter] = 0x1f;
163   register_data[register_couter++] = mask | ADC_PD | TEMP_EN;
164 }
165
166 void LIS3DHTR::CTRL_REG1_MASK() {
167   int fsample = 50;
168   int low_power_mode = 0;
169
170   x_en = 1;
171   y_en = 1;
172   z_en = 1;
173
174   if (!enable) {
175     x_en = 0;
176     y_en = 0;
177     z_en = 0;
178   }
179
180   uint8_t Xen = x_en << 0;
181   uint8_t Yen = y_en << 1;
182   uint8_t Zen = z_en << 2;
183   uint8_t ODR;
184   uint8_t LPen = low_power_mode << 3;
185   switch (fsample) {
186     case 0:
187       ODR = 0b0000 << 4;
188       break;
189     case 1:
190       break;
191     case 10:
192       ODR = 0b0010 << 4;
193       break;
194     case 25:
195       ODR = 0b0011 << 4;
196       break;
197     case 50:
198       ODR = 0b0100 << 4;
199       break;
200     case 100:
201       ODR = 0b0101 << 4;
202       break;
203     case 200:
204       ODR = 0b0110 << 4;
205       break;
206     case 400:
207       ODR = 0b0111 << 4;
208       break;
```

```
209       case 1250:
210         LPen = 0 << 3;
211         ODR = 0b1000 << 4;
212         break;
213       case 1600:
214         LPen = 1 << 3;
215         ODR = 0b1000 << 4;
216         break;
217       case 5000:
218         LPen = 1 << 3;
219         ODR = 0b1001 << 4;
220         break;
221     }
222     uint8_t mask = 0;
223     register_addr[register_couter] = 0x20;
224     register_data[register_couter++] = mask | ODR | LPen | Xen | Yen | Zen;
225   }
226
227   void LIS3DHTR::CTRL_REG2_MASK() {
228     int highpass_filter_mode = 0;
229     int highpass_filter_cutoff = 0;
230     int filter_en = 0;
231     int highpass_en_click = 0;
232     int highpass_en_aoi2 = 0;
233     int highpass_en_aoi1 = 0;
234
235     uint8_t HPM = highpass_filter_mode << 6;
236     uint8_t HPCF = highpass_filter_cutoff << 4;
237     uint8_t FDS = filter_en << 3;
238     uint8_t HPCLICK = highpass_en_click << 2;
239     uint8_t HPIS2 = highpass_en_aoi2 << 1;
240     uint8_t HPIS1 = highpass_en_aoi1 << 0;
241     uint8_t mask = 0;
242     register_addr[register_couter] = 0x21;
243     register_data[register_couter++] = mask | HPM | HPCF | FDS | HPCLICK | HPIS2 | HPIS1;
244   }
245
246   void LIS3DHTR::CTRL_REG3_MASK() {
247     int watermark_interupt_en = 0;
248     uint8_t I1_WTM = watermark_interupt_en << 2;
249     uint8_t mask = 0;
250     register_addr[register_couter] = 0x22;
251     register_data[register_couter++] = mask | I1_WTM;
252   }
253
254   void LIS3DHTR::CTRL_REG4_MASK() {
255     int rollover_en = 0;
256     int ble = 0;
257     int scale = 2;
258     int high_res = 1;
259     int self_test = 0;
260
261     uint8_t BDU = rollover_en << 7;
262     uint8_t BLE = ble << 6;
263     uint8_t FS;
264     uint8_t HR = high_res << 3;
265     uint8_t ST;
266
267     switch (scale) {
268       case 2:
269         FS = 0b00 << 4;
270         break;
271       case 4:
272         FS = 0b01 << 4;
273         break;
274       case 8:
275         FS = 0b10 << 4;
276         break;
277       case 16:
278         FS = 0b11 << 4;
279         break;
```

```
280   }
281
282   switch (self_test) {
283     case 0:
284       ST = 0b00 << 1;
285       break;
286     case 1:
287       ST = 0b01 << 1;
288       break;
289     case 2:
290       ST = 0b10 << 1;
291       break;
292   }
293   uint8_t mask = 0;
294   register_addr[register_couter] = 0x23;
295   register_data[register_couter++] = mask | BDU | BLE | FS | HR | ST;
296 }
297
298 void LIS3DHTR::CTRL_REG5_MASK() {
299   int boot = 0;
300   int fifo_en = 1;
301   int fourd_en = 0;
302
303   uint8_t BOOT = boot << 7;
304   uint8_t FIFO_EN = fifo_en << 6;
305   uint8_t D4D_INT1 = fourd_en << 2;
306   uint8_t mask = 0;
307   register_addr[register_couter] = 0x24;
308   register_data[register_couter++] = mask | BOOT | FIFO_EN | D4D_INT1;
309 }
310
311 void LIS3DHTR::CTRL_REG6_MASK() {
312   uint8_t mask = 0;
313   register_addr[register_couter] = 0x25;
314   register_data[register_couter++] = mask;
315 }
316
317 void LIS3DHTR::REVERENCE_MASK() {
318   uint8_t mask = 0;
319   register_addr[register_couter] = 0x26;
320   register_data[register_couter++] = mask;
321 }
322
323 void LIS3DHTR::FIFO_CTRL_REG_MASK() {
324   int fifo_mode = 2;
325   int trigger_select = 0;
326   int fifo_th = 20;
327
328   uint8_t FM;
329   uint8_t TR = trigger_select << 5;
330   uint8_t FTH = fifo_th & 0x1f;
331   switch (fifo_mode) {
332     case 0:
333       FM = 0b00 << 6;
334       break;
335     case 1:
336       FM = 0b01 << 6;
337       break;
338     case 2:
339       FM = 0b10 << 6;
340       break;
341     case 3:
342       FM = 0b11 << 6;
343       break;
344   }
345   uint8_t mask = 0;
346   register_addr[register_couter] = 0x2e;
347   register_data[register_couter++] = mask | FM | TR | FTH;
348 }
349
350 void LIS3DHTR::INT1_CFG_MASK() {
```

```
351   uint8_t mask = 0;
352   register_addr[register_couter] = 0x30;
353   register_data[register_couter++] = mask;
354 }
355
356 void LIS3DHTR::NT1_THS_MASK() {
357   uint8_t mask = 0;
358   register_addr[register_couter] = 0x32;
359   register_data[register_couter++] = mask;
360 }
361
362 void LIS3DHTR::INT1_DURATION_MASK() {
363   uint8_t mask = 0;
364   register_addr[register_couter] = 0x33;
365   register_data[register_couter++] = mask;
366 }
367
368 void LIS3DHTR::CLICK_CFG_MASK() {
369   uint8_t mask = 0;
370   register_addr[register_couter] = 0x38;
371   register_data[register_couter++] = mask;
372 }
373
374 void LIS3DHTR::CLICK_THS_MASK() {
375   uint8_t mask = 0;
376   register_addr[register_couter] = 0x3A;
377   register_data[register_couter++] = mask;
378 }
379
380 void LIS3DHTR::TIME_LIMIT_MASK() {
381   uint8_t mask = 0;
382   register_addr[register_couter] = 0x3B;
383   register_data[register_couter++] = mask;
384 }
385
386 void LIS3DHTR::TIME_LATENCY_MASK() {
387   uint8_t mask = 0;
388   register_addr[register_couter] = 0x3C;
389   register_data[register_couter++] = mask;
390 }
391
392 void LIS3DHTR::TIME_WINDOW_MASK() {
393   uint8_t mask = 0;
394   register_addr[register_couter] = 0x3D;
395   register_data[register_couter++] = mask;
396 }
```

F.7. EDA.h

Header file for the EDA class.

```
#ifndef _EDA_H
#define _EDA_H

#include <stdint.h>
#include <Arduino.h>
#include <string.h>
#include "sensormonitor.h"

class EDA : public sensor {
public:
  EDA(int analogpin, int buffer_size, int ppull, bool enable);
  uint8_t* toBytes();
  void storeData();
  void getData();
  void writeRegister(uint8_t reg, uint8_t data);
  uint8_t readRegister(uint8_t reg);
  void initiateRegisters();
  void printData();
  uint16_t* getEDADataPtr();
  void begin();
private:
  int analogpin;
  uint16_t* edaData;
  uint16_t* edaData_stored;
};
#endif
```

F.8. EDA.cpp

Member functions of class EDA.

```
#include "EDA.h"

//EDA constructor
EDA::EDA(int analogpin, int buffer_size, int ppulltime, bool enable)
  : sensor(buffer_size, ppulltime, enable),
    analogpin(analogpin) {

  dataByteArray_size = buffer_size * 2;
  edaData = new uint16_t[buffer_size];
  edaData_stored = new uint16_t[buffer_size];
  dataByteArray = new uint8_t[dataByteArray_size];
}

//sample data
void EDA::getData() {
  if (enable) {
    edaData[buffer_level++] = analogRead(analogpin);
  }
}

//returns ptr to data
uint16_t* EDA::getEDADataPtr() {
  return edaData_stored;
}

//print data in serial port
void EDA::printData() {
  for (int i = 0; i < buffer_size; i++) {
    Serial.println(edaData_stored[i]);
  }
}

//copys data erray to a byte array
uint8_t* EDA::toBytes() {
  memcpy(dataByteArray, edaData, buffer_size * 2);
  return dataByteArray;
}

//saves a copy of the data buffer
void EDA::storeData() {
  memcpy(edaData_stored, edaData, buffer_size * 2);
}

//empty
void EDA::writeRegister(uint8_t reg, uint8_t data){

};

//empty
uint8_t EDA::readRegister(uint8_t reg) {
  return 0;
};

//empty
void EDA::initiateRegisters() {
}

//empty
void EDA::begin() {
}
```

F.9. sensormonitor.h

Header file for the sensor monitor base class.

```
#ifndef _SENSOR_H
#define _SENSOR_H

//#include <stdint.h>
#include <Wire.h>
#include <HardwareSerial.h>

class sensor {
protected:
  int addr;
  uint8_t fifo_level;
  int buffer_size;
  int buffer_level;
  int pull_timer_old;
  int pull_timer;
  int ppulltime;
  uint8_t* dataByteArray;
  uint8_t* register_addr;
  uint8_t* register_data;
  int register_couter;
  int num_register;
  bool new_data;
  int dataByteArray_size;
  int fpullCheck;
  bool enable;

public:
  sensor(int buffer_size, int ppulltime, bool enable);
  virtual void writeRegister(uint8_t reg, uint8_t data) = 0;
  virtual uint8_t readRegister(uint8_t reg) = 0;
  virtual void initiateRegisters() = 0;
  virtual uint8_t* toBytes() = 0;
  virtual void storeData() = 0;
  virtual void getData() = 0;
  virtual void begin() = 0;

  void programm(bool check);
  void resetNew();
  bool checkNew();
  bool bufferFull(bool reset);
  bool checkPull();
  uint8_t* getByteArray();
  int getByteArraySize();
};
#endif
```

F.10. sensormonitor.cpp

Member functions of base class sensor monitor.

```
#include "sensormonitor.h"

//constructor of base-class sensor
sensor::sensor(int buffer_size, int ppulltime, bool enable)
  : buffer_size(buffer_size),
    ppulltime(ppulltime),
    enable(enable),
    fifo_level(0),
    buffer_level(0),
    pull_timer_old(0){};

//When called, the settings specified in initiateRegisters() are applied
void sensor::programm(bool check = true) {
  //initialize I2c obj
  this->begin();

  //retrieve sensor settings
  this->initiateRegisters();

  //write settings to sensor
  for (int i = 0; i < num_register; i++) {
    this->writeRegister(register_addr[i], register_data[i]);
  }

  //check if the settings are applied correctly
  if (check) {
    Serial.println("Registers to write...");
    Serial.println("----------------");
    for (int i = 0; i < num_register; i++) {
      Serial.print(register_addr[i], HEX);
      Serial.print(":  ");
      Serial.println(register_data[i], BIN);
    }
    Serial.println("----------------");
    Serial.println("Registers read...: ");
    for (int i = 0; i < num_register; i++) {

      Serial.print(register_addr[i], HEX);
      Serial.print(": ");
      Serial.println(this->readRegister(register_addr[i]), BIN);
    }
    Serial.println("----------------");
  }
  delete[] register_addr;
  delete[] register_data;
}

//returns ptr to the byte array of the sensor
uint8_t* sensor::getByteArray() {
  return dataByteArray;
}

//returns true when the buffer is full and unread
bool sensor::checkNew() {
  return new_data;
}

//mark saved data as read
void sensor::resetNew() {
  new_data = false;
}

//when its time to extract new data; extract new data
bool sensor::bufferFull(bool reset = true) {
  if (this->checkPull()) {
    this->getData();
  }
```

```
68
69    //when the buffer is full save buffer content, mark contend as unread, and emptys buffer
70    if (buffer_level == buffer_size) {
71      new_data = true;
72      buffer_level = 0;
73      this->storeData();
74    }
75    return new_data;
76  }
77
78  //checks it it is time to extract new data
79  bool sensor::checkPull() {
80    pull_timer = millis();
81    fpullCheck = pull_timer - pull_timer_old;
82    if (fpullCheck >= ppulltime && buffer_level < buffer_size) {
83      pull_timer_old = pull_timer;
84      return true;
85    }
86    return false;
87  }
88
89  //returns the size of the byte array
90  int sensor::getByteArraySize() {
91    return dataByteArray_size;
92  }
```

F.11. DataLogger.py

Python code for data collection.

```python
import csv
import os
import serial
from datetime import datetime, date
import time
import serial.tools.list_ports as ports
import struct
import numpy as np
from SockReadSerial import *
from SockReadUDP import *

sensorIP = "192.168.2.12"  # The IP that is printed in the serial monitor from the ESP32
port = 4210

def main():
# use this for reading the sensor data using serial communication
    # sock = SockReadSerial(["ppg1", "acc", "eda"])
    # sock.begin(collect=True)

# use this for reading the sensor data using Wifi
    sock = SockReadUDP(["ppg1", "acc", "eda"])
    sock.begin(UDP_IP, SHARED_UDP_PORT, collect=True)

    try:
        while True:
            if sock.readData():
                #(Here put code for when sock.sensor_data is updated)
                pass
    except KeyboardInterrupt:
        # Close the CSV file and serial connection when the program is terminated
        print("Data collection has stopped at ", datetime.now().strftime("%d/%m/%Y %H:%M:%S")
              ,"")
        time.sleep(5)

if __name__ == "__main__":
    main()
```

F.12. SockReadSerial.py

Python class for data collection using serial communication.

```
import csv
import os
import serial
from datetime import datetime, date
import time
import serial.tools.list_ports as ports
import struct
import numpy as np

class SockReadSerial:
    def __init__(self ,sensor_keys=["ppg1", "acc", "eda"], sensor_sizes=None):
        self.sensor_keys = sensor_keys
        self.filename = None
        self.sensors_used = {}
        self.sensor_data = {}
        self.collect = False
        self.serialPort = None
        self.dataSize = 0
        if sensor_sizes is None:
            self.sensors_used = {"ppg1": 256, "ppg2": 256, "acc": 192, "eda": 32}
        else:
            for i, key in enumerate(sensor_keys):
                self.sensors_used[key] = sensor_sizes[i]

        # calculate receiving byte array size
        for sensor in sensor_keys:
            self.dataSize += self.sensors_used[sensor]

    # Initialize Serial obj to start data collection
    # collect: save sensor data
    def begin(self, port=None, baudrate=115200, collect=False):
        if port is None:
            print('Available COM-ports:')
            com_ports = list(ports.comports())
            for i in com_ports:
                print(i.device)  # returns 'COMx'
            print('What COM-port is the ESP32 connected to? Enter the number and press enter.
                ')
            port = "COM" + input()

        print("Selected COM port: "+port)
        self.serialPort = serial.Serial(port=port, baudrate=115200)
        self.serialPort.reset_input_buffer()
        self.collect = collect

        if collect:
            if not (os.path.isdir("DATA")):
                os.mkdir('DATA')

            # Open a CSV file for writing
            self.filename = "DATA/" + str(datetime.now().strftime("%d-%m-%Y-%H-%M-%S")) + ".
                csv"
            row = ["timestamp"]
            for sensor in self.sensor_keys:
                match sensor:
                    case "ppg1":
                        row.extend([sensor+"r_s"+str(i) for i in range(int(self.sensors_used[
                            sensor]/8))])
                        row.extend([sensor + "ir_s" + str(i) for i in range(int(self.
                            sensors_used[sensor] / 8))])

                    case "ppg2":
                        row.extend([sensor + "r_s" + str(i) for i in range(int(self.
                            sensors_used[sensor] / 8))])
                        row.extend([sensor + "ir_s" + str(i) for i in range(int(self.
                            sensors_used[sensor] / 8))])

```

```
62                      case "acc":
63                          row.extend([sensor + "x_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 6))])
64                          row.extend([sensor + "y_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 6))])
65                          row.extend([sensor + "z_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 6))])
66                      case "eda":
67                          row.extend([sensor + "_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 2))])
68
69          with open(self.filename, 'a', newline='') as file:
70              writer = csv.writer(file)
71              writer.writerow(row)
72          file.close()
73
74      # when new data; read from buffer
75      # return 0: no new data
76      # return 1: new data avalible in self.sensor_data
77      def readData(self):
78          if self.serialPort.in_waiting >= self.dataSize:
79              offset = 0
80              data = self.serialPort.read(self.dataSize)
81              for sensor in self.sensor_keys:
82                  temp = data[offset:offset+self.sensors_used[sensor]]
83
84                  offset += self.sensors_used[sensor]
85                  match sensor:
86                      case "ppg1":
87                          temp = np.frombuffer(bytes(temp), dtype=np.uint32).newbyteorder('<')
88                          red = temp[:int(temp.size/2)]
89                          ir = temp[int(temp.size/2):temp.size]
90                          self.sensor_data[sensor] = {"red": red, "ir": ir}
91
92                      case "ppg2":
93                          temp = np.frombuffer(bytes(temp), dtype=np.uint32).newbyteorder('<')
94                          red = temp[:int(temp.size / 2)]
95                          ir = temp[int(temp.size / 2):temp.size]
96                          self.sensor_data[sensor] = {"red": red, "ir": ir}
97
98                      case "acc":
99                          temp = np.frombuffer(bytes(temp), dtype=np.int16).newbyteorder('<')
100                         x = temp[:int(temp.size / 3)]
101                         y = temp[int(temp.size / 3):int((temp.size*2)/ 3)]
102                         z = temp[int((temp.size * 2) / 3):temp.size]
103                         self.sensor_data[sensor] = {"x": x, "y": y, "z": z}
104
105                     case "eda":
106                         temp = np.frombuffer(bytes(temp), dtype=np.uint16).newbyteorder('<')
107                         self.sensor_data[sensor] = {"sensor": temp}
108
109             if self.collect:
110                 newrow = [time.time()]
111                 for sensor in self.sensor_data:
112                     for type in self.sensor_data[sensor]:
113                         newrow.extend(self.sensor_data[sensor][type])
114
115                 with open(self.filename, 'a', newline='') as file:
116                     writer = csv.writer(file)
117                     writer.writerow(newrow)
118                 file.close()
119             return 1
120         else:
121             return 0
```

F.13. SockReadUDP.py

Python class for data collection using WiFi.

```
import csv
import os
from datetime import datetime, date
import time
import socket
import numpy as np
import errno

class SockReadUDP:
    def __init__(self, sensor_keys=["ppg1", "acc", "eda"], sensor_sizes=None):
        self.sensor_keys = sensor_keys
        self.filename = None
        self.sensorIP = None
        self.port = None
        self.sensors_used = {}
        self.sensor_data = {}
        self.collect = False
        self.sock = None
        self.dataSize = 0
        if sensor_sizes is None:
            self.sensors_used = {"ppg1": 256, "ppg2": 256, "acc": 192, "eda": 32}
        else:
            for i, key in enumerate(sensor_keys):
                self.sensors_used[key] = sensor_sizes[i]

        # calculate receiving byte array size
        for sensor in sensor_keys:
            self.dataSize += self.sensors_used[sensor]

    # Initialize Serial obj to start data collection
    # collect: save sensor data
    def begin(self, sensorIP, port, collect=False):
        self.sensorIP = sensorIP
        self.port = port
        self.collect = collect
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  # Internet  # UDP
        self.sock.connect((self.sensorIP, self.port))

        print("Current sensor ip: " + self.sensorIP)
        print("Type \"start\" to start data collection.")
        while input() != "start":
            print("Type \"start\" to start data collection.")
        self.sock.send('START!'.encode())

        if collect:
            print("Data collection has started at ", datetime.now().strftime("%d/%m/%Y %H:%M
                :%S"))
            print("Press CTRL+C to stop data collection")
            if not (os.path.isdir("DATA")):
                os.mkdir('DATA')

            # Open a CSV file for writing
            self.filename = "DATA/" + str(datetime.now().strftime("%d-%m-%Y-%H-%M-%S")) + ".
                csv"
            row = ["timestamp"]

            for sensor in self.sensor_keys:
                match sensor:
                    case "ppg1":
                        row.extend([sensor+"r_s"+str(i) for i in range(int(self.sensors_used[
                            sensor]/8))])
                        row.extend([sensor + "ir_s" + str(i) for i in range(int(self.
                            sensors_used[sensor] / 8))])

                    case "ppg2":
                        row.extend([sensor + "r_s" + str(i) for i in range(int(self.
                            sensors_used[sensor] / 8))])
```

```
63                          row.extend([sensor + "ir_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 8))])
64
65                      case "acc":
66                          row.extend([sensor + "x_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 6))])
67                          row.extend([sensor + "y_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 6))])
68                          row.extend([sensor + "z_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 6))])
69                      case "eda":
70                          row.extend([sensor + "_s" + str(i) for i in range(int(self.
                                sensors_used[sensor] / 2))])
71
72          with open(self.filename, 'a', newline='') as file:
73              writer = csv.writer(file)
74              writer.writerow(row)
75          file.close()
76
77      # when new data; read from buffer
78      # return 0: no new data
79      # return 1: new data avalible in self.sensor_data
80      def readData(self):
81          offset = 0
82          self.sock.setblocking(False)
83          try:
84              if len(self.sock.recv(2048, socket.MSG_PEEK)) >= self.dataSize:
85                  data, newIP = self.sock.recvfrom(2048)
86                  self.sock.connect((newIP[0], self.port))
87                  self.sock.send(b'ACK')
88
89                  for sensor in self.sensor_keys:
90                      temp = data[offset:offset+self.sensors_used[sensor]]
91                      offset += self.sensors_used[sensor]
92
93                      match sensor:
94                          case "ppg1":
95                              temp = np.frombuffer(bytes(temp), dtype=np.uint32).newbyteorder('
                                    <')
96                              red = temp[:int(temp.size/2)]
97                              ir = temp[int(temp.size/2):temp.size]
98                              self.sensor_data[sensor] = {"red": red, "ir": ir}
99
100                         case "ppg2":
101                             temp = np.frombuffer(bytes(temp), dtype=np.uint32).newbyteorder('
                                    <')
102                             red = temp[:int(temp.size / 2)]
103                             ir = temp[int(temp.size / 2):temp.size]
104                             self.sensor_data[sensor] = {"red": red, "ir": ir}
105
106                         case "acc":
107                             temp = np.frombuffer(bytes(temp), dtype=np.int16).newbyteorder('<
                                    ')
108                             x = temp[:int(temp.size / 3)]
109                             y = temp[int(temp.size / 3):int((temp.size*2)/ 3)]
110                             z = temp[int((temp.size * 2) / 3):temp.size]
111                             self.sensor_data[sensor] = {"x": x, "y": y, "z": z}
112
113                         case "eda":
114                             temp = np.frombuffer(bytes(temp), dtype=np.uint16).newbyteorder('
                                    <')
115                             self.sensor_data[sensor] = {"sensor": temp}
116
117                 if self.collect:
118                     newrow = [time.time()]
119                     for sensor in self.sensor_data:
120                         for type in self.sensor_data[sensor]:
121                             newrow.extend(self.sensor_data[sensor][type])
122
123                     with open(self.filename, 'a', newline='') as file:
124                         writer = csv.writer(file)
```

```
125                         writer.writerow(newrow)
126                     file.close()
127                 return 1
128             else:
129                 return 0
130         except socket.error as e:
131             if isinstance(e.args, tuple):
132                 if e.errno == errno.EWOULDBLOCK or e.errno == errno.EAGAIN:
133                     # If the socket would block, return 0 (no data available)
134                     return 0
135             raise
```