An ECG- and PPG-Based Wearable Atrial Fibrillation Detection Device

Signal Acquisition

Amar Kohabir
Alex Smit

TU Delft

An ECG- and PPG-Based Wearable Atrial Fibrillation Detection Device

Signal Acquisition

by

Amar Kohabir
Alex Smit

to obtain the degree of Bachelor of Science in Electrical Engineering
at Delft University of Technology,
to be defended publicly on Tuesday June 29, 2021 at 10:45 AM.

Student number:	4715519	(Amar Kohabir)
	4532023	(Alex Smit)
Project duration:	April 19, 2021 – June 18, 2021	
Thesis committee:	Ir. B. Abdikivanani,	TU Delft, supervisor
	Dr. ir. R.C. Hendriks,	TU Delft, supervisor
	Dr. ir. F. Fioranelli,	TU Delft
	Prof. dr. ir. A. Neto,	TU Delft

Abstract

When symptoms of atrial fibrillation (AF), a common cardiac arrhythmia, are experienced, a Holter monitor or event recorder is used for official diagnosis. Apart from the fact that these devices are experienced as inconvenient, AF can already manifest damage in a pre-symptomatic phase. This thesis is aimed at developing a method for recording heart activity using a wearable device to permit convenient early detection of AF. For this, heart activity is measured continuously by means of photoplethysmography (PPG). A classification algorithm is used to detect AF episodes in the PPG recording. If the algorithm suspects AF, a limb lead I ECG recording is requested from the user. The ECG recording can be analyzed by a clinician for official diagnosis. The Maxim Integrated Max86150 chip is used for the implementation of PPG and ECG. Acceleration data is gathered by means of the Adafruit MMA8451 accelerometer to allow for detection of motion artefacts. These sensors and the data they retrieve are controlled and processed by the ARM Cortex-M7 microcontroller. From the results, PPG recordings have a higher quality when infrared light is used as compared to when red light is used. However, both types of recordings are of sufficient quality for monitoring the heart rate accurately when in stasis. Although complete functionality of the system could not be verified, the results are promising for future work.

Preface

The past weeks have been an exceptional journey for us. Dedicating part of our curriculum to this project amidst the COVID-19 pandemic has amplified our conscience of the importance of health and well-being. With people suffering atrial fibrillation being particularly vulnerable to the virus, it is ever more important to find a remedy for the cardiac disease to prevent future premature casualties. Though no such cure exists as of yet, development of one requires early detection of atrial fibrillation to allow for fruitful medicinal research. It has been an extraordinary opportunity and experience to contribute to this as engineers.

We would like to express our gratitude to our family and friends, whom have supported us throughout our journey thus far. In addition, we would like to thank our fellow colleagues for their collaboration with us on the project. Moreover, we would like to acknowledge ir. B. Abdikivanani, dr. ir. R.C. Hendriks and prof. dr. ir. W.A. Serdijn for their thoughtful input, guidance and supervision during the project. Finally, we would like to thank, ir. M. van Schie and ir. F. Wesselius from Erasmus University Medical Center, P. Omidi from Praxa Sense, dr. E. Ronner from Reinier de Graaf and L. Leenheer from Dienst Elektronische en Mechanische Ontwikkeling for allocating their time and attention to us to share their knowledge and expertise relevant to the project.

Amar Kohabir
Alex Smit
June 18, 2021

Contents

1 Introduction

Atrial fibrillation (AF) is the most common type of cardiac arrhythmia among the world population. According to the Global Burden of Disease study of 2017 [1], the worldwide prevalence of AF was reported to be 37.57 million cases, which composes approximately 0.51% of the global population, and this number is ever increasing. AF is characterized by irregular beating of the heart, which is caused by disorganized electrical signals that emerge from the atria and propagate through the heart. Long-term AF results in inefficient blood perfusion, which in turn may cause fatigue, coagulation, strokes and even demise. The probability of developing AF increases with age and additionally depends on other genetic and lifestyle-related diseases and conditions such as hypertension [2]. Moreover, several studies suggest that vigorous exercise may increase the probability of triggering AF episodes [3], [4], [5]. The frequency and length of AF episodes generally increase over time, the corollary being that AF is classified as a progressive disease [6]. Although medication exists that can alleviate the consequences of AF, no cure nor treatment is available at the time of writing that can permanently remedy the disease itself. Nonetheless, since this is currently an active field of research, fast and accurate diagnosis of AF has become of increasing interest.

1.1. Problem definition

Over the past decades, several methods for diagnosing AF have been used and are still being used in medical environments. A prominent example of this is 12-lead electrocardiography (ECG) used in hospitals, based on the findings of its founding father Willem Einthoven, MD, PhD [7]. 12-lead ECG yields accurate recordings of heart activity but demands the patient to be situated in the hospital, which can be inconvenient. Even more so, diagnosis of early AF, formally known as paroxysmal AF, requires long-term monitoring since AF episodes are relatively infrequent and short at this stage of the disease.

To resolve the issue of immobility, ambulatory ECG devices such as the Holter monitor and event recorder have been introduced. The Holter monitor records heart activity continuously, whereas the event recorder is only activated when the patient experiences symptoms. Although the use of such a device is more convenient for the patient in terms of mobility, the device must still be carried around, which may be experienced as cumbersome. Additionally, electrodes are used which yield more accurate recordings but cause skin irritation when worn over an extended period of time. A more concerning issue, however, is that the use of these devices is only prescribed after the patient has informed a medical professional about symptoms he or she has experienced that are characteristic of AF. This is problematic since AF can already manifest irreversible damage at a pre-symptomatic stage. Naturally, the public would refrain from purchasing such a device when they do not experience any symptoms due to its inconvenience and cost. Therefore, early detection of AF is not feasible via this method and an alternative must be found.

(a) AliveCor KardiaMobile. Courtesy of [8].

(b) Fitbit Sense. Courtesy of [11].

(c) Apple Watch Series 4. Courtesy of [12].

Figure 1.1: State-of-the-art wearables for AF detection.

In recent years, the focus has shifted towards the development of convenient, non-invasive, mobile devices with the aim to realize early detection of AF. This also introduced the use of artificial intelligence (AI) to facilitate automated detection. Expansion of wearable technology, or wearables as these devices are referred to, has rapidly progressed over the past decade to stimulate proactive health monitoring. The AliveCor KardiaMobile [8], for example, is a clinically-validated handheld device that is capable of making single-lead ECG recordings. The device can be paired with a smartphone, on which the recordings can be stored, analyzed by an AI algorithm and sent to a medical professional for official diagnosis. The Fitbit Sense [9] offers similar functionalities and can also be worn on the wrist. In addition to recording an ECG, the Apple Watch Series 4 [10] utilizes photoplethysmography (PPG) for AF detection. PPG is a light-based technology which can be used to assess heart rate and blood oxygen saturation. The advantage of PPG is that it is easier to implement into a wearable than ECG.

The issue with these devices, however, remains that continuous monitoring of heart activity is not feasible. Making an ECG recording with these devices requires the subject to actively maintain physical contact with the electrodes, which is unrealistic to demand on the long term. Although the Apple Watch utilizes PPG to complement this problem, it only monitors in intervals of 15 minutes, which is too long for early AF detection.

1.2. Project description and subsystem division

The goal of this project is to design a wearable device for early detection of AF. The device should be affordable, non-invasive and convenient to wear. In the proposed solution, heart activity is continuously monitored using PPG technology. Acceleration data is recorded to provide a reference signal for motion of the device. The PPG signal is processed to remove noise and motion artefacts. A machine learning algorithm is used to detect the presence of AF in the PPG signal. In the event that AF is detected, the subject is prompted to record a single-lead ECG recording. The ECG recording is then shared with and reviewed by a medical professional for official diagnosis.

To realize the goal of the project, the complete system is divided into the following subsystems:

- Signal acquisition
- Signal processing
- Classification

A block diagram of the interactions between these subsystems is shown in figure 1.2.

The objectives of the signal acquisition subsystem is to acquire continuous PPG and accelerometer recordings. Whenever an AF episode is detected with the PPG recording an ECG recording must be made. The user is informed of this by means of a visual or haptic notification. The PPG, ECG and accelerometer recordings must all be digitized.

The signal processing subsystem is responsible for filtering and processing of the digital signals. The PPG signal is enhanced by reducing noise and motion artefacts. Furthermore, the beat-to-beat heart rate is extracted from the PPG signal.

The objective of the classification subsystem is to extract features from the filtered PPG signal and the accompanying beat-to-beat heart rate. A machine learning classification algorithm should be designed that uses these features to detect AF episodes in the PPG recordings. Detected AF episodes should be noted and the subject should receive feedback if AF is present. Furthermore, if an ECG recording is made, the recording should be stored and prepared to share with a clinician.

This thesis is focused on the realization of the signal acquisition subsystem.

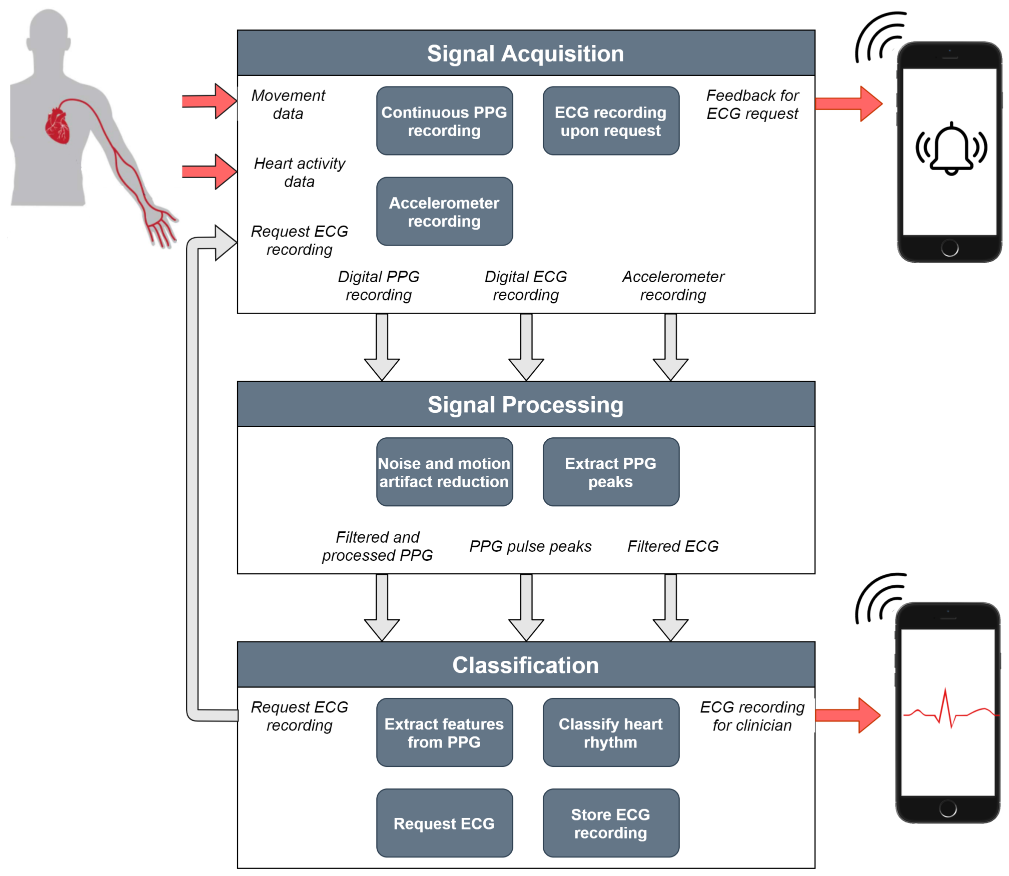


Figure 1.2: Top level overview of the complete system.

1.3. Thesis outline

This thesis has the following structure. In chapter 2, a Program of Requirements (PoR) is established in which the requirements of both the complete system and the subsystem focused on in this thesis are stated. Subsequently, in chapter 3, AF is introduced in more detail and techniques used to detect AF are discussed. In particular ECG and PPG, and important considerations for their implementation in a wearable are thoroughly examined. This forms the foundation for the hardware and software design of the prototype, which is elaborated in chapter 4. The practical implementation and validation of the prototype are then presented in chapter 5, and the results are discussed in 6. Finally, this thesis is concluded with an outlook on future work in chapter 7.

2 Program of Requirements

The program of requirements is split into distinct parts. First, the global requirements of the complete system are given. Next the requirements that are related to the signal acquisition subsystem are presented. These are derived from and therefore categorized according to the global requirements.

2.1. Global requirements

The general requirements of the prototype are split into functional and non-functional requirements. The functional requirements describe what is required to allow the prototype to function properly. The non-functional requirements encompass the requirements that do not contribute to the functioning of the prototype, but are important for its realization.

Functional requirements

1 The device must measure the user's heart activity.

2 The device must be able to detect all atrial fibrillation episodes.

3 The device must generate statements and statistics that are interpretable by a medical professional and are able to aid in the diagnosis of atrial fibrillation.

4 The device must give feedback to the user about the classification outcome.

5 The device must be able to provide core functionality without interaction with other devices. Core functionality consists of recording and storing measurements, AF classification and user feedback.

6 The device should be able to be used during day-to-day activities while maintaining its accuracy.

Non-functional requirements

7 The device must be wearable as an accessory.

8 The device can only provide information and advice; no definitive diagnosis is given.

9 The device should be affordable to be used in a wide audience and therefore components should not cost more than €30,-.

2.2. Signal acquisition requirements

From global requirement **PoR:1**[1] it is evident that the device requires a means of acquiring a heart signal. Chapter 3 discusses the various methods that can be used for this in more detail. As a result of this research, the following requirements have been derived to fulfill requirement **PoR:1**:

1.1 The device must be able to make a digital single-lead ECG recording.

- **1.1.1** The device must be able to detect an AC voltage signal within the range of 5 mV to 10 mV [13].
- **1.1.2** The resolution of the ECG signal must be at least 12 bits [14].
- **1.1.3** The minimum sample rate must be 150 Hz.
- **1.1.4** The minimum common mode rejection ratio must be 89 dB [13].
- **1.1.5** The minimum duration of an ECG recording must be 30 seconds.

1.2 The device must be able to continuously make digital PPG recordings.

- **1.2.1** The photodiode must transduce light into an electrical current, whose AC component has a peak-to-peak value of at least 10 nA [15], [16].
- **1.2.2** The resolution of the PPG signal must be at least 12 bits.
- **1.2.3** The minimum sample rate must be 20 Hz.

1.3 When no tissue is in range of the PPG sensor, the device must idle to reduce power consumption.

Since motion artefacts contaminate these signals as will be explained in chapter 3, measures must be taken to fulfill requirement **PoR:2**. For the signal acquisition subsystem, this leads to the following requirement:

2.1 The device must provide data correlated to movement for the detection of motion artefacts.

Additionally, the signal acquisition department is responsible for integration of the hardware. Therefore global requirements **PoR:4**, **PoR:7** and **PoR:9** are composed of the following sub requirements within the signal acquisition subsystem:

4.1 The device must have a visual indicator that can be used for giving feedback on the classification outcome.

4.2 A vibration motor could be implemented to allow for haptic feedback. This is a trade off requirement concerning time and cost (**PoR:9**).

7.1 Employed technologies must be implementable in the device with a maximum area size of 2000 mm^2 and height of 10 mm.

7.2 The device must use a rechargeable battery and function a minimum of 12 hours on a single charge.

7.3 The device lifetime on a single battery charge should be maximized while maintaining the required performance. This is therefore a trade off requirement regarding measurement of the user's heart activity (**PoR:1**), dimension constraints (**PoR:7.1**), battery capacity (**PoR:7.2**) and cost (**PoR:9**).

9.1 The components of the device should not cost more than €30,-.

[1]References to a specific requirement are denoted by **PoR:***number*. E.g. **PoR:1.1.2** refers to requirement 1.1.2.

3 Background

In order to enhance the design approach, it is insightful to investigate the mechanism of AF and methods used to detect its manifestation. To this end, first a physiological analysis is introduced explaining the functioning of a normal, healthy heart during a cardiac cycle and the anomalies indicative of AF. Subsequently, a survey of different techniques that are used for AF detection is presented.

3.1. Cardiac conduction system

The cardiac conduction system consists of specialized muscle cells that serve as a pathway for the electrical signals propagating through the heart. As shall become clear in this section, AF is characterized by deviations manifested in the conduction of these signals.

3.1.1. Normal sinus rhythm

A heartbeat is constituted by the contraction and relaxation of myocardium (cardiac muscle), formally referred to as systole and diastole respectively. The systolic and diastolic phases are regulated by electrical signals produced within the heart that propagate through the cardiac conduction system. The signals are initiated within the sinoatrial (SA) node, which is located in the right atrium. The initially polarized muscle cells depolarize as the signals travel across the atria, resulting in atrial systole. As a result, blood present in the atria is pumped into the ventricles. The signals propagate towards the atrioventricular (AV) node that is positioned at the top of the interventricular septum. The AV node delays the signals to give the atria sufficient time to transfer the blood to the ventricles. Subsequently, the signals continue from the AV node towards the ventricles, giving rise to depolarization of ventricular myocardium and thus ventricular systole. This allows the blood, that is now present in the ventricles, to be pumped into the arteries. Simultaneously, the atria repolarize which is denoted by atrial diastole. Finally, ventricular systole is followed by ventricular diastole, during which the ventricles relax and repolarize before the SA node re-initiates the cardiac cycle. Figure 3.1 visualizes the trajectory of the electrical signals through the heart in case of NSR.

3.1.2. Atrial fibrillation

In case of AF, the signals in the atria are not conducted properly from the SA node to the AV node. Instead, they propagate in a disorganized fashion, resulting in irregular atrial systole and diastole and thus inefficient transfer of blood to the ventricles. Since some of the signals arrive earlier at the AV node than others, the signals traveling across the ventricles are disordered as well. As a consequence, also the ventricles contract irregularly, ultimately resulting in irregular heartbeats. Hence, the heart rate (HR) and the heart rate variability (HRV) are two characteristics of the heart that distinguishes AF from NSR. The propagation of the signals in case of AF is depicted in figure 3.1.

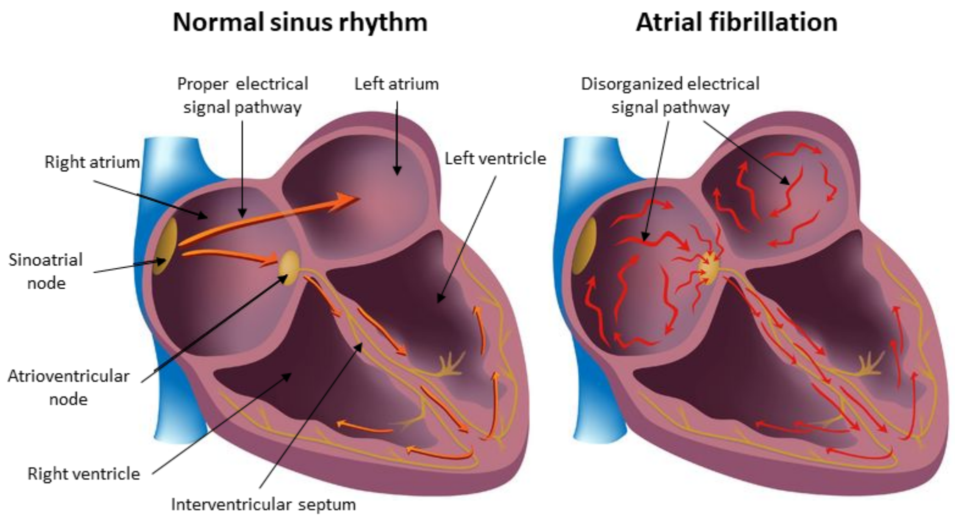


Figure 3.1: The pathway of the electrical signals through the cardiac conduction system during normal sinus rhythm (left) and atrial fibrillation (right). Adapted from [17].

3.2. Techniques for AF detection

Before addressing the various techniques that exist for AF detection, it is important to emphasize the distinction between detection and diagnosis of AF. According to [18], "[a]n electrocardiogram (ECG) recording is necessary to diagnose AF. Any arrhythmia that has the ECG characteristics of AF and lasts sufficiently long for a 12-lead ECG to be recorded, or at least 30 seconds on a rhythm strip, should be considered an AF episode." Thus, currently AF can exclusively be diagnosed by means of ECG. Therefore, ECG is considered the gold standard for AF detection.

ECG is often combined with other techniques to establish a more precise diagnosis. Echocardiography, for example, is used to assess the physical structure of the heart, as dilation of the atria is associated with AF [19]. This technique involves high frequency sound waves that are projected towards the heart. The heart reflects these waves back in a manner that depends on its geometrical structure, which alters throughout a cardiac cycle. From the reflected waves, a two-dimensional image can be rendered that indicates the physical structure of the heart.

In addition to echocardiography, a blood sample is taken and examined on its biochemical contents. Abnormal presence or concentrations of biochemicals that are affiliated with AF are analyzed, including blood cells [20], cardiac troponin [21] and thyroid hormones [22].

Another technology that has recently sparked interest for AF detection is photoplethysmography (PPG). Though PPG has a considerable history, its use was predominantly limited to pulse oximetry and heart rate determination. Nevertheless, various studies with promising results are currently promoting the potential of PPG-based AF detection [23], [24], [25], [26], [27], [28].

The two techniques that are focused on in this study are electrocardiography and photoplethysmography. These techniques offer relatively more freedom with respect to the design of a wearable and can be implemented non-invasively, which is one of the requirements of the device to be developed. Unfortunately, the techniques are plagued by phenomena such as motion artefacts that cannot be neglected and must be resolved. These challenges and other design aspects shall be analyzed critically in the rest of this chapter.

3.3. Electrocardiography

As elaborated in chapter 1, electrocardiography (ECG) is a technique with an extensive history. Being used in contemporary medical environments, ECG serves as the gold standard technology for diagnosis of various cardiovascular diseases, including AF. In this section a brief overview of the principles of ECG is presented with information adapted from [29]. Additionally, aspects relevant for the integration of ECG into a wearable device are discussed.

3.3.1. Underlying principle

As explained in the previous section, muscles contract as a result of depolarization of its cells. Due to a difference in intra- and extracellular ion concentrations, an electrical potential difference exists across a muscle cell's membrane, which is known as the membrane potential. At rest, the membrane potential is negative for muscle cells. However, as the cell depolarizes, this potential becomes positive. Since a muscle consists of numerous cells, this change in potential difference is measurable at the body surface. Two electrodes are used for this measurement, since the potential at one electrode (active) must be measured with respect to other (reference). This yields a voltage signal that changes according to the propagation of the electrical signals in the heart, and thus indicates the electrical activity of the heart.

3.3.2. ECG waveform

The waveform of an ECG recording depends on the placement of the electrodes on the body. This attribute is used in hospital-grade ECG monitors, which employ 10 electrodes for 12-lead recordings. By examining the voltage between various combinations of electrode pairs, the signals propagating through the heart can be viewed from different perspectives, which allows medical professionals to give a three-dimensional interpretation of the heart activity.

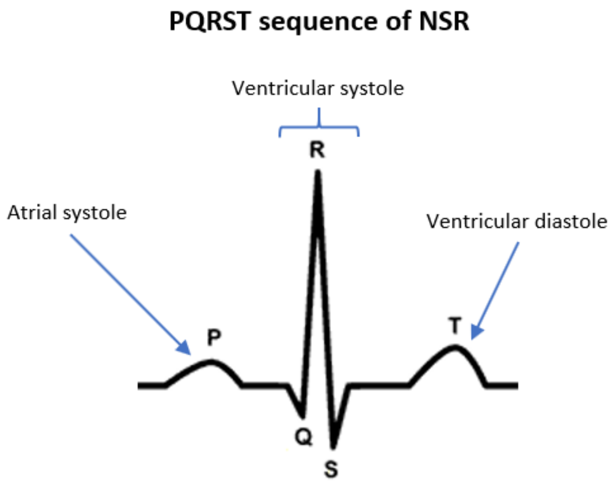


Figure 3.2: PQRST sequence of in case of NSR, showing a clear P wave (atrial systole), QRS comples (ventricular systole) and T wave (ventricular diastole). Atrial diastole is not visible as it is masked by the QRS complex. Adapted from [29].

Figure 3.2 shows the signal of normal heart beat, also known as PQRST sequence, that contains characteristic sections present in an ECG recording. The different sections of the signal each signify a different stage within a heart beat. The first section represents the P wave and indicates depolarization of the atria. At this moment the impulse signal initiated by the SA node travels towards the AV node. The QRS complex denotes depolarization of the ventricles, which occurs as the impulse travels from the AV node through the ventricular muscles. HR and HRV can be determined by examining the repetition rate and time interval between consecutive QRS complexes. Finally, the T wave appears as a result of ventricular repolarization. Interestingly, repolarization of the atria is not visible in an ECG recording. This is because the magnitude of the signal produced by atrial repolarization is relatively small as compared to ventricular depolarization. As a result, atrial repolarization is masked by the QRS complex.

Limb lead I ECG recordings of a subject with NSR and a subject with AF are depicted in figure 3.3. Two differences between the recordings can be recognized that distinguish AF from NSR. One is the absence of the P wave. As mentioned previously, the signals in the heart do not propagate uniformly from the SA node to the AV node. Instead, they travel in an omnidirectional and chaotic fashion. On the recording, this results in no distinct P wave. The disorganized signals persevere within the ventricular muscles after they have reached the AV node. This causes the second visible difference in the recording, namely the irregular repetition of the QRS complex. Due to irregular ventricular systole, HR is not consistent and HRV increases. Hence, these biomarkers aid to distinguish NSR and AF cases.

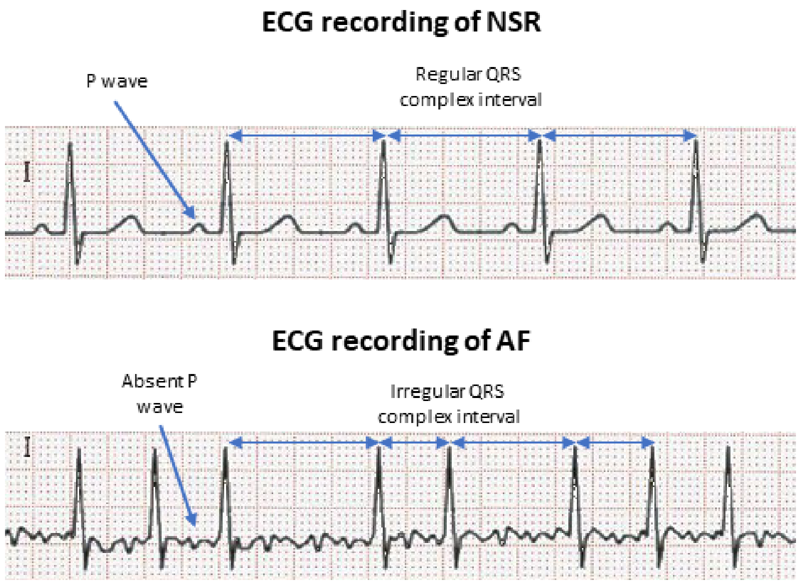


Figure 3.3: Lead I ECG recording during normal sinus rhythm (top) and atrial fibrillation (bottom). The distinct P wave in case of NSR is not visible in case of AF. Also, the interval between consecutive QRS complexes is constant for NSR whereas for AF it is irregular. Adapted from [30].

3.3.3. Single-lead electrodes placement

As explained in the previous section, the waveform of an ECG recording depends on the placement of the electrodes. Attaching 10 electrodes is impractical in non-medical, daily life situations. Fortunately, a single-lead recording suffices for the detection of AF as only the absence of the P-wave and irregular repetition of the QRS complex must be identified [31], [32], [33]. Therefore, a pair of electrodes must be attached such that this information can be accurately extracted from the ECG recording.

Precordial leads require electrodes to be positioned on the chest, which can be inconvenient for the user. Similarly, apart from limb lead I, all limb leads require one electrode to be placed on the left leg, which is also uncomfortable. Fortunately, as shown in the previous section, the P wave and QRS complex can be identified clearly in limb lead I, which allows for clear HR and HRV determination. Although this lead requires one electrode on each arm, this can be realized with minimal effort from the user by placing two electrodes on the device.. One electrode is constantly in contact with, for example, the wrist of the arm on which the device is mounted. The other electrode can then simply be touched with the other hand, which completes the circuit. It is important to note, however, that making long-term continuous ECG recordings is not feasible with this method, as this demands the user to constantly place his or her hand on the device.

Detection of AF in lead I has been proven to be possible [34], [35]. In fact, several commercially available devices utilize lead I ECG for AF detection. For example, the Apple Watch Series 4 is able to record accurate 30-second lead I ECGs, which has been validated using the conventional 12-lead hospital-grade ECG recorder as ground truth [36], [37]. Similar studies have drawn an identical conclusion about the AliveCor KardiaMobile, which has comparable lead I ECG recording capabilities [38], [39].

3.3.4. Bias electrode

In addition to the active and reference electrodes that are used to obtain the ECG traces, 12-lead ECG monitors include a bias electrode. The purpose of the bias electrode is to reduce common-mode noise such as power line interference. This is realized by a Driven Right Leg (DRL) circuit. A simplified front-end amplifier circuit conventionally used in ECG monitors is shown in figure 3.4. The two inputs are the signals from the active and the reference electrode. These are fed into an instrumentation amplifier, which ensures a larger common-mode rejection ratio (CMRR) than the standard differential amplifier. The instrumentation amplifier additionally prevents loading the body since the input impedance is larger than that of a differential amplifier. The gain of the instrumentation amplifier is defined as

$$G_{IA} = \frac{R_4}{R_3}\frac{2R_1 + R_g}{R_g} \tag{3.1}$$

The common-mode signal is then inverted and amplified before it is fed back to the body via the bias electrode in the DRL circuit. The gain of the inverting amplifier is calculated as

$$G_{DRL} = -\frac{R_{f2}}{R_{f1}} \tag{3.2}$$

As a result, the common-mode signal is cancelled out. Resistors R_i and R_o serve as a safety feature and limit the current that can flow into the body in the event that any part of the circuit fails [40].

Given that the CMRR of the instrumentation amplifier is sufficient, omission of the bias electrode is acceptable [41]. Apart from the fact that this enhances the user experience, it is also convenient for the device of this study, since the number of required electrodes reduces from three to two thus reducing the design complexity. Furthermore, power consumption is decreased as an amplification stage can be excluded.

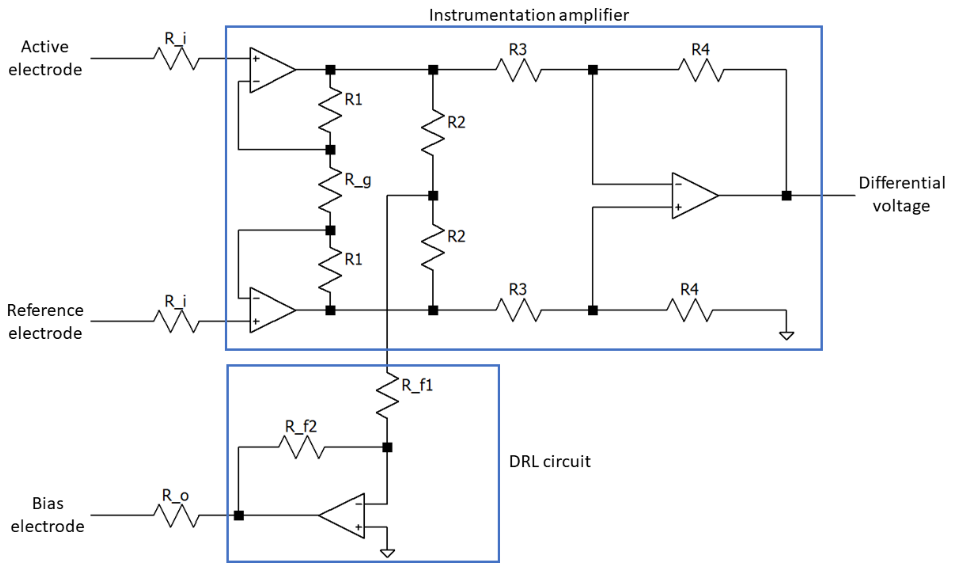
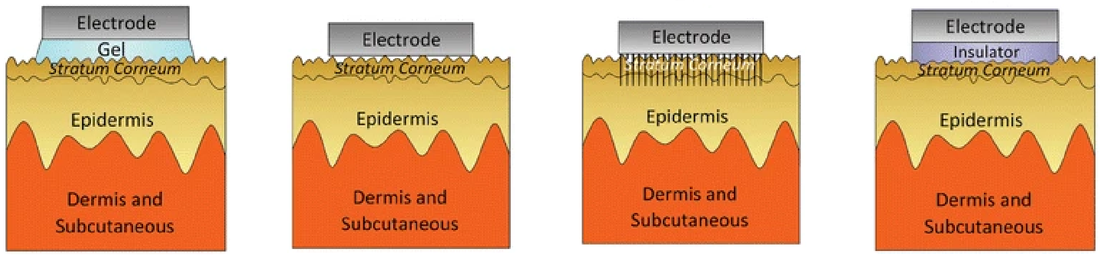


Figure 3.4: A simplified ECG amplifier circuit. The signals from the active and reference electrodes are fed into an instrumentation amplifier. The common-mode signal is inverted, amplified and fed back to the body.



(a) Wet electrode (b) Dry electrode (c) Dry electrode with microneedles (d) Non-contact electrode

Figure 3.5: The different electrode types in the following order: (a) wet electrode, (b) dry electrode, (c) dry electrode with microneedles and (d) capacitive electrode. Adapted from [44].

3.3.5. Electrode types

In addition to the placement of the electrodes, it is important to consider the physical implementation of the electrodes. There exist three types of electrodes: wet electrodes, dry electrodes and non-contact or capacitive electrodes. The electrodes are depicted in figure 3.5. Each electrode has its own advantages and disadvantages, which will be highlighted in this section.

Wet electrodes are electrodes that use a gel as an electrolyte to lower the skin-electrode impedance. The gel is used to improve signal quality. Since these electrodes are adhesive, they are also more resistant to motion artefacts (MAs) caused by physical displacement of the electrodes. Due to these benefits, wet electrodes are currently still the standard in hospitals. A major drawback, however, is that the gel can dry out over time or can be contaminated by sweat through perspiration. This requires periodic replacement of the electrodes. On top of that, the electrodes could cause skin irritation when worn for a long time. Wet electrodes are therefore not suitable for daily usage beyond medical environments.

To resolve the issues of wet electrodes, dry electrodes have been developed, which do not require any gel. Their use is intended for long-term continuous ECG monitoring in wearable devices. However, due to the absence of electrolyte, the skin-electrode impedance is larger and as a result the quality of the signal is diminished. Also, since dry electrodes are nonadhesive, they are more prone to MAs. This is because the skin-electrode impedance is more likely to fluctuate due to physical movement of the user. Remarkably, sweat accumulated over time can serve as an electrolyte and diminish this effect, though it is not as effective as the gel used for wet electrodes. Dry electrodes that make use of microneedles have been developed to overcome these two problems. These needles are injected past the stratum corneum, i.e. the outermost layer of the epidermis, to reduce the skin-electrode impedance. Simultaneously, they fix the electrode and prevent it from moving. Though the needles are painless, wearing them for a long-term may cause skin irritation.

Non-contact electrodes differ from wet and dry electrodes in the sense that they are galvanically isolated from the epidermis. Galvanic isolation prevents the dependence on skin surface properties such as non-uniformity and perspiration. However, the ECG signals sensed by non-contact electrodes are significantly weaker as compared to wet and dry electrodes. Next to that, these electrodes are more prone to MAs.

Recently developed commercially available wearables primarily make use of dry electrode, as its advantages outweigh its disadvantages more than the other electrode types. Examples include the Apple Watch Series 4, FitBit Sense and Withings ScanWatch [42]. Though the quality of the signal obtained by the latter two has not been scientifically validated, the Apple Watch Series 4 has been reported to score accurate results when compared to the standard 12-lead ECG monitor [36], [37], [43].

3.3.6. Removal of motion artefacts

As briefly indicated before, MAs adversely affect the signal quality of an ECG recording. MAs can arise from various sources. An example is variable contact between the skin and the electrode. Even if the electrode is firmly attached to the skin, any movement of the body can still introduce MAs. This is because similarly to the cardiac muscles, other muscles in the body produce their own signals as they contract and relax. Though these signals are of interest for electromyography (EMG), they can contaminate or mask signals relevant for ECG and must therefore be filtered out. In addition to improving electrode properties, alternative methods have been proposed to minimize the presence of MAs. In this section, the focus will mainly be on the auxiliary hardware used for these methods as the implementation of the filtering algorithms is outside the scope of this thesis.

One method examines properties of the ECG signal itself and does not require additional hardware. Techniques such as principal component analysis and independent component analysis are utilized to remove MAs [45], [46]. The disadvantage is that this method relies on multi-lead ECG, which is not feasible with the device of this study. Next to that, these algorithms could be computationally complex, which may result in high power consumption and reduced battery life.

Another approach employs adaptive filtering techniques. This requires a reference signal that is correlated with the MAs present in the ECG lead. The reference signal is realized with auxiliary sensors. An example is the triaxial accelerometer, which detects inertial motion [47], [48], [49]. Skin-electrode impedance has been shown to be suitable for adaptive MA removal as well [50], [51]. Though the hardware design and integration challenges become more complex with the addition of such sensors, the device could be more power efficient.

3.4. Photoplethysmography

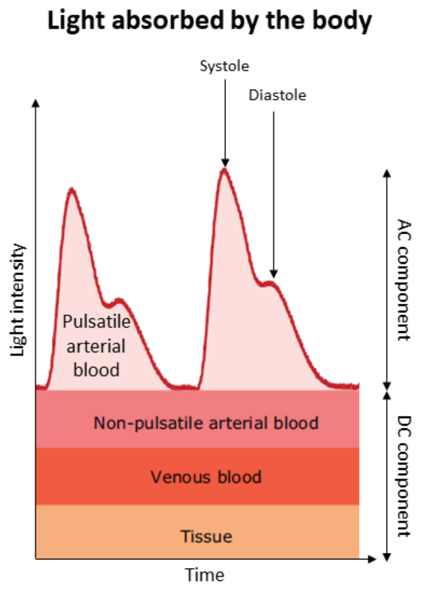
In 1938, Hertzman had found a method to observe the dynamics of blood perfusion with photoelectric-based plethysmography [52], which later became known simply as photoplethysmography (PPG). Despite the fact that the technique itself has an extensive history, consideration of its usage for AF detection has only gained interest in recent years. A brief overview of PPG with information adapted from [53] and aspects relevant for this study is presented in this section.

3.4.1. Underlying principle

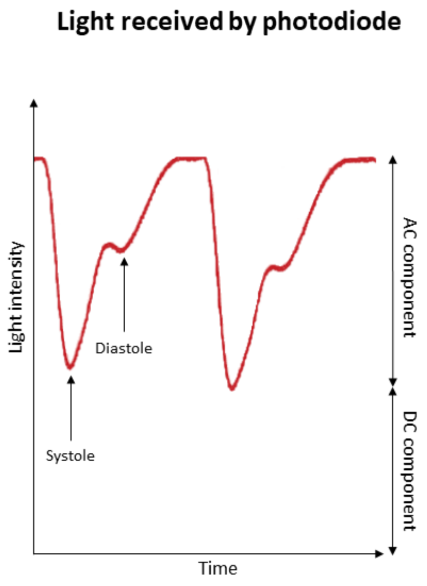
PPG is an optical technique used for monitoring of numerous biomarkers. Prominent biomarkers extracted from a PPG signal include pulse rate (PR), pulse rate variability (PRV) and blood oxygen saturation (SpO_2). As will be elaborated shortly, determination of PR and HR are fundamentally different. A similar statement applies to PRV and HRV. However, research has suggested that PR and PRV can be used as surrogates for HR and HRV respectively, although a discrepancy may arise depending on the ambient temperature and body site on which PPG is performed [54], [55], [56], [57], [58], [59]. Thus, for simplicity, PR and PRV will henceforth be denoted as HR and HRV respectively as well.

As opposed to ECG, PPG is not directly based on the biopotential signals produced by the heart. Instead, an external light[1] emitting source is used, which is conventionally realized with a light emitting diode (LED). Light from this source is directed towards blood vessels. Due to the blood present in the vessels, light will either be transmitted, absorbed or reflected. As the heart circulates the blood in periodic bursts, the amount of blood encapsulated within a certain volume of a vessel will vary accordingly. As a result, the transmission, absorption and reflection of the emitted light will differ over time as well. A photoelectric transducer serving as a receiver is then used to measure either the transmitted or reflected light intensity. Commonly, this is realized with a photodiode.

[1]A more general term would be electromagnetic radiation. However, since specifically the infrared and visible light spectra are used for PPG, for simplicity this will be denoted merely by light.



(a) Light intensity absorbed by the body.



(b) Light intensity received by the photodiode.

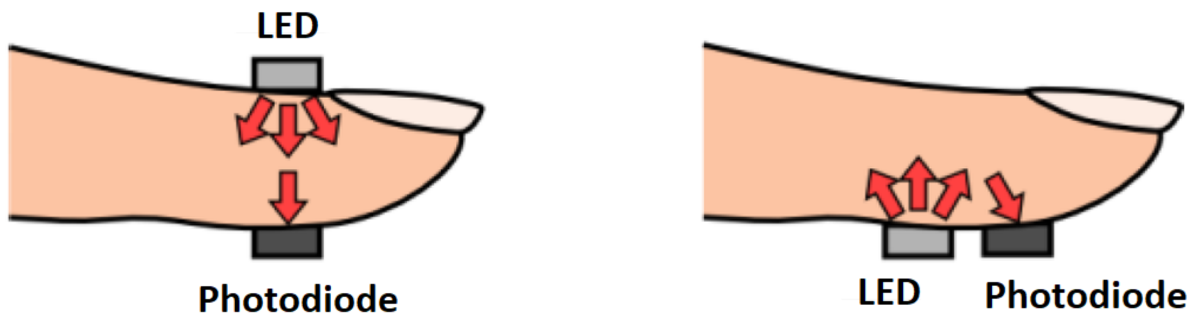
Figure 3.6: The light intensity absorbed by the body and received by the photodiode during a cardiac cycle. The received light signal appears as an inversion of the absorbed light. Adapted from [53].

3.4.2. PPG waveform

The light that is received by the photodiode is composed of a DC component and an AC component. The DC component remains approximately constant and is independent of heart activity. It is determined by light absorption of tissue, venous blood and non-pulsatile arterial blood. The AC component, on the other hand, varies in accordance with heart activity and is characterized by pulsatile arterial blood. Systole and diastole can be identified by two consecutive peaks in the AC component. Figure 3.6a shows how the absorption of light by the body changes during a cardiac cycle. Note that this is not drawn to scale. Theoretically, venous blood introduces an AC component as well, but its magnitude is negligible when compared to that of pulsatile arterial blood. The DC component may still vary as a result of deformation of the tissue, for example through movement. The light received by the photodiode has a waveform that appears as a vertical inversion of the absorbed light waveform and is shown in figure 3.6b.

After the light received by the photodiode is transduced into an electrical signal, it must be amplified and processed to obtain a meaningful PPG signal. The photodiode can be used in two modes: photovoltaic mode and photoconductive mode. In the photovoltaic mode, the incident light intensity is related to the output voltage of the photodiode, whereas in the photoconductive mode the incident light intensity is related to the output current of the photodiode. The voltage-light intensity relation is non-linear, in contrast to the linear current-light intensity relation. Since a linear relation is more convenient to interpret and implement, the photoconductive mode is generally preferred for PPG applications [60].

The current that is output by the photodiode is first passed through a front-end pre-amplifier before any further processing can be performed on the signal. This is done to prevent further signal-to-noise ratio (SNR) degradation. A transimpedance amplifier is used for this as it simultaneously converts the current into a voltage, which is required for the signal processing equipment that follows the front-end amplifier [61]. Since the gain of a transimpedance amplifier is negative, the waveform is inverted yielding a PPG signal that has a shape similar to that of figure 3.6a.



(a) Transmissive PPG. (b) Reflective PPG.

Figure 3.7: Transmissive and reflective PPG. For transmissive PPG, the LED and photodiode are positioned on opposite sides wheres for reflective PPG they are positioned adjacently. Courtesy of [66].

3.4.3. Transmissive and reflective PPG

As briefly mentioned before, there are two modes of PPG: transmissive and reflective PPG. The preferred technique depends on the body site of interest [62], [63]. Transmissive PPG relies on the transmission of light through the body and is therefore generally used for thin body parts such as the fingers, toes or earlobes. An example of a device that uses transmissive PPG is the pulse oximeters used in hospitals. Reflective PPG, on the other hand, is based on the reflection of light and is more suitable for parts of the body that cannot easily be penetrated by light, such as the forehead, chest or wrists. Wearables such as smartwatches generally have a reflective PPG sensor as they are less obtrusive to implement and thus contribute to the user's comfort [64]. Despite the differences in terms of implementation, the waveform of the two different modes is similar as this is largely determined by light absorption of blood [65].

3.4.4. Sensor placement

The quality of a PPG signal highly depends on the body site on which measurements are performed. The most relevant parameter for AF detection that can be extracted from a PPG signal are HR and HRV. Thus, the signal quality must be such that these can be determined accurately. Another criterion is that the PPG sensor should be located such that it is accessible and comfortable for the user. Hence, a body site must be chosen that yields an acceptable trade-off between these criteria.

Various studies have investigated and compared different body sites at resting and moving conditions. Longmore et al. [67] have considered the following eight body sites: forehead, temple, neck, rib cage, wrist, finger, lower back and tibia. The forehead was generally found to yield the most accurate results both when at rest and when moving, followed by the finger. Likely, the finger suffered MAs during walking condition. The same applies to the wrist, which yielded the third most accurate PPG data during walking condition. A remarkable notion from this study, however, is that at rest the wrist yielded significantly worse PPG recordings. This is objected by the vast majority of studies that do find a reasonable accuracy from the wrist under this condition [68], [69], [70]. In fact, smartwatches that are already commercially available have been proven to yield an acceptable PPG signal, which only tends to degrade as the user physically moves [71], [72], [73].

In terms of accessibility and comfort, the majority of the aforementioned body sites are cumbersome. In recent years, alternative body sites that conform to the current comfort standard of wearing conventional jewelry have been considered. For example, the Oura Ring is a commercial ring that can accurately track nocturnal HR and HRV [74]. However, the problem is that this ring is only intended for use during sleep. Additionally, measurements are performed in intervals of 5 minutes. For the purpose of early AF detection this does not suffice, since it requires continuous monitoring. One of the challenges here is that consumer-grade batteries available at this scale can-

Figure 3.8: The Oura Ring and its PPG sensors. Courtesy of [75].

not support long-term continuous measurements.

3.4.5. Light wavelength

Since the human body consists of layers of different tissues each with their distinct absorbent characteristics, it is insightful to investigate what the dependence of a PPG signal is on the wavelength of the light that is used. As light travels through a medium, a fraction of its waves is absorbed by the medium resulting in attenuation of the light. The Beer-Lambert law describes the absorbance of a material and is defined as:

$$A = \log_{10} \frac{I_0}{I(l)} \tag{3.3}$$

where I_0 is the light intensity incident upon the medium and I the light intensity in the medium as a function of the path length l. Alternatively, the absorbance can be expressed in terms of the medium properties as:

$$A = \epsilon(\lambda) M l = \alpha(\lambda) l \tag{3.4}$$

with $\epsilon(\lambda)$ the molar attenuation coefficient as a function of the light's wavelength λ and M the molar concentration. Assuming that the medium is uniform, the product of ϵ and M composes the attenuation coefficient $\alpha(\lambda)$. Using equations 3.3 and 3.4, I can be expressed as:

$$I(l) = I_0 10^{-\alpha(\lambda) l} \tag{3.5}$$

Thus, the transmitted light intensity depends on the characteristics of the medium, the wavelength that is used and the path length.

The different tissues of the human body can be considered different media. Determination of the total reflected light intensity for a particular wavelength is therefore complex, especially since the properties of the tissues change dynamically. Nevertheless, several studies have empirically investigated the optimal wavelength for PPG. Maeda et al. [76], [77] have compared the use of green (525 nm) and infrared (880 nm) light at different temperatures for reflective PPG on a finger. The perfusion index, defined as the ratio between the magnitude of the AC and DC component of a PPG waveform, was the metric used to compare the two signals. At both temperatures, green light was found to yield a larger perfusion index than infrared light. Since the penetration depth of light decreases with smaller wavelengths [78], infrared light travels further within the human body than green light. However, since the information relevant for PPG is only present in blood vessels, reflection of light by any other tissue that is deeper within the body does not carry any information. In fact, since the light is reflected in a randomly scattered fashion, this could be interpreted as a source of noise. Hence, in reflective mode, infrared light PPG is more prone to noise artefacts than green light PPG.

Vizbara et al. [79] have conducted a similar experiment with blue (465 nm), green (520 nm) and infrared (940 nm) light. Reflective PPG was analyzed on the wrist. Also here green light was found to yield better results than infrared light for identical reasons. Interestingly, blue light was found to perform worse than green light. Instead of having a penetration depth that is too large as is the case with infrared light, the opposite is suspected for blue light: its penetration depth is too small. Since blue light cannot penetrate deep enough to reach the blood vessels, the reflected light does not contain sufficient information about HR and HRV. The results of a study performed by Lee et al. [80] agree with this notion. In this study, the use of blue (470 nm), green (530 nm) and red (645 nm) light for reflective PPG on a finger was analyzed. Additionally, the influence of physical movement for the different wavelengths was observed. It was found that infrared light was the most susceptible to MAs, as deformation of non-pulsatile tissue caused the DC component to differ the most over time. Contrarily, blue light was found to be the least impacted by MAs as it was less affected by deformation of deeper tissue. However, the problem remained that the AC component of blue light is limited by the penetration depth. Thus, green light was found to be the optimal wavelength for reflective PPG on peripheral body sites.

3.4.6. Skin pigmentation

An interesting aspect to consider that may influence the signal quality of PPG is skin pigmentation. Skin pigmentation is determined to a significant extent by the concentration of melanin, which is larger in darker skin. Melanin protects the internal cells of the human body by absorbing hazardous UV radiation. However, melanin also absorbs EM radiation in proximity of the UV spectrum, including wavelengths in the visible spectrum [81], [82], [83]. Therefore, dark skin absorbs more of the light used for PPG than light skin, which may result in worse signal quality in the former case.

This concern was further investigated in a study conducted by Fallow et al. [84]. The effect of skin pigmentation was examined in conjunction with different wavelengths for reflective PPG, both at rest and while exercising. Skin pigmentation was assessed based on the Fitzpatrick scale. Blue (470 nm), green (520 nm), red (630 nm) and infrared (880 nm) light were used as light source. In most cases, green light was found to yield the best quality PPG signals. Furthermore, the darkest skin type (type V) was repeatedly found to yield the worst quality. One of the shortcomings of this study, however, is that merely 23 subjects were considered, which were additionally unequally distributed over all skin pigmentation types. The results also have a large standard deviation, which may deem the results to be non-representative.

In recent years, allegations have been made by media accusing consumer electronics companies of manufacturing racially biased smartwatches as they supposedly provide inaccurate HR measurements for darker skin pigmentation [85], [86], [87], [88]. From an ethical perspective, it is of great importance that technology does not discriminate between different ethnicities to endorse racial equity. Thus, in order to verify these allegations, researchers have evaluated the accuracy of HR determination of different smartwatches for different skin types.

One study queried the accuracy of HR determination from PPG signals obtained by the commercially available Apple Watch Series 1 [89]. 45 subjects were uniformly distributed over Fitzpatrick skin types II, III and IV. Their HR was extracted from PPG recordings while at rest and afterwards while exercising. No significant correlation was found between HR accuracy and skin type. Various smartwatches, among which the Apple Watch Series 1 were examined in a similar study [90]. The subjects consisted of 15 individuals with skin type IV or smaller and 7 individuals with skin type V or VI. Interestingly, in this case the Apple Watch was found to be the only smartwatch which showed correlation between skin type and HR accuracy. An equivalent study agrees with this conclusion concerning the Apple Watch, and even infers the same about other smartwatches [91].

Currently, it is still undecided whether skin pigmentation has a significant impact on PPG signal quality. Comparable suggestions have been made for pulse oximetry, which utilizes PPG technology. Although there are both proponents [92], [93], [94] and opponents [95], [96], [97] of such claims, the majority agrees that further research has to be done to draw a definitive conclusion.

Nevertheless, several solutions can be suggested that may solve or circumvent the issue. For example, a greater light intensity may be used, as this may increase the AC component of the PPG signal [79], [98]. Though this also amplifies the noise and DC component as more light can penetrate deeper into the body, these signals should be filtered out regardless as they contain no information about HR nor HRV. A more problematic issue, however, is that using a greater light intensity may have a drastic impact on power consumption. Another solution could be to use reflective PPG on the palm side of a finger, since here the difference in melanin concentration between different skin pigmentations is significantly smaller. However, as mentioned before, this could introduce constraints on other design aspects such as feasibility of implementation and maximum battery capacity.

3.4.7. Removal of motion artefacts

Similarly to ECG, PPG suffers MAs, which degrade signal quality and could even result in futile signals. Multiple methods have been proposed for the removal of MAs in PPG signals that are analogous to those presented for MA removal in ECG signals. As before, the processing of the signals itself is outside the scope of this thesis, hence only the relevant hardware considerations are discussed here.

One approach considers characteristics of the PPG signal itself, such as its periodicity and fundamental frequency within the frequency range of interest [99], [100]. The primary advantage of this method is that no additional hardware is required. However, such algorithms are generally computationally heavy, which may raise power consumption and therefore diminish battery life.

A more conventional method utilizes adaptive filtering techniques and correlates the PPG signal with data obtained by a triaxial accelerometer [101], [102], [103], [104], [105]. Since the accelerometer data contains information about movement, correlating it with the PPG data aids in the detection of where MAs are present and also how severe they are. Furthermore, accelerometer data helps with quantification of the overlap between the frequency components of MAs and the frequency range of interest for HR determination.

Another favored method is similar to the previous, except that the accelerometer is replaced with an additional PPG sensor operating at a wavelength different from the main PPG sensor [106], [107], [108], [109], [110]. A great advantage of this over the preceding method is that micro-MAs can be detected as well [111]. Micro-MAs are induced by movement that cannot be registered by an accelerometer. For example, if a PPG sensor and an accelerometer are positioned on the wrist, movement of a finger will corrupt the PPG signal while the accelerometer does not detect any movement. This is because deformation of the body's tissue due to such movements affect the reflection of the light used for PPG. Hence, using another PPG sensor as a reference for MAs can detect such micro-MAs as well.

4 System Design

The choices made for the design of the signal acquisition system are based on the research presented in the preceding chapter and are elaborated in this chapter. It is important that the device meets the criteria that must be fulfilled. Therefore, first the requirements from chapter 2 are recalled. Subsequently, the design choices regarding the hardware that is used for the prototype are discussed. Afterwards, the software that is required to control the hardware is explained.

4.1. Design choices

The goal of the prototype is to show the system works as intended. Therefore all of the functional requirements should be met. The non-functional requirements, however, are more aimed at a final product. To show that the technological aspects of the system are functional, these do not have to be met explicitly. Nevertheless, they must be met implicitly in the following sense:

- **PoR:7**: The placement of the sensors in the prototype must be identical to the placement of the sensors in the final product. The prototype itself does not have to be wearable: it should merely show that the location of the sensors is viable.
- **PoR:9**: The costs of the prototype may exceed the €30.- range, as equipment needed for development could be more costly. However, the total price of the main components selected may not exceed the €30,- mark.

From the analysis presented in chapter 3 it becomes clear that although there are various methods of detecting atrial fibrillation, the medical standard is ECG. However, to be able to record an ECG two points of contact are required. If these points of contact are too close together they will mainly record the electrical signals produced by the muscle in between. To obtain a clear PQRST complex, the most accessible lead is the limb lead I. The other limb leads require an electrode to be placed on the leg and the precordial leads require electrodes to be placed on the chest. Since these leads are not easily accessible, they are not in line with requirement **PoR:7**. Therefore, in order to be able to make an ECG recording of sufficient quality, the design will consist of one electrode placed on the bottom side (skin side) of the device, and the second electrode will be placed on the top side (air side) of the device. In this manner, an ECG recording can be made by placing a finger from the other hand onto the top side electrode. Dry electrodes have been chosen for this as the advantages of this electrode type outweigh its disadvantages to a greater extent than the other types of electrodes.

This design does not allow for continuous monitoring of the heart activity and therefore does not necessarily comply with requirement **PoR:2** because AF episodes occurring while no ECG recording is being made will not be detected. To meet this requirement a PPG sensor is used, which allows for continuous monitoring of the heart activity. PPG is, however, sensitive to skin pigmentation.

To prevent the device from having a potential racial bias it is desired that the PPG sensor is placed at a location on the palm side of the hand. Here, regardless of ethnicity, the melanocyte (pigment producing skin cells) density is five times lower than elsewhere [112]. This can be done by placing the PPG sensor inside a ring, allowing it to be placed at the palm side of the finger.

To fulfill **PoR:2.1**, it has been decided to utilize an accelerometer and an auxiliary PPG sensor. The accelerometer serves as a reference for the device's acceleration, and can be used to correlate noise in the PPG and ECG recordings to motion. The auxiliary PPG sensor allows for the detection of more subtle movements caused by deformation of body tissue, which cannot be detected by the accelerometer.

4.2. Hardware design

As can be deduced from research presented in the preceding chapter, an increase in interest has been established in the development of wearables over the past decade. As a response, a market has emerged that focuses on the optimization of front-end technology for development and commercial purposes. Although an interesting direction would be to explore the improvement of ECG and PPG technology in wearables, this is beyond the scope of this thesis. Therefore, to avoid reinventing the wheel and possibly working with sub-optimal performance, it has been decided to use existing modules for the sensor implementation. The details of the hardware that will be used for the system is described in this section.

4.2.1. ECG and PPG: Maxim Integrated Max86150

To realize the hardware required for ECG and PPG, the Maxim Integrated Max86150 chip (figure 4.2a) has been chosen. This chip is designed for mobile applications and has both ECG and PPG capabilities encapsulated in a 3.3 mm x 5.6 mm x 1.3 mm form factor. Apart from the fact that its size is small and thus complies with requirement **PoR:7.1**, it also offers benefits such as synchronized ECG and PPG data, simplified storage and communication of data, and a variety of adjustable settings. The cost of the chip is €5.25. Features of the ECG and PPG modules of the chip are highlighted below and the datasheet of the Max86150 chip can be found in [113]. Since certain components are proprietary hardware of Maxim Integrated, the exact details of these cannot be given.

ECG module

The input of the ECG module is provided directly by the electrodes. This signal is passed through a proprietary analog front-end where it is amplified and digitized. The analog front-end rejects interfering signals coming from radio frequency sources, power lines, muscles other than the heart and noise. The signal is fed to an instrumentation amplifier with a voltage gain that can be adjusted between 5 and 50. Its CMRR is 136 dB, which is in accordance with requirement **PoR:1.1.4**. On top of that, a programmable-gain amplifier is used whose voltage gain can be programmed to values between 1 and 8.

After the signal has been processed by the analog front-end it is fed to an 18-bit analog-to-digital (ADC) converter. Since a 12-bit resolution is the minimum requirement, this ADC obeys requirement **PoR:1.1.2**. The sample rate of the ADC can be adjusted within the range of 200 Hz to 3200 Hz, which fulfills requirement **PoR:1.1.3**.

The processed and digitized signal is then stored in the chip's First In First Out (FIFO) buffer. The buffer can be read out externally over an I^2C interface.

PPG module

The PPG module is intended for reflective PPG measurements. It contains two LEDs, one of which emits infrared light (880 nm) and the other red light (660 nm). The LEDs are controlled by LED drivers. The LED current can be adjusted with the drivers and can take on values from 0 mA to 100

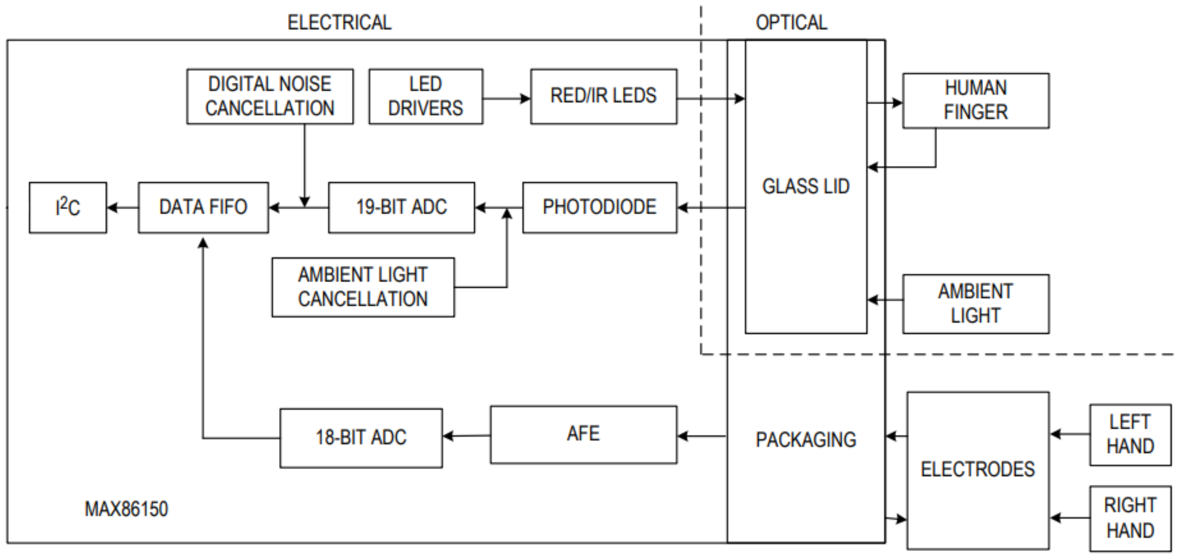


Figure 4.1: Signal flow of the ECG and PPG modules. Courtesy of [113].

mA. Next to that, the pulse width of the LEDs can be programmed from 50 µs to 400 µs. This allows great flexibility in terms of making a trade-off between signal accuracy and power consumption.

The reflected light is registered by a photodiode and transduced into a current signal. A proprietary ambient light cancellation circuit is used to diminish the presence of ambient light captured by the photodiode. This increases the effective dynamic range of ADC.

Subsequently, the current signal is digitized with a 19-bit ADC, which is in line with requirement **PoR:1.2.2**. The sample rate of the ADC can be set to a value between 10 Hz and 3200 Hz, thus fulfilling requirement **PoR:1.2.3**. The PPG module is also capable of sample averaging, which allows reduction of the data throughput. Up to 32 adjacent samples can be averaged into a single sample.

The digital signal is then passed through a proprietary discrete-time filter to reject power line interference and noise, after which the sample is eventually stored in the FIFO buffer.

One of the PPG module's key features is that the LEDs can be turned off if the user is not in proximity of the sensor. This reduces the power consumption in case the sensor is not utilized, which fulfills requirement **PoR:1.3**.

Shortcomings

The chip does have a few shortcomings. Firstly, a custom PCB must be designed on which the chip and other components required for its implementation must be soldered. This is because no adapter exists for this specific chip. Secondly, the chip does not have a green LED, which was concluded to be optimal for reflective PPG in chapter 3. Thirdly, an extensive driver must be written to control the chip and retrieve its recorded data. Nevertheless, since the benefits of the chip outweigh these shortcomings, its usage is a viable trade-off.

4.2.2. Accelerometer: Adafruit MMA8451

To monitor movement of a user and identify motion artefacts, the accelerometer that is selected is the Adafruit MMA8451 (figure 4.2b). This device is chosen because of its configurable scale setting and its built-in signal processing unit. The measurement range can be configured to be 2g, 4g and 8g, where g is the gravitational constant at the Earth's surface, which is ideal for development as it is not known what the optimal range is. The built-in signal processing unit is capable of detecting when the device is in free fall and when a motion exceeds a specified threshold. A high-pass filter with adjustable cutoff frequency is also incorporated to prevent baseline wandering. The on-chip signal processing saves processing power of the micro controller, allowing for more headroom for the rest of the software [114]. The cost of the accelerometer is €6.95.

4.2.3. PCB design

As mentioned in section 4.2.1, a custom PCB must be designed for the implementation of the Max86150 chip. The PCB also contains other components that are important for the functioning of the chip. The components are derived from the chip's datasheet [113] and include:

- three pull-up resistors (1 kΩ each) for the communication ports;
- four decoupling capacitors (one 1 μF, one 10 μF, two 4.7 μF) to reduce AC interference on DC lines;
- one capacitor (1 μF) for the ECG's analog front-end filter and;
- two resistors (50 kΩ) and two capacitors (22 nF) for current limiting and radio frequency filtering of the ECG inputs.

The design of the PCB requires careful consideration of the placement of components and traces to minimize interference and to allow for efficient troubleshooting in case of faults. A schematic of the PCB circuit and its corresponding layout can be found in appendix B. The cost of the PCB is €45.76.

4.2.4. Development board: STMicroelectronics NUCLEO-H743ZI2

A microcontroller is required to control the sensors. For this, the ARM Cortex-M7 is selected. This choice is based on the performance of the Cortex-M7. ARM has various models for microcontrollers. Listed from low performance to high performance these are the Cortex-M0, Cortex-M3, Cortex-M4 and Cortex-M7. The supported instruction set also depends on the model [115]. During the development process, the highest performing microcontroller is desired to minimize limitations imposed on the implementation of software that can be run on the microcontroller. In a later stage the algorithm of the software can be improved, and the final microcontroller can be selected based on the processing demands of the finished software.

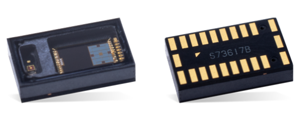
The development board used must have the following features:

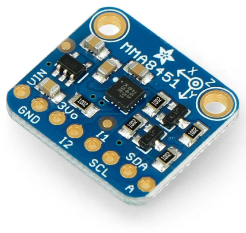
- Support for an I^{2}C bus.
- The logic level voltage must be 1.8 V to be compatible with the Max86150.
- The development board must be able to be connected to the computer with a USB cable.

The following features are desired, but no required:

- The development board can supply 1.8 V to the Max86150 power supply pin.
- The development board can supply 3.3 V with a current of 150 mA to the Max86150 LED power supply pin and the MMA8451.
- An LED that can be used for visual feedback.

Given these requirements, the STMicroelectronics NUCLEO-H743ZI2 development board [116] (figure 4.2c) has been selected, which supports all the required features as well as the desired features. The NUCLEO-H743ZI2 also has 2Mbytes of flash memory, which allows a lot of strings to be stored for debugging statements. There is also 1 Mbyte of RAM available, which is important as the recordings will require a lot of memory. A single recording of 60 seconds, made at a sample rate of 100Hz where each sample takes up 4 bytes would already require 24 Kbytes. The cost of the development board is €25.72.


(a) Maxim Integrated Max86150


(b) Adafruit MMA8451

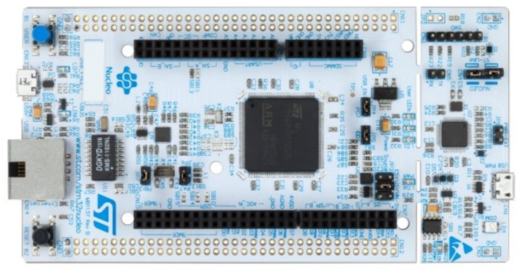

(c) STMicroelectronics NUCLEO-H743ZI2

Figure 4.2: The hardware that is used for the implementation of the prototype.

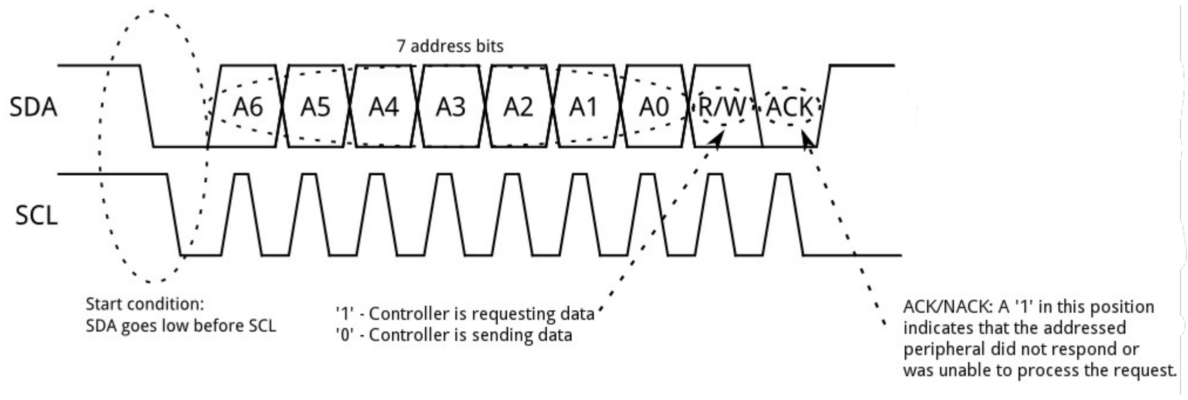

7 address bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
Start condition:
SDA goes low before SCL
'1' - Controller is requesting data
'0' - Controller is sending data
ACK/NACK: A '1' in this position indicates that the addressed peripheral did not respond or was unable to process the request.

Figure 4.3: Master requesting transmission from a slave using an I^{2}C bus.Courtesy of [118].

4.3. Software design

In order to control the sensors and retrieve data from them, software must be developed that facilitates this. In this section, an overview of the software and the drivers that are required for the communication of hardware is given.

4.3.1. I^{2}C protocol

The sensors communicate through an I^{2}C interface. The I^{2}C bus is a synchronized serial communication protocol using two wires. The first wire is the Serial Data wire, usually denoted by SDA, and the second wire is the Serial Clock wire denoted by SCL. Both SDA and SCL are pulled high by a pull-up resistor. A master and up to 127 slaves can be connected on a single I^{2}C bus. The master can initiate a transmission by transmitting a start bit (0), followed by the 7 bit slave address, and finally a read (1) or write (0) bit. The slave then acknowledges the message by sending an acknowledgement bit (0) on the SDA line. After the transaction of the last byte is complete the master transmits a stop bit (0). The master controls the clock signal on the SCL line during the transactions, the maximum frequency for standard I^{2}C implementations is 400 kHz. Figure 4.3 gives a visual representation of the I^{2}C protocol [117].

4.3.2. Mbed OS

The software development must be done using a real time operating system (RTOS), due to the various modules sending interrupts to the microcontroller. These interrupts need to be handled in real time, and can occur while a different block of code is being executed. An RTOS offers support for real time processing requirements in various manners.

The RTOS of choice is Mbed OS [119]. This is an open source operating system, and can be used commercially and personally under an Apache 2.0 license (a license for free of charge software usage). Mbed OS can be used with the ARM Compiler 6, which is the default setting, or the GNU Arm Embedded Compiler (GCC). Both compilers use C++ as programming language. Mbed OS also has an integrated development environment (IDE) called Mbed Studio. Within Mbed Studio, the prototyping board for which the program is written can be selected, which automatically includes all of the correct files at compile time.

Mbed OS comes with various application programming interfaces (API's), a complete overview of which can be found at [119], under the 'API references and tutorials' section. The most notable API's are:

- **I^{2}C**: used for creating an I^{2}C master, and handling transactions.
- **DigitalIn**: used for reading signals on the I/O pins of the microcontroller.
- **DigitalOut**: used for controlling signals on the I/O pins of the microcontroller.
- **InterruptIn**: used for creating interrupt signals on I/O pins of the microcontroller.
- **EventQueue**: used for creating a queue, containing pointers to functions that should be executed.
- **Thread**: used for creating and defining parallel tasks. An EventQueue can also be Attached to a thread, allowing the code added to the queue to be executed in parallel.

4.3.3. System communication

The system must be able to communicate with the Max86150, the MMA8451 and the PC. The Max86150 and the MMA8451 both use an I^{2}C bus for communication. The PC is connected to the NUCLEO-H743ZI2 by USB. The system must be monitored on the PC, which allows for more convenient debugging of code and also for checking whether the system operates as intended. The software is written in C++, and runs on top of Mbed OS. A high-level system overview is given in figure 4.4.

For the Max86150 chip a driver has to be designed that can run on Mbed OS. The MMA8451, however, already has an existing driver for Mbed OS that can be imported into the project. The drivers' task is to handle communication with the external devices. Functions within the driver for changing specific settings are implemented. The driver however, does not have any data control; it simply makes the retrieved data available. For controlling the drivers another class is implemented, entitled the control class. The Max86150 driver and the MMA8451 driver are initialized from within this class. The control class also synchronizes the data from both modules, handles the interrupts received, controls the length of the recording and keeps collecting data until a recording is complete. Once complete, a callback function is called for processing by the signal processing subsystem. An overview of the classes is given in figure 4.5. In the following subsection, the design of the Max86150 driver will be discussed in further detail.

4.3.4. Max86150 driver

In this section, the driver for the Max86150 chip is described. First, the most important data structures of the Max86150 class will be introduced along with the related class methods. The structures and methods that handle information regarding the settings of the Max86150 chip are discussed in subsection **Settings** and the structure and methods that keep track of the various sensors (otherwise known as Data Elements) and their related data are elaborated in subsection **Data elements**. In figure 4.9 an overview of the data-structure of the class is given, which serves as a schematic overview and reference throughout these subsections. Thereafter, the I^{2}C class from Mbed OS will

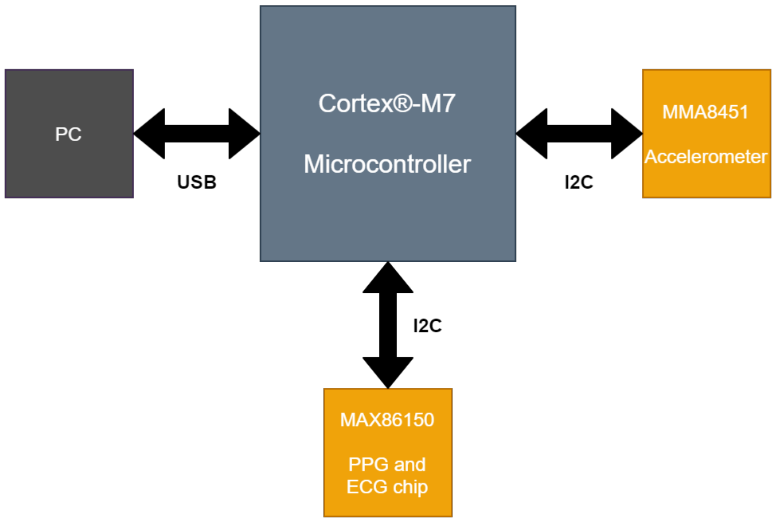


Figure 4.4: Overview of the communication between the different subsystems.

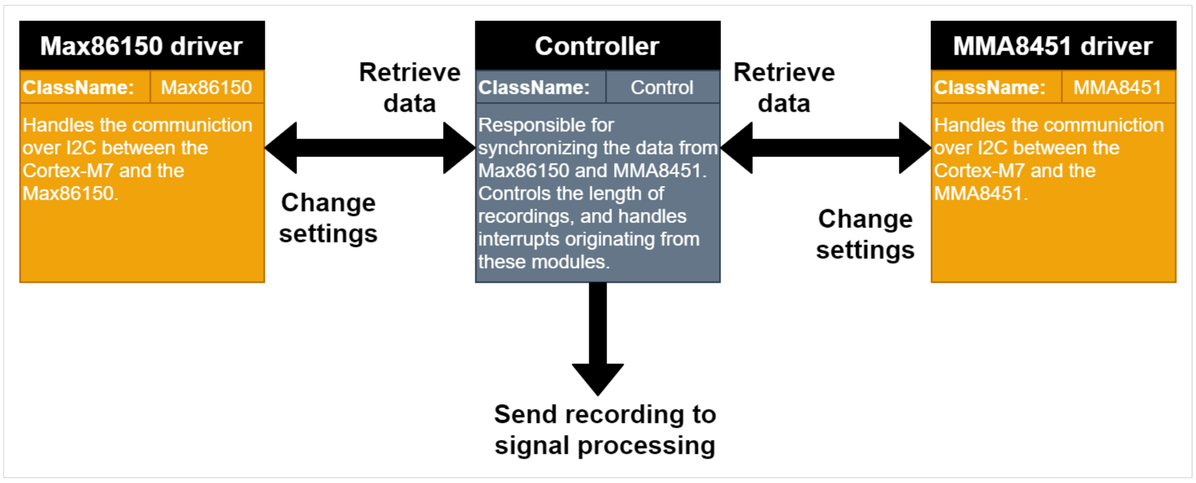

Max86150 driver
ClassName: Max86150
Handles the communiction over I2C between the Cortex-M7 and the Max86150.
Retrieve data
Change settings
Controller
ClassName: Control
Responsible for synchronizing the data from Max86150 and MMA8451. Controls the length of recordings, and handles interrupts originating from these modules.
Retrieve data
Change settings
MMA8451 driver
ClassName: MMA8451
Handles the communiction over I2C between the Cortex-M7 and the MMA8451.
Send recording to signal processing

Figure 4.5: Overview of the designed classes.

be introduced together with the class methods responsible for handling the communication with the Max86150 in subsection **I²C communication**. Finally, the class constructor will be analyzed in subsection **Class constructor**.

Settings

The Max86150 has many configurable settings. A complete overview of the registers is given in Appendix A.1. These register names are linked to the register addresses using an enumeration, namely `enum RegisterAddresses`. Multiple settings can be changed within a single register. To keep track of every setting the structure `struct Setting` is used. This structure contains `RegisterAddresses address`, which holds the register address for the setting, and `uint8_t bitmask`, which contains a bitmask that can be used to select specific bits within the register. The `struct SettingsMap` contains a constant instance of `struct Setting` for each individual setting, and the instance `settings` of this structure is used to hold the information of all the settings. Due to the way constant variables are handled in C++ the values cannot be changed. To change a specific setting, the address can be accessed using `settings.<specific_setting>.address`, and the bitmask can be accessed in the same manner using `setting.<specific_setting>.bitmask`. Using this method avoids bugs arising from typos and avoids having to reference the Max86150 throughout development.

In order for settings to be adjusted easily and reliably, the options from which can be chosen are defined using enumerations. In figure 4.9 these enumerations are shown in red. When calling a member function that changes a setting, the argument must be of the specified enumeration type (i.e. it must be one of the items inside the enumeration). This method has the benefit that, when using a linter, the list of options will appear while filling in the function arguments. Most settings have their own unique member function which can be called to change it. Other settings are grouped together in a single function as it would make no sense to change them individually. The following public member functions can be used to configure the Max86150:

- `setInterrupt(bool enable, Max86150::InterruptEnableConf specific_interrupt)`
- `setFifoReadPointer(uint8_t address)`
- `setFifoAlmostFullInterruptClear(bool value)`
- `setFifoFullBehaviour(bool value)`
- `setFifoAlmostFullBehaviour(bool value)`
- `setFifoAlmostFullValue(uint8_t value)`
- `setFifoEnable(bool enable)`
- `sleep(bool enable)`
- `reset()`
- `setPpgAdcRange(Max86150::PpgAdcRangeConf value)`
- `setPpgSampleRate(Max86150::PpgSampleRateConf value)`
- `setPpgPulseWidth(Max86150::PpgPulseWidthConf value)`
- `setPpgSampleAveraging(Max86150::SampleAveragingConf value)`
- `setProximityThreshold(uint8_t threshold)`

- `setLedPulseAmplitude(uint8_t amplitude, Max86150::LedType led)`
- `setEcgSampleRate(Max86150::EcgAdcSampleRateConf base_sample_rate, Max86150::OverSamplingRatioConf over_sampling_ratio)`
- `setAmplifierGains(Max86150::PgaGainConf pga_gain, Max86150::InstrumentationAmplifierGainConf ia_gain)`
- `getPartId()`

Data elements

The Max86150 allows for enabling and disabling its various sensors dynamically. To enable a sensor (from here on referred to as a data element) the corresponding value must be written to one of the FIFO Data Control Registers. Each FIFO Data Control Register can hold two data elements. The five available data elements are:

- Infra Red PPG
- Red PPG
- Infra Red Led in pilot mode
- Red Led in pilot mode
- ECG

Once a data element is enabled, its samples are stored in the FIFO buffer. The FIFO buffer can store up to 32 samples for each data element, and each sample is three bytes. Once the FIFO buffer reaches the limit defined by the FIFO_A_FULL setting an interrupt is triggered. The FIFO can then be read from the I^{2}C bus by reading the FIFO Data Register. First, the first sample of the data element (FD1) is sent, which consists of three bytes. Then the first sample of the second, third and fourth data elements (FD2, FD3 and FD4) are sent unless one of these is disabled. When the first sample of all enabled data elements is sent the second sample is sent for FD1, FD2, FD3 and FD4, unless one of these is disabled. This goes on until the transmission is terminated by the master. Therefore it is convenient to read the FIFO write pointer and read pointer registers before reading the FIFO buffer. These can be used to determine the number of new samples which the master can then keep track of and end the transmission accordingly. The bytes are all stored in the `char[384] fifo_buffer`. This buffer is capable of storing 32 samples of three bytes for each of the four data elements. The choice to store the read data in a single buffer is from a performance perspective, as this allows all samples to be read in sequence. If each data element would have its own separate buffer, the function to read from an I^{2}C bus would have to be called for each individual sample, and the slave address of the Max86150 together with the register address to read the FIFO would have to be sent with each call. All the information regarding a data element is stored in the structure `struct FifoDataElement`. This structure contains the following properties (also see figure 4.9):

- `char* read_buffer[32]`: This array of pointer contains pointers to `fifo_buffer`. The locations pointed to correspond to the first byte of each one of the 32 samples that can be read from FIFO.
- `FifoSensorsConf sensor`: This contains information regarding the type of data element that populates this FIFO data element.
- `uint8_t new_samples`: This indicates the number of new samples that have been read into `fifo_buffer` from this data element.

Anytime changes are made to a FIFO data element, the `read_buffer` pointers for all the FIFO data elements are updated. The class methods associated with the data elements are:

- `setDataElementPointers(void)`
- `getSensorData(FifoSensorsConf sensor, int32_t *write_buffer)`
- `setFifoDataElement(Max86150::FifoSensorsConf sensor, bool enable)`
- `readFifo()`

I^{2}C communication
One of the main tasks of the Max86150 driver is to handle the communication over the I^{2}C bus. Mbed OS has a class called I^{2}C, which can be used to implement a master on an I^{2}C bus. The basic I^{2}C protocol is already implemented within this class. Upon creating an object of the I^{2}C class, the SDA and SCL pins are defined. The public member functions `write(int address, const char* data, int length, bool repeated)` and `read(int address, char* data, int length, bool repeated)` can be called after initialization, and handle the transaction of `data` between the master and the slave with the corresponding `address` as defined by the I^{2}C protocol [120]. In the Max86150 chip's datasheet [113] the protocols for reading from a register and writing to a register are defined. The slave address for the Max86150 is `0xBC` for write and `0xBD` for read. For writing, the protocol is:

1. Write register address from which data should be read to slave address `0xBC`.
2. Transmit data that should be written to the register.

This is illustrated in figure 4.6. For reading, the protocol is:

1. Write register address from which data should be read to slave address `0xBC`.
2. Send repeated start bit.
3. Request read from slave address `0xBD`.

This is illustrated in figure 4.7.

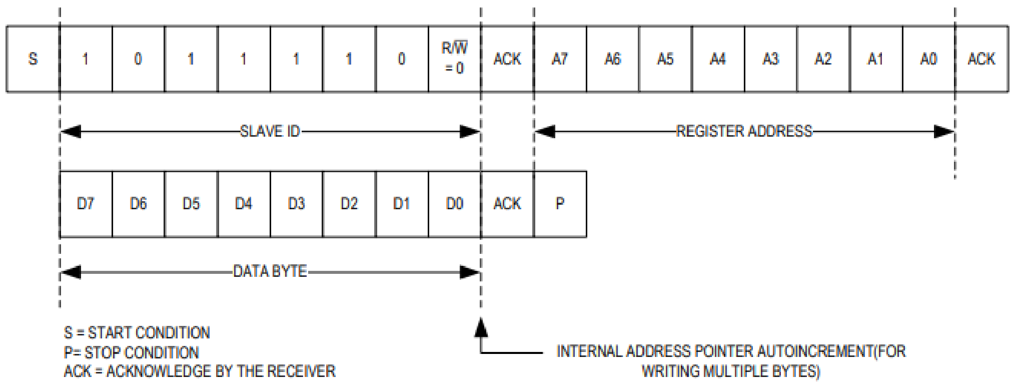


Figure 4.6: Protocol for writing data to a register on the Max86150. Courtesy of [113].

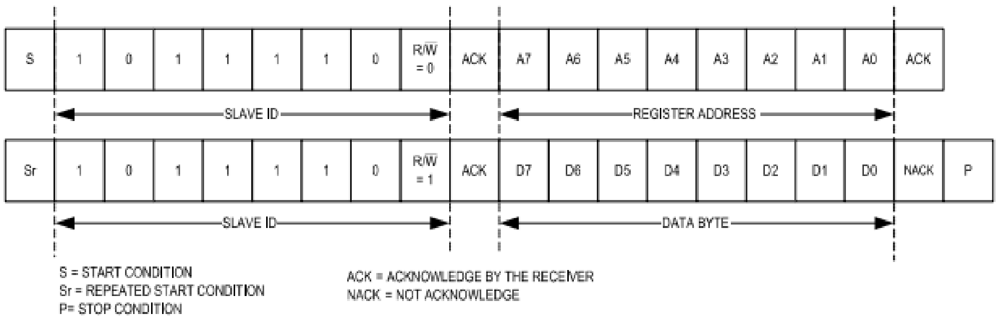


Figure 4.7: Protocol for reading data from a register on the Max86150. [113]

These reading and writing protocols are implemented in the class as private member functions `readRegister(Max86150::RegisterAddresses address, char* read_buffer, uint8_t repeat=1)` and `writeRegister(Max86150::RegisterAddresses address, uint8_t data))` of the class Max86150 (see Appendix A). This is to ensure that the register addresses that are written to or read from are defined within the class as discussed in Settings. This avoids undefined behaviour as a result of writing to incorrect register addresses.

Class constructor

The class constructor is responsible for initializing all of the required objects together with the Max86150. The most important object is the I^{2}C object, which is initialized with its corresponding pins. An interrupt pin is also created, allowing the Max86150 interrupts to be handled properly. As described in the Max86150 datasheet [113] the analog power supply must be turned on first, followed by the digital power supply and finally the LED power supply. If the power-on sequence is successfully executed an interrupt is sent. This is also illustrated in figure 4.8. The power-on sequence is controlled using the I/O pins of the Cortex-M7 in the constructor. Once the interrupt is detected, the register containing the interrupt cause is read. If the interrupt flag corresponds to the PWR_RDY flag, the Max86150 has been successfully initialized. When no interrupt is detected the Max86150 has not been initialized correctly and the program exits. For a final check, the constructor attempts to read the contents of the Part ID register. If the received data does not equal `0x1E` something is wrong with the physical I^{2}C connections and the program sends a notification before exiting.

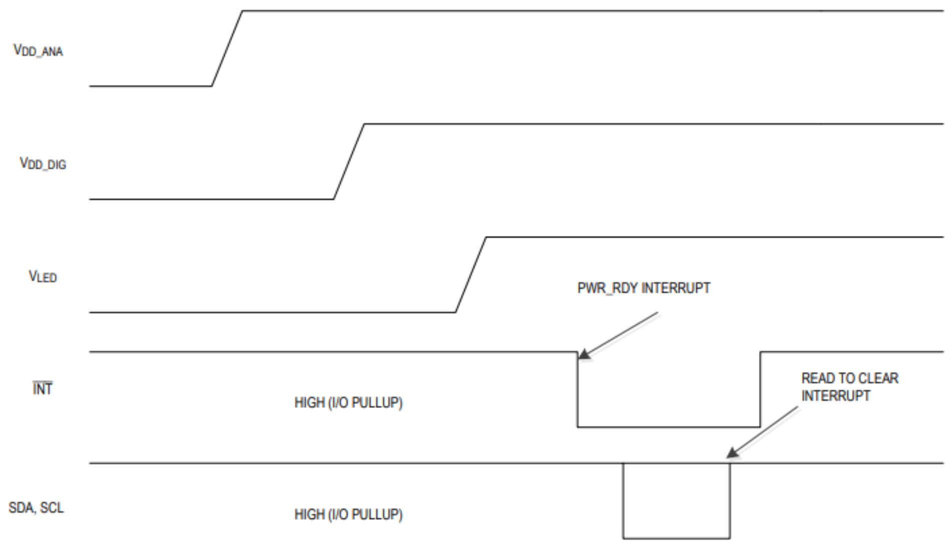


Figure 4.8: Power on sequence from Max86150 datasheet. Courtesy of [113].

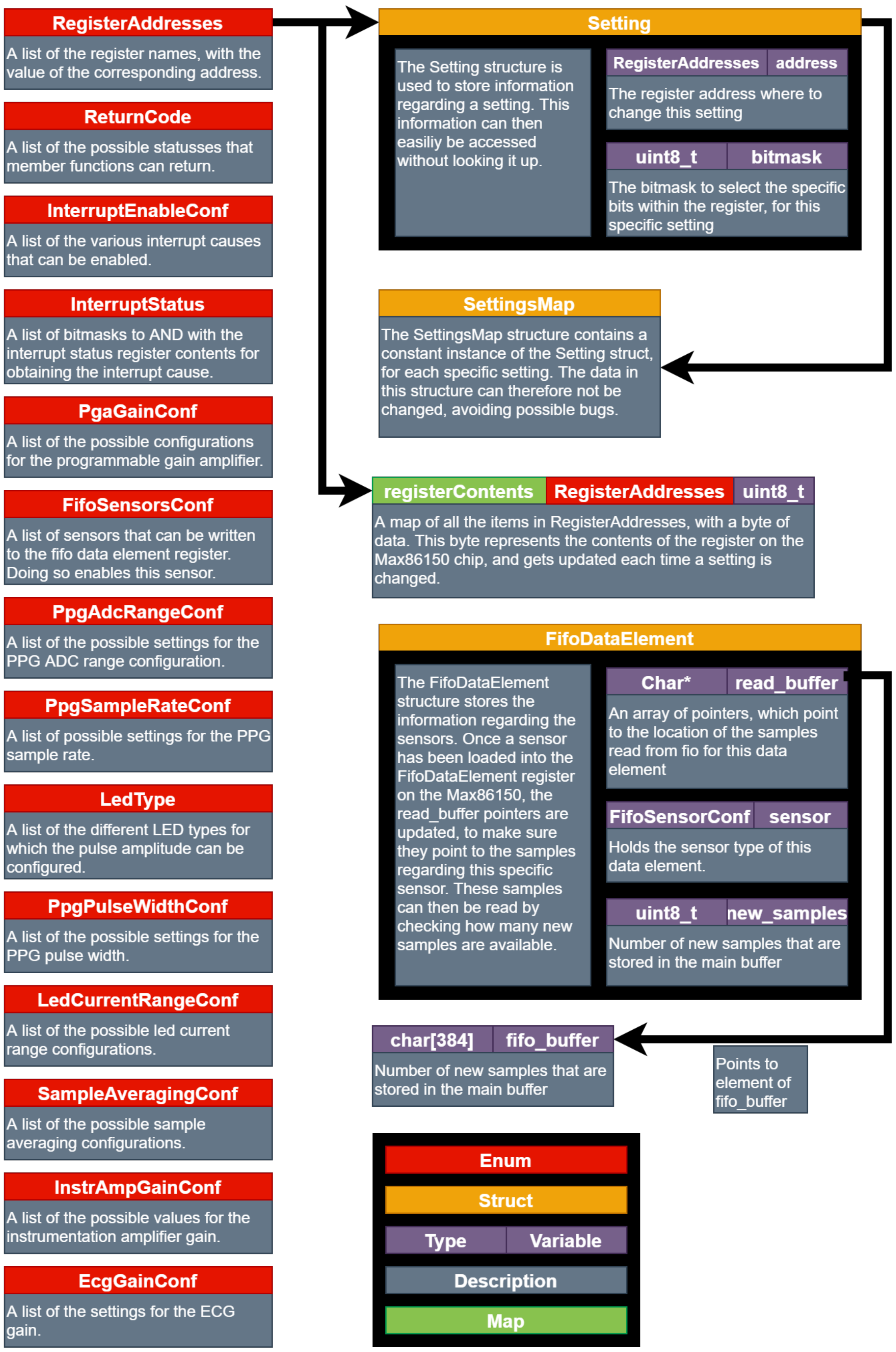

RegisterAddresses
A list of the register names, with the value of the corresponding address.
ReturnCode
A list of the possible statusses that member functions can return.
InterruptEnableConf
A list of the various interrupt causes that can be enabled.
InterruptStatus
A list of bitmasks to AND with the interrupt status register contents for obtaining the interrupt cause.
PgaGainConf
A list of the possible configurations for the programmable gain amplifier.
FifoSensorsConf
A list of sensors that can be written to the fifo data element register. Doing so enables this sensor.
PpgAdcRangeConf
A list of the possible settings for the PPG ADC range configuration.
PpgSampleRateConf
A list of possible settings for the PPG sample rate.
LedType
A list of the different LED types for which the pulse amplitude can be configured.
PpgPulseWidthConf
A list of the possible settings for the PPG pulse width.
LedCurrentRangeConf
A list of the possible led current range configurations.
SampleAveragingConf
A list of the possible sample averaging configurations.
InstrAmpGainConf
A list of the possible values for the instrumentation amplifier gain.
EcgGainConf
A list of the settings for the ECG gain.
Setting
The Setting structure is used to store information regarding a setting. This information can then easiliy be accessed without looking it up.
RegisterAddresses address
The register address where to change this setting
uint8_t bitmask
The bitmask to select the specific bits within the register, for this specific setting
SettingsMap
The SettingsMap structure contains a constant instance of the Setting struct, for each specific setting. The data in this structure can therefore not be changed, avoiding possible bugs.
registerContents RegisterAddresses uint8_t
A map of all the items in RegisterAddresses, with a byte of data. This byte represents the contents of the register on the Max86150 chip, and gets updated each time a setting is changed.
FifoDataElement
The FifoDataElement structure stores the information regarding the sensors. Once a sensor has been loaded into the FifoDataElement register on the Max86150, the read_buffer pointers are updated, to make sure they point to the samples regarding this specific sensor. These samples can then be read by checking how many new samples are available.
Char* read_buffer
An array of pointers, which point to the location of the samples read from fio for this data element
FifoSensorConf sensor
Holds the sensor type of this data element.
uint8_t new_samples
Number of new samples that are stored in the main buffer
char[384] fifo_buffer
Number of new samples that are stored in the main buffer
Points to element of fifo_buffer
Enum
Struct
Type Variable
Description
Map

Figure 4.9: An overview of important class properties. The color coding legend is defined at the bottom right.

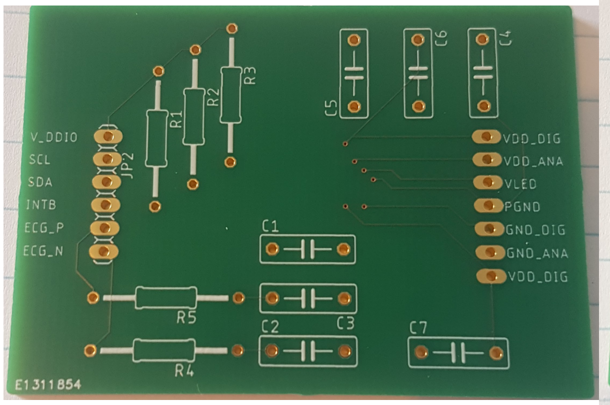
5 System Implementation and Validation

Following the design in chapter 6, the physical implementation and validation of the system is discussed in this chapter. First, the hardware is implemented, which includes the soldering of the Max86150 chip and the other components. The hardware is then validated for any potential faults to rule out physical defects. Given that the hardware functions as intended, the Max86150 driver can be tested to verify that the software operates properly as well.

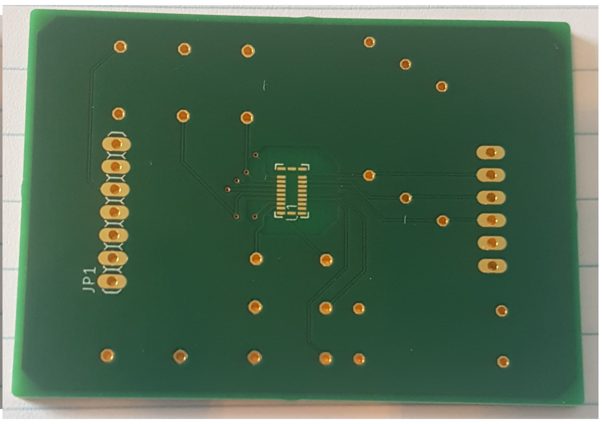
5.1. Hardware implementation and validation

The Max86150 chip must soldered onto the PCB in order to connect it to the development board. First, the connections on the PCB are tested using a multimeter in resistance mode. One of the probes is pressed against the solder pad where the I/O pins will be connected. These are shown on the left and right sides in figure 5.1a . The other probe is pressed against the corresponding pad of the Max86150's landing pattern which is at the center of figure 5.1b. If the two pads are connected then the resistance is below 10 Ω, otherwise a value in the order of megaohms is measured. This check is done to ensure that the PCB was manufactured correctly and all of the required connections are made.

The pads used to solder the Max86150 are located underneath the chip. Therefore, a soldering iron cannot be used as it cannot reach these pads. To solder the chip onto the PCB reflow soldering must be used. With reflow soldering a solder paste is used. This solder paste consists of microscopic balls of tin, suspended in a resin, which functions as flux. The solder paste is usually deposited



(a) The top side of the Max86150 adapter PCB.



(b) The bottom side of the Max86150 adapter PCB.

Figure 5.1: The front and back side of the PCB designed for the Max86150 chip.

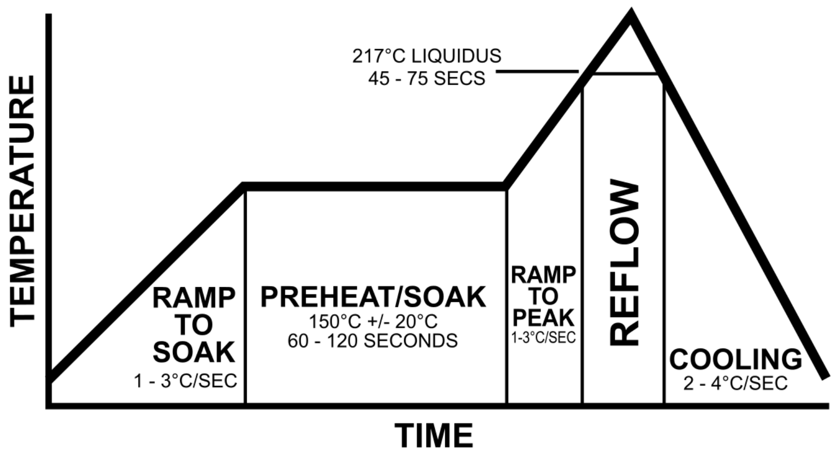


Figure 5.2: Typical reflow profile. Courtesy of [122].

onto the pads using either a stencil or special equipment [121]. Once the solder paste is applied the component is placed. The solder paste then functions as a glue, keeping the component in place. Next, the PCB is placed in an oven. The oven follows a reflow profile, which determines the temperature over time. A reflow profile's exact characteristics depend on factors such as the type of solder paste, PCB material and thickness and type of surface mount components. However, all profiles have the same four phases as illustrated in figure 5.2, which are:

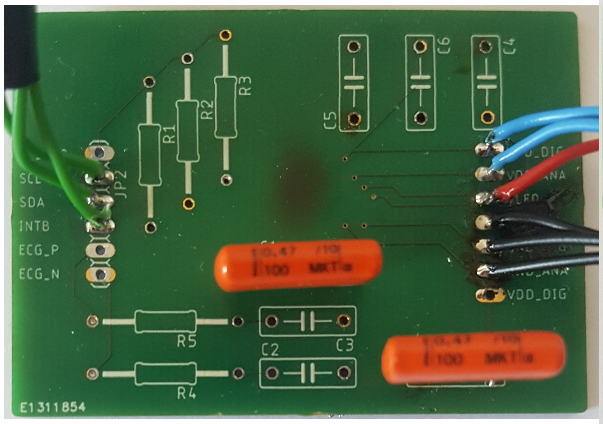
1. Pre-heating phase: the PCB, components and solder paste are gradually heated to a specific temperature. Increasing the temperature too quick may cause defects in the components.

2. Soak: the temperature is kept steady for 60 to 120 seconds to ensure all components have the same temperature, regardless of thickness.

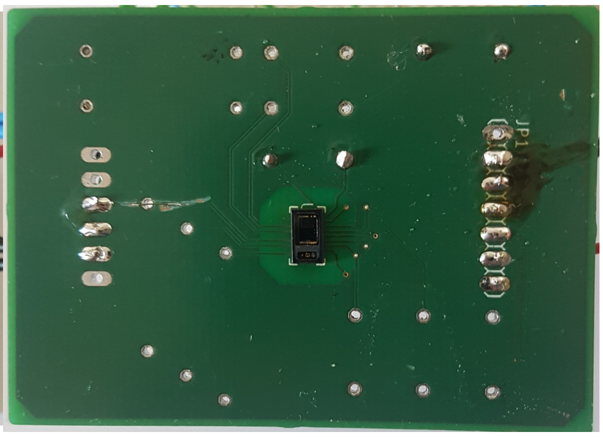
3. Reflow phase: the temperature is increased to above the melting point of the solder paste allowing it to form a liquid. The temperature is held above the melting point for 45 to 75 seconds to ensure the solder properly adheres to the components and PCB. Some components might fail when the temperature rises above a given threshold, therefore the peak temperature must be properly controlled during the reflow phase.

4. Cooling phase: the PCB and components are gradually cooled. If the temperature drops too quick cracks may form in the solder joints.

To verify whether any solder bridges have formed after soldering the chip, a digital power supply is used as opposed to measuring the resistance with a multimeter. This choice is made due to the Max86150 having an absolute maximum rating of 2.2 V on each pin, except for the V_{led} pin. The resistance measurement with the multimeter is performed using a voltage of 2.65 V, and would therefore damage the Max86150. On the digital power supply the voltage is set to 1.8 V. The current is limited to 20 mA, which is the absolute maximum rating of the Max86150. The positive and negative leads are then connected to all adjacent pad pairs one by one. Solder bridges would create a short circuit, and cause the current to reach 20 mA. Therefore, if the current is below 20 mA no solder bridge is present. When no solder bridges are found the remaining components can be soldered onto the PCB, and the Max86150 is ready for use. Only two capacitors that are required for the internal functioning of the Max86150 are soldered onto the PCB. Therefore, the dimensions of the PCB are 60 mm x 43 mm x 16 mm, or 2580 mm^2 x 16 mm. Figure 5.3 shows the completed PCB. The other components are placed on a breadboard as shown in figure 5.4 as this simplifies the testing procedure.



(a) The top side of the completed PCB.



(b) The bottom side of the completed PCB.

Figure 5.3: The front and back side of the completed PCB.

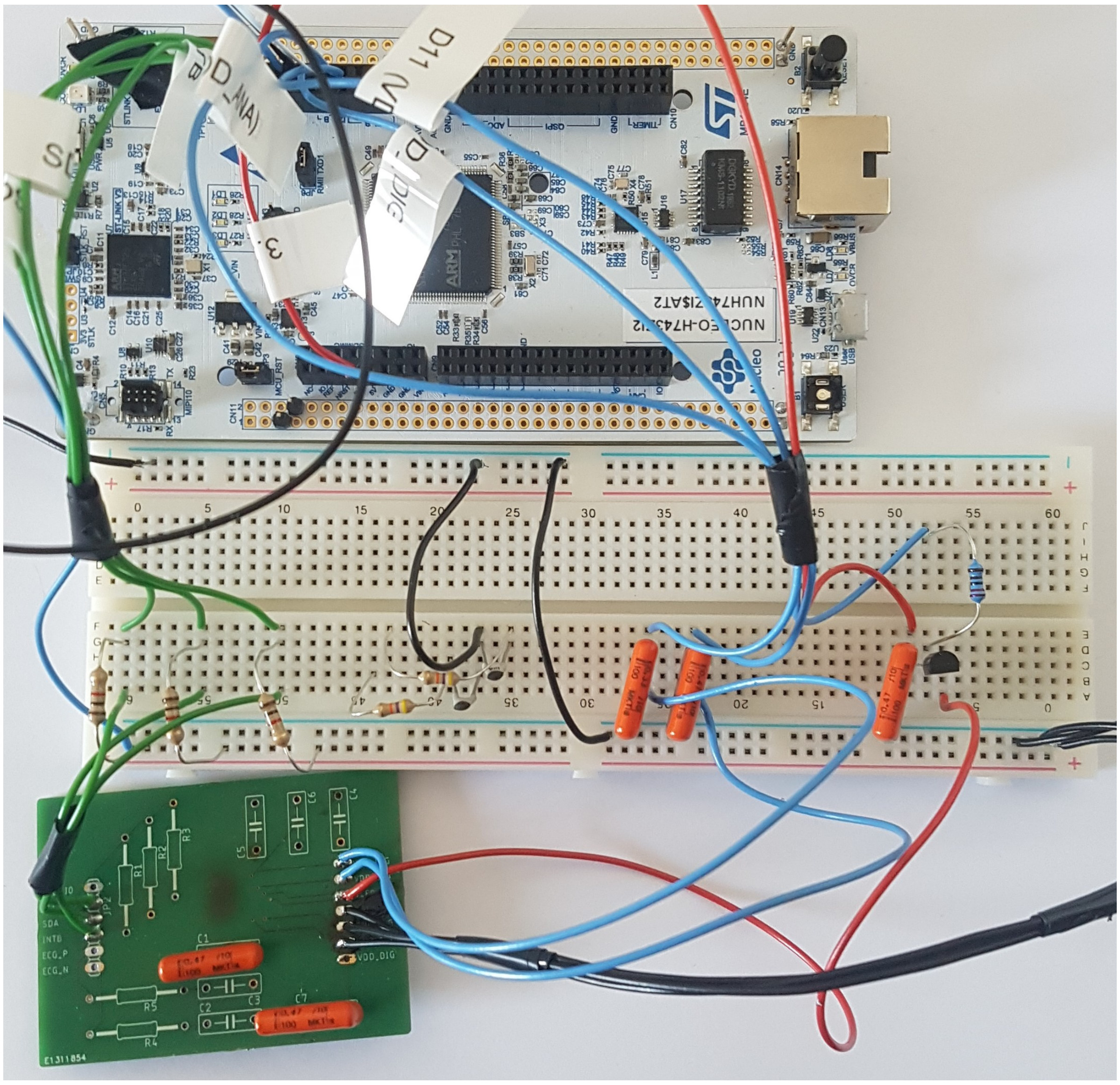


Figure 5.4: The complete circuit with the Max86150 chip connected to the development board.

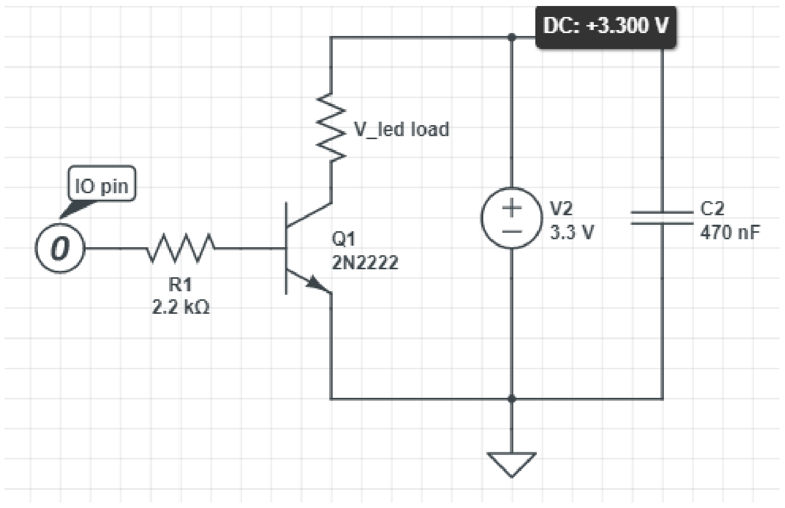


Figure 5.5: The V_{led} switching circuit

To allow the development board to switch the V_{led} power supply on and off, a BJT NPN transistor is used. This way the 1.8 V signal of the I/O pins can be used to turn the 3.3 V V_{led} power supply on and off. The circuit used is illustrated in figure 5.5. The capacitor C2 is added as a bypass capacitor, filtering out AC voltage signals on the 3.3 V line.

5.2. Software implementation

The Max86150 driver is compiled using Mbed Studio. Within Mbed Studio the target "NUCLEO-H743ZI2" can be selected, and the required files are then automatically included at compile time. The compiled program is saved as a binary. The development board is recognized as a storage device when connected to the PC with a USB cable. To flash the program to the development board, the binary file is copied to its root directory. To validate the proper operation of the driver software a test program is used (see Appendix A.4). The function `void testall()` initializes an object of the Max86150 class, and then calls each method of the Max86150 class. The output of the development board inside the Mbed Studio terminal is then analyzed and used to determine if the driver works properly. In appendix A.4 the comments after each method call contain the expected results. The output of the test program is shown in appendix A.5. Note that the warning-statements are intended to verify whether warnings are correctly prompted in case invalid arguments are given within the code.

5.2.1. PPG recording verification

The control class module could not be finished due to time constraints. However, the basic functionality required of the control class has been implemented. This allows a PPG recording to be made, and output over the USB connection. The code is listed in appendix A.6. The program flow is as follows:

1. The Max86150 driver is initialized
2. A new thread is created to move certain function calls out of the interrupt service routine (ISR) context.
3. A queue is created and attached to the thread. This allows function calls to be added to the queue, which are then handled by Mbed OS.
4. The Max86150 settings are configured.

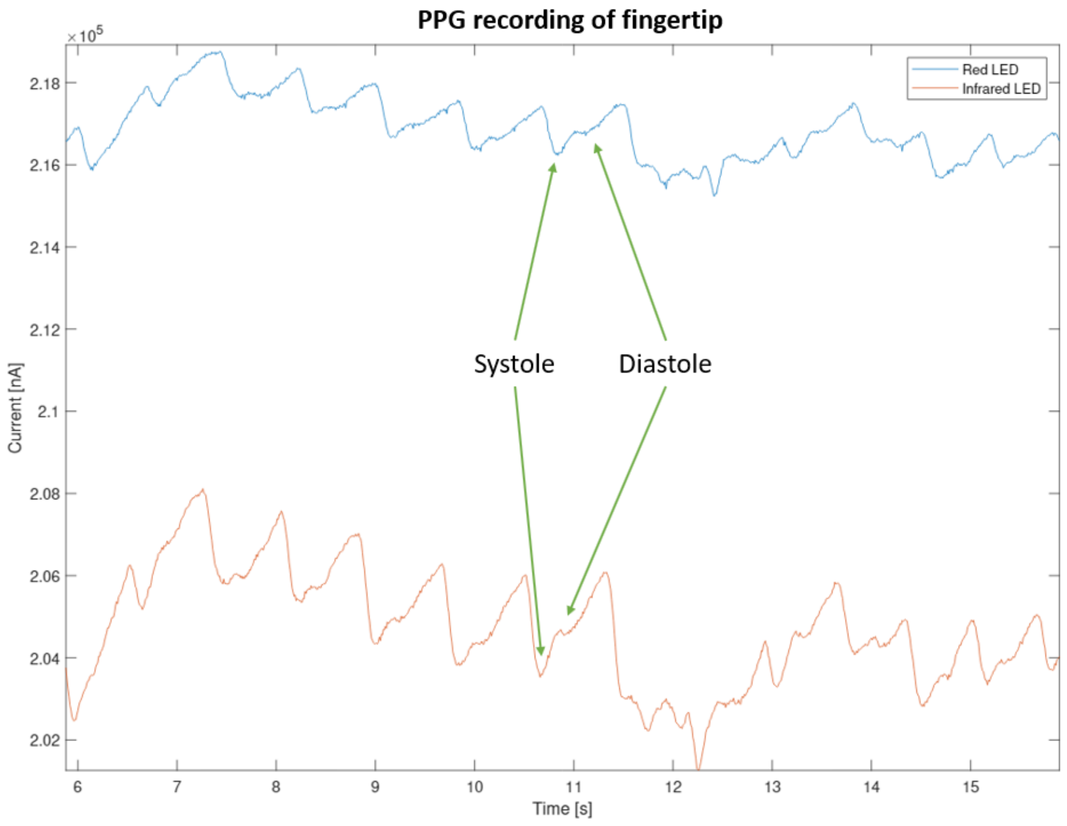


Figure 5.6: PPG recording of the tip of the middle finger at rest with a red and infrared LED.

5. The Red and Infrared LED's are enabled.

6. An interrupt pin is created, and a callback function is attached to the falling edge event of this pin. This callback function reads the interrupt cause from the Max86150 chip. If the cause is A_FULL (which indicates the FIFO buffer exceeded the set value) the FIFO buffer is read, and the values are then stored in the recording array of the corresponding sensor.

7. 30 seconds of data is gathered, and then transmitted over the USB connection as comma-separated values.

The recording shown in figure 5.6 is made while keeping the tip of the middle finger still and lightly pressed against the glass cover of the Max86150. The complete 30-second recording can be found in appendix C. The following settings were used:

- PPG sample rate: 200 Hz
- Sample averaging: 2
- Red LED pulse amplitude: 5 mA
- LED pulse width: 50 µs
- PPG ADC range: 32768 nA

Both waveforms exhibit a clear indication of systole and diastole, as annotated in the figure. The average peak-to-peak AC component of red light PPG is 1300 nA. For infrared light this is 2000 nA.

6 Discussion

The aim of this thesis was to develop a system capable of continuously monitoring a subject's heart activity using PPG technology and requesting an ECG recording in the event that an AF episode has been detected in a PPG recording. For this, the Maxim Integrated Max86150 chip has been selected as it offers the hardware for both technologies integrated in one form factor. Although the chip itself is compact, the dimensions of the PCB including the additional components are 2580 mm^2 x 16 mm. Hence, the device does not satisfy requirement **PoR:7.1**. Nevertheless, in order to meet this requirement, the size of the PCB can easily be reduced by, for example, using SMD components for the capacitors and resistors.

From the output of the test program presented in section 5.2, it may be concluded that the driver developed for the chip has been implemented successfully. The I^{2}C communication between the development board and the chip is established correctly since each data segment is acknowledged. The power up sequence of the chip evolves properly since the `PWR_RDY` interrupt is initiated directly after the sequence has been completed. All adjustable configurations of the chip have been evaluated to verify whether the driver correctly recognizes invalid value designations. As can be seen from the test program output, the driver appropriately displays a `TRACE` level statement when valid values are assigned. `WARNING` level statements are properly printed when a value is out of range.

A PPG signal has been obtained with the the Max86150 chip as has been demonstrated in section 5.2.1. The PPG waveform of red and infrared light both display a clear indication of systole and diastole. Furthermore, the peak-to-peak value of the AC component of red light PPG is approximately 1300 nA. For infrared light PPG this is approximately 2000 nA. Therefore, requirement **PoR:1.2** has been successfully satisfied. The discrepancy between the magnitude of AC components may be explained by the difference in penetration depth of red light and infrared light. As infrared light penetrates deeper into tissue due to its larger wavelength, it reaches blood vessels deeper inside the finger. Since these blood vessels are larger than peripheral blood vessels, the variation in blood density is more substantial. As a result, the reflected light intensity fluctuates to a greater extent.

As this PPG recording was made at rest, further verification is required to evaluate the PPG signal quality during motion of the user. Baseline wandering is clearly present and minor motion artefacts can be identified in the recording as well. It is expected that the waveform will look more distorted with increased motion. Nevertheless, as the signal must still be processed and filtered by the signal processing subsystem, the current quality of the PPG signal may suffice.

Unfortunately, due to time constraints, complete functionality of the chip could not be verified as no control class has been written for efficient control of the chip and transmission of the data it retrieves. As a result, the ECG recording capabilities of the chip have not been verified and requirement **PoR:1.1** is therefore not fully satisfied. Nevertheless, since each of its subrequirements with the exception of requirements **PoR:1.1.1** and **PoR:1.1.5** have been achieved with the chip and the chip appears to be fully operational, it is expected that verification of the ECG recording is feasible in the near future.

Although an accelerometer has been considered during the design, insufficient time has led to the exclusion of its implementation. Therefore, requirement **PoR:2.1** could not have been realized. Since appropriate software resources for the accelerometer are available, implementing it will be less cumbersome as opposed to the implementation of the Max86150 chip. A challenge that would remain, however, is the integration of both sensor modules. To ensure synchronization of the data, communication between the development board and each module must be in parallel. However, since I^2C is a serial communication protocol, this may introduce a conflict. Hence, it is insightful to investigate the unification of the sensors into one module similarly to the integration of ECG and PPG into the Max86150. In addition to simplifying the data flow and communication within the system, this could also result in a physically more compact system.

As the functionality of the chip has not been fully tested, no feedback system has been implemented yet. Therefore, requirements **PoR:4.1** and **PoR:4.2** have not been attained. A similar conclusion may be drawn for requirement **PoR:7.2** and **PoR:7.3** because the power consumption of the device could not be verified empirically and the system could not be tested using a battery.

In terms of the costs, the complete system has a cumulative value of €83.68. However, excluding the PCB and development board as these are intended for the development of the device, this reduces to €12.20 for only the Max86150 chip and MMA8451 accelerometer, thereby satisfying requirement **PoR:9.1**.

6.1. Limitations

Several limitations have been experienced throughout the course of the project. These limitations delayed certain objectives and restricted the completion of others.

One of the prominent limitations was the soldering of the Max86150 chip. Because of limited financial resources, the chip could not have been soldered by the PCB supplier and had to be soldered manually. Unforeseen challenges were introduced due to inadequate solder equipment. Repeated attempts were unsuccessful and cost a substantial amount of time. Eventually, professional assistance was sought from an external company, which had the appropriate equipment and properly soldered the chip.

Another limitation emerged from the proprietary drive documentation of Maxim Integrated. Since the driver for the chip was not clearly documented, it was unclear how the chip should be operated. To avoid potential hidden complications amidst the project, it had been decided to develop a new, properly documented driver. Though this has been proven to be successful, it was at the cost of the available time for the project.

7 Conclusion

The goal of this project was to acquire information about a subject's heart activity and motion in a non-invasive and continuous manner. To monitor the heart activity, ECG and PPG technology have been implemented by means of the Maxim Integrated Max86150 integrated circuit. An adapter PCB has been created to implement the chip together with other electronic components required for its functioning, which allows for easy interfacing with a development board. Additionally, a well-documented driver for the Max86150 has been developed. The PPG signal acquired from the Max86150 chip is of high quality when this recording is made at rest, and usage of the infrared LED resulted in a higher AC component than usage of the red LED. Extensive verification of the Max86150 chip's settings and performance on different body sites could not be realized within the given time frame. Therefore, the chip's ECG capabilities could not be verified. An equivalent conclusion applies to the implementation of the accelerometer. Nevertheless, it is expected that it would take at most two weeks to make these functionalities fully operational.

7.1. Future work and recommendations

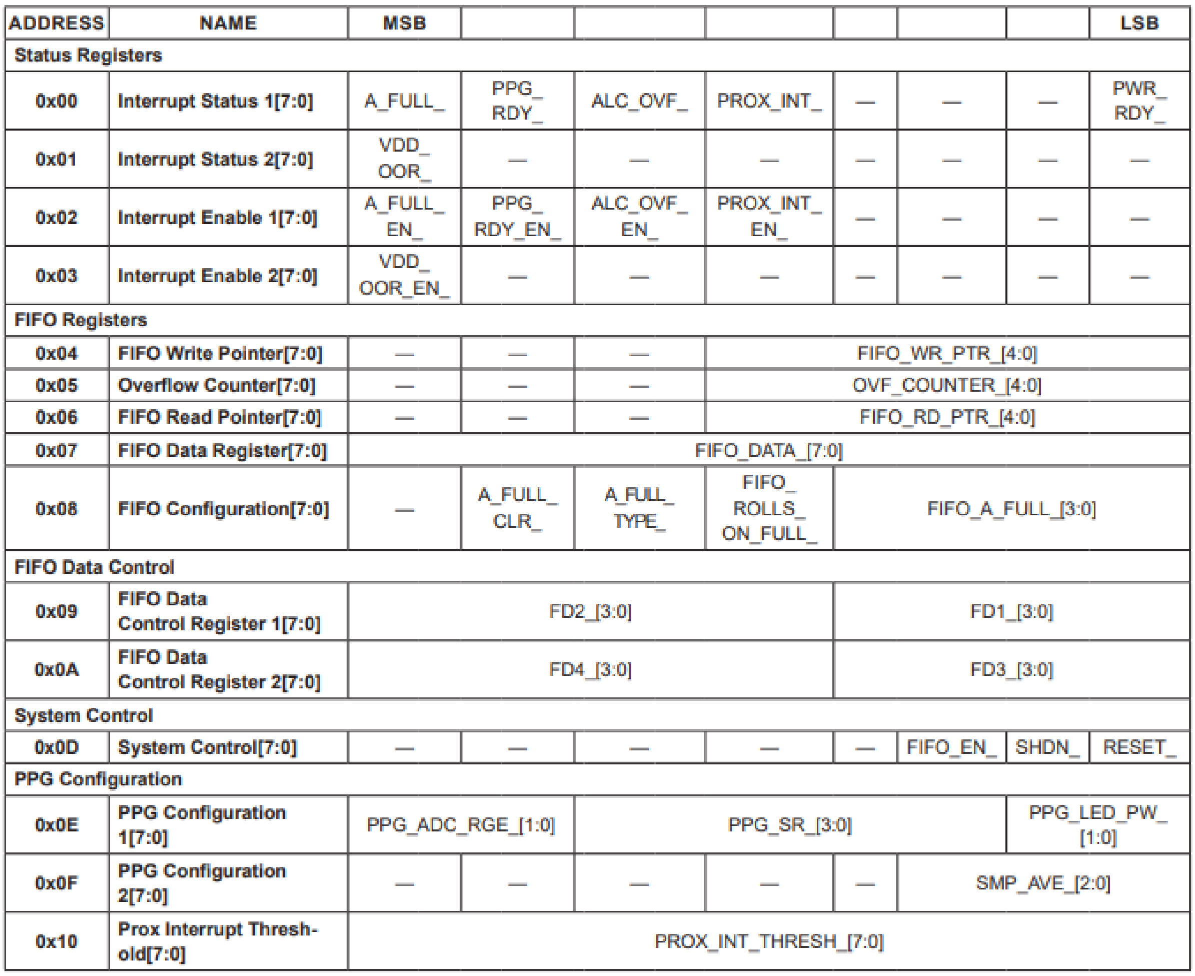
In addition to the functionalities focused on in this thesis, it is insightful to suggest improvements for future iterations. These include, but are not limited to:

- **Additional sensors to alleviate MAs.** For example, a piezoelectric sensor could detect minor changes in the pressure of the PPG sensor or the ECG electrodes on the body surface. Alternatively, the ECG electrodes could be used for skin-electrode impedance measurements.
- **Integration into one form factor.** Though the current device is still in its early development phase, it is advisable to consider the potential of hardware integration with the aim to minimize the device's dimensions. In particular, it would be interesting to consider a system-on-chip as final product with the aim to add it to existing wearables.
- **Improvement of ECG electrodes.** It is preferred to continuously monitor heart activity by means of ECG over PPG since the former is considered the gold standard for assessing AF. A next step would be to improve the practical implementation of electrodes for continuous and daily usage. If this would enable monitoring of leads other than limb lead I, other cardiac arrhythmia's could be detected as well, which would enrich the application of the device.

A

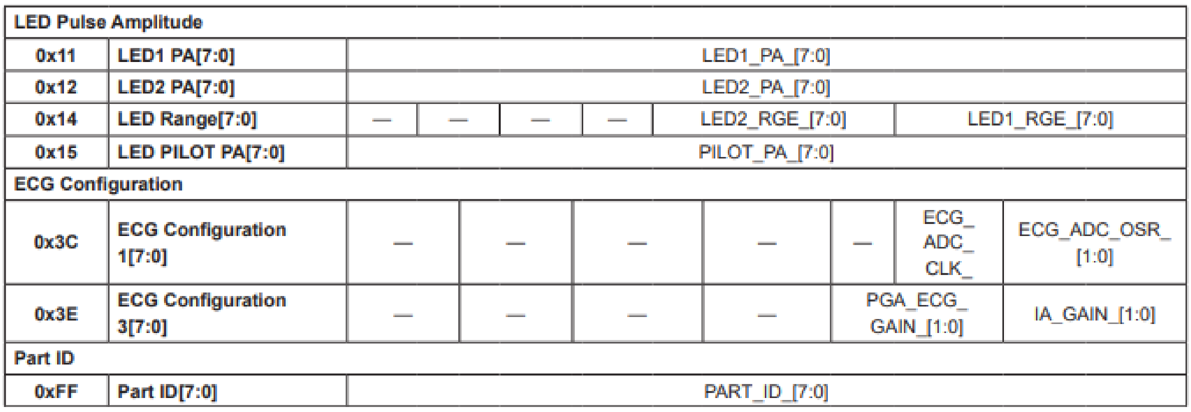
Max86150 Driver Documentation

A.1. Max86150 register map



ADDRESS	NAME	MSB							LSB
Status Registers									
0x00	Interrupt Status 1[7:0]	A_FULL_	PPG_RDY_	ALC_OVF_	PROX_INT_	—	—	—	PWR_RDY_
0x01	Interrupt Status 2[7:0]	VDD_OOR_	—	—	—	—	—	—	—
0x02	Interrupt Enable 1[7:0]	A_FULL_EN_	PPG_RDY_EN_	ALC_OVF_EN_	PROX_INT_EN_	—	—	—	—
0x03	Interrupt Enable 2[7:0]	VDD_OOR_EN_	—	—	—	—	—	—	—
FIFO Registers									
0x04	FIFO Write Pointer[7:0]	—	—	—	FIFO_WR_PTR_[4:0]				
0x05	Overflow Counter[7:0]	—	—	—	OVF_COUNTER_[4:0]				
0x06	FIFO Read Pointer[7:0]	—	—	—	FIFO_RD_PTR_[4:0]				
0x07	FIFO Data Register[7:0]	FIFO_DATA_[7:0]							
0x08	FIFO Configuration[7:0]	—	A_FULL_CLR_	A_FULL_TYPE_	FIFO_ROLLS_ON_FULL_	FIFO_A_FULL_[3:0]			
FIFO Data Control									
0x09	FIFO Data Control Register 1[7:0]	FD2_[3:0]				FD1_[3:0]			
0x0A	FIFO Data Control Register 2[7:0]	FD4_[3:0]				FD3_[3:0]			
System Control									
0x0D	System Control[7:0]	—	—	—	—	—	FIFO_EN_	SHDN_	RESET_
PPG Configuration									
0x0E	PPG Configuration 1[7:0]	PPG_ADC_RGE_[1:0]		PPG_SR_[3:0]				PPG_LED_PW_[1:0]	
0x0F	PPG Configuration 2[7:0]	—	—	—	—	—	SMP_AVE_[2:0]		
0x10	Prox Interrupt Threshold[7:0]	PROX_INT_THRESH_[7:0]							

Figure A.1: Register map from Max86150 datasheet. [113]



LED Pulse Amplitude									
0x11	**LED1 PA[7:0]**	LED1_PA_[7:0]							
0x12	**LED2 PA[7:0]**	LED2_PA_[7:0]							
0x14	**LED Range[7:0]**	—	—	—	—	LED2_RGE_[7:0]		LED1_RGE_[7:0]	
0x15	**LED PILOT PA[7:0]**	PILOT_PA_[7:0]							
ECG Configuration									
0x3C	**ECG Configuration 1[7:0]**	—	—	—	—	—	ECG_ ADC_ CLK_	ECG_ADC_OSR_ [1:0]	
0x3E	**ECG Configuration 3[7:0]**	—	—	—	—	PGA_ECG_ GAIN_[1:0]		IA_GAIN_[1:0]	
Part ID									
0xFF	**Part ID[7:0]**	PART_ID_[7:0]							

Figure A.2: Continued register map from Max86150 datasheet. [113]

A.2. Max86150.h

```
/*
Author: Alex Smit
Author: Amar Kohabir
*/

#ifndef MAX86150_H
#define MAX86150_H
#include "mbed.h"
#include <array>
#include <cstdint>
#include <stdio.h>
#include <map>
#include "logger.h"
#include "config.h"
#include <EventQueue.h>

//! The number of tries for an i2c request.
#define I2C_RETRY_LIMIT    3

namespace driver{
    class Max86150{
        private:
            const uint8_t SLAVE_ADDRESS = 0xBC; //1011110[W]
            I2C i2c;
            InterruptIn intb;
            AFLogger::Logger *logger;
            EventQueue *queue;
            DigitalOut vdd_dig;
            DigitalOut vdd_ana;
            DigitalOut vdd_led;

            // These are the addresses for the MAX86150 registers
            typedef enum RegisterAddresses{
                INTERRUPT_STATUS_1 = 0x00,
                INTERRUPT_STATUS_2 = 0x01,
                INTERRUPT_ENABLE_1 = 0x02,
                INTERRUPT_ENABLE_2 = 0x03,

                FIFO_WRITE_POINTER = 0x04,
                OVERFLOW_COUNTER = 0x05,
                FIFO_READ_POINTER = 0x06,
                FIFO_DATA_REGISTER = 0x07,
                FIFO_CONFIGURATION = 0x08,

                FIFO_DATA_CONTROL_REGISTER_1 = 0x09,
                FIFO_DATA_CONTROL_REGISTER_2 = 0x0A,

                SYSTEM_CONTROL = 0x0D,

                PPG_CONFIGURATION_1 = 0x0E,
                PPG_CONFIGURATION_2 = 0x0F,
                PROX_INTERRUPT_THRESHOLD = 0x10,

                LED1_PA = 0x11,
                LED2_PA = 0x12,
                LED_RANGE = 0x14,
                LED_PILOT_PA = 0x15,

                ECG_CONFIGURATION_1 = 0x3C,
                ECG_CONFIGURATION_3 = 0x3E,

                PART_ID = 0xFF
            }RegisterAddresses;

        public:
            /**
             * All the possible statusses that can be returned by member functions.
             */
            typedef enum ReturnCode{
                OK,
                ERROR,
                INVALID_DATA,
                I2C_NO_ACK,
                MAX_SENSORS_EXCEEDED,
                SENSOR_DISABLED,
                VALUE_OOR,
                RESETTING
            }ReturnCode;


            //! A list of the settings that can be written to the Max86150 to enable specific interrupts. The value of the setting corresponds to the bit to set

            typedef enum InterruptEnableConf{
                //! Set whether an interrupt is given when the FIFO buffer is almost full. Indicates that the FIFO buffer overflows the threshold set by FFIFO_A_FULL[3:0] on the next sample. This bit is cleared when the Interrupt Status 1 register is read. It is also cleared when FIFO_DATA register is read, if A_FULL_CLR = 1
                A_FULL_EN,
                //! Set whether an interrupt is given when new PPG data is ready. In SpO2 and HR modes, this interrupt triggers when there is a new sample in the data FIFO. The interrupt is cleared by reading the Interrupt Status 1 register (0x00), or by reading the FIFO_DATA register.
                PPG_RDY_EN,
                //! Set whether an interrupt is given when the ambient light cancelation analog to digital converter has an overflow, indicating the ambient light is to bright for usefull measurements. This interrupt triggers when the ambient light cancellation function of the SpO2/HR photodiode has reached its maximum limit due to overflow, and therefore, ambient light is affecting the output of the ADC. The interrupt is cleared by reading the Interrupt Status 1 register (0x00).
                ALC_OVF_EN,
                //! Set whether an interrupt is given when the proximity threshold is crossed. The level at which this triggers can be configured by Max86150::setProximityThreshold().
                PROX_INT_EN,
                //! Set whether an interrupt is given when the power supply voltage is out of range. Indicated that VDD_ANA is greater than 2.05V or less than 1.65V. This bit is automatically cleared when the Interrupt Status 2 register is read. The detection circuitry has a 10ms delay time and continues to trigger as long as the VDD_ANA is out of range.
                VDD_OOR_EN,
                //! Set whether an interrupt is given when new ECG data is ready to be read from the FIFO buffer.
                ECG_RDY_EN
            }InterruptEnableConf;

            //! A list of bitmasks to AND with the interrupt status register contents and obtain the corresponding interrupt cause
            typedef enum InterruptStatus{
                A_FULL      = 0b10000000,
                PPG_RDY     = 0b01000000,
                ALC_OVF     = 0b00100000,
                PROX_INT    = 0b00010000,
                PWR_RDY     = 0b00000001,
                VDD_OOR     = 0b10000001, // added 1 as to indicate this status is in the second register
                ECG_RDY     = 0b00000101, // added 1 as to indicate this status is in the second register
                NONE        = 0
            }InterruptStatus;

            //! A list of sensors that can be written to the fifo data element register. Doing so enables this sensor.
            typedef enum FifoSensorsConf{
                SENSOR_NONE     = 0b0000,
                LED1_IR         = 0b0001,
                LED2_RED        = 0b0010,
                PILOT_LED1_IR   = 0b0101,
                PILOT_LED2_RED  = 0b0110,
                ECG             = 0b1001
            }FifoSensorsConf;

            //! The PPG_ADC_RGE values that can be set.
            typedef enum PpgAdcRangeConf{
                NANO_AMPERE_4096 = 0b00,
                NANO_AMPERE_8192 = 0b01,
                NANO_AMPERE_16384 = 0b10,
                NANO_AMPERE_32768 = 0b11
            }PpgAdcRangeConf;

            //! The sample rates that can be set for the PPG_SR with the amount
             ↪  of pulses per second.
            typedef enum PpgSampleRateConf{
                SAMPLES_10_PPS_1    = 0x0,
                SAMPLES_20_PPS_1    = 0x1,
                SAMPLES_50_PPS_1    = 0x2,
                SAMPLES_84_PPS_1    = 0x3,
                SAMPLES_100_PPS_1   = 0x4,
                SAMPLES_200_PPS_1   = 0x5,
                SAMPLES_400_PPS_1   = 0x6,
                SAMPLES_800_PPS_1   = 0x7,
                SAMPLES_1000_PPS_1  = 0x8,
                SAMPLES_1600_PPS_1  = 0x9,
                SAMPLES_3200_PPS_1  = 0xA,
                SAMPLES_10_PPS_2    = 0xB,
                SAMPLES_20_PPS_2    = 0xC,
                SAMPLES_50_PPS_2    = 0xD,
                SAMPLES_84_PPS_2    = 0xE,
                SAMPLES_100_PPS_2   = 0xF
            }PpgSampleRateConf;

            //! The different types of LED's, used in setting the pulse
             ↪  amplitude.
            typedef enum LedType{
                INFRA_RED,
                RED,
                PILOT
            }LedType;

            //! The pulse width of the PPG LED's in microseconds.
            typedef enum PpgPulseWidthConf{
                MICRO_50 = 0b00,
                MICRO_100 = 0b01,
                MICRO_200 = 0b10,
                MICRO_400 = 0b11
            }PpgPulseWidthConf;

```

```
        //! Configurations for the LED current range setting
        typedef enum LedCurrentRangeConf{
            MILLI_AMP_51    = 0b00,
            MILLI_AMP_102   = 0b01
        }LedCurrentRangeConf;

        //! The possible configuration that can be used in Configurations::SMP_AVE.
        typedef enum SampleAveragingConf{
            SAMPLE_AVERAGE_1 = 0b000,
            SAMPLE_AVERAGE_2 = 0b001,
            SAMPLE_AVERAGE_4 = 0b010,
            SAMPLE_AVERAGE_8 = 0b011,
            SAMPLE_AVERAGE_16 = 0b100,
            SAMPLE_AVERAGE_32 = 0b101
        }SampleAveragingConf;

       //! Configurations for setting the ECG ADC base clockrate.
        typedef enum EcgAdcSampleRateConf{
            ECG_CLK_200 = 0b0,
            ECG_CLK_400 = 0b1
        }EcgAdcSampleRateConf;

        //!Configurations for setting the ECG oversampling ratio.
        typedef enum OverSamplingRatioConf{
            OSR_16  = 0b00,
            OSR_8   = 0b01,
            OSR_4   = 0b10,
            OSR_2   = 0b11
        }OverSamplingRatioConf;

        //! Configurations for setting the ECG gain.
        typedef enum EcgGainConf{
            ECG_GAIN_1 = 0b00,
            ECG_GAIN_2 = 0b01,
            ECG_GAIN_4 = 0b10,
            ECG_GAIN_8 = 0b11
        }EcgGainConf;

        //! Configurations for setting the programmable gain amplifier gain.
        typedef enum PgaGainConf{
            PGA_GAIN_1 = 0b00,
            PGA_GAIN_2 = 0b01,
            PGA_GAIN_4 = 0b10,
            PGA_GAIN_8 = 0b11
        }PgaGainConf;

        //! Configurations for setting the Instrumentation Amplifier Gain.
        typedef enum InstrAmpGainConf{
            IA_GAIN_5   =0b00,
            IA_GAIN_9_5 =0b01,
            IA_GAIN_20  =0b10,
            IA_GAIN_50  =0b11
        }InstrumentationAmplifierGainConf;

    private:
        static const uint8_t FIFO_BUFFER_SIZE = 32;
        static const uint8_t ELEMENT_SAMPLE_SIZE = 3; // Number of bytes per element sample.
        static const uint8_t FIFO_MAX_DATA_ELEMENTS = 4;

        struct Setting{
            Max86150::RegisterAddresses address;
            uint8_t bitmask;
        };

        /**
         * The FifoDataElement structure stores the information regarding the sensors.
         * Once a sensor has been loaded into the FifoDataElement register on the Max86150,
         * the read_buffer pointers are updated, to make sure they point to the samples regarding
         * this specific sensor. These samples can then be read by checking how many new samples are available.
         */
        struct FifoDataElement{
            //! Points to the first byte of the coinciding sample in the fifo_buffer private variable.
            char* read_buffer[32];
            //! Indicates what sensor is set to this fd element.
            FifoSensorsConf sensor = Max86150::FifoSensorsConf::SENSOR_NONE;
            //! Stores the number of new samples that have been read.
            uint8_t new_samples = 0;
        };

        //! The different data elements.
        FifoDataElement fd1, fd2, fd3, fd4;
        //! Array of pointers to the different data element structs defined above.
        FifoDataElement *data_elements[4] = {&fd1, &fd2, &fd3, &fd4};

        //! Read buffer for all the fifo data.
        char fifo_buffer[(FIFO_BUFFER_SIZE*ELEMENT_SAMPLE_SIZE*FIFO_MAX_DATA_ELEMENTS)];

        //! Keeps track of which sensors are enabled.
        std::map<FifoSensorsConf, bool> sensorStatus = {
            {FifoSensorsConf::LED1_IR,          false},
            {FifoSensorsConf::LED2_RED,         false},
            {FifoSensorsConf::PILOT_LED1_IR,    false},
            {FifoSensorsConf::PILOT_LED2_RED,   false},
            {FifoSensorsConf::ECG,              false}
        };

        uint8_t active_data_elements = 0;               // Keeps track of the number of sensors writing data to the Fifo.

        bool resetting = false;

        std::map<Max86150::RegisterAddresses, uint8_t> registerContents = {
            {Max86150::INTERRUPT_STATUS_1,              0x00},
            {Max86150::INTERRUPT_STATUS_2,              0x00},
            {Max86150::INTERRUPT_ENABLE_1,              0x00},
            {Max86150::INTERRUPT_ENABLE_2,              0x00},
            {Max86150::FIFO_WRITE_POINTER,              0x00},
            {Max86150::OVERFLOW_COUNTER,                0x00},
            {Max86150::FIFO_READ_POINTER,               0x00},
            {Max86150::FIFO_DATA_REGISTER,              0x00},
            {Max86150::FIFO_CONFIGURATION,              0x0F},
            {Max86150::FIFO_DATA_CONTROL_REGISTER_1,    0x00},
            {Max86150::FIFO_DATA_CONTROL_REGISTER_2,    0x00},
            {Max86150::SYSTEM_CONTROL,                  0x00},
            {Max86150::PPG_CONFIGURATION_1,             0x00},
            {Max86150::PPG_CONFIGURATION_2,             0x00},
            {Max86150::PROX_INTERRUPT_THRESHOLD,        0x00},
            {Max86150::LED1_PA,                         0x00},
            {Max86150::LED2_PA,                         0x00},
            {Max86150::LED_RANGE,                       0x00},
            {Max86150::LED_PILOT_PA,                    0x00},
            {Max86150::ECG_CONFIGURATION_1,             0x00},
            {Max86150::ECG_CONFIGURATION_3,             0x02},
            {Max86150::PART_ID,                         0x1E}
        };

        /**
         * The settingsMap is an struct containing constant setting structs:
         * {registerAdress, bitMask}
         * The registerAdress is the address of the register where the specified setting can be found.
         * The bitMask is the mask used to specify which bits in the register control the specified setting,
         *
         */
        struct settingsMap{
            const Setting _A_FULL_EN           =
             ↪  {RegisterAddresses::INTERRUPT_ENABLE_1,
             ↪  0b10000000};
            const Setting _PPG_RDY_EN          =
             ↪  {RegisterAddresses::INTERRUPT_ENABLE_1,
             ↪  0b01000000};
            const Setting _ALC_OVF_EN          =
             ↪  {RegisterAddresses::INTERRUPT_ENABLE_1,
             ↪  0b00100000};
            const Setting _PROX_INT_EN         =
             ↪  {RegisterAddresses::INTERRUPT_ENABLE_1,
             ↪  0b00010000};

            const Setting _VDD_OOR_EN          =
             ↪  {RegisterAddresses::INTERRUPT_ENABLE_2,
             ↪  0b10000000};
            const Setting _ECG_RDY_EN          =
             ↪  {RegisterAddresses::INTERRUPT_ENABLE_2,
             ↪  0b00000100};

            const Setting _FIFO_WR_PTR         =
             ↪  {RegisterAddresses::FIFO_WRITE_POINTER,
             ↪  0b00011111};

            const Setting _OVF_COUNTER         =
             ↪  {RegisterAddresses::OVERFLOW_COUNTER,
             ↪  0b00011111};

            const Setting _FIFO_RD_PTR         =
             ↪  {RegisterAddresses::FIFO_READ_POINTER,
             ↪  0b00011111};

            const Setting _FIFO_DATA           =
             ↪  {RegisterAddresses::FIFO_DATA_REGISTER,
             ↪  0b11111111};


            const Setting _A_FULL_CLR          =
             ↪  {RegisterAddresses::FIFO_CONFIGURATION,
             ↪  0b01000000};
            const Setting _A_FULL_TYPE         =
             ↪  {RegisterAddresses::FIFO_CONFIGURATION,
             ↪  0b00100000};
            const Setting _FIFO_ROLLS_ON_FULL  =
             ↪  {RegisterAddresses::FIFO_CONFIGURATION,
             ↪  0b00010000};
            const Setting _FIF_A_FULL          =
             ↪  {RegisterAddresses::FIFO_CONFIGURATION,
             ↪  0b00001111};

            const Setting _FD1                  =
             ↪  {RegisterAddresses::FIFO_DATA_CONTROL_REGISTER_1,
             ↪  0b00001111};
            const Setting _FD2                  =
             ↪  {RegisterAddresses::FIFO_DATA_CONTROL_REGISTER_1,
             ↪  0b11110000};
            const Setting _FD3                  =
             ↪  {RegisterAddresses::FIFO_DATA_CONTROL_REGISTER_2,
             ↪  0b00001111};
            const Setting _FD4                  =
             ↪  {RegisterAddresses::FIFO_DATA_CONTROL_REGISTER_2,
             ↪  0b11110000};

            const Setting _FIFO_EN              =
             ↪  {RegisterAddresses::SYSTEM_CONTROL,
             ↪  0b00000100};
            const Setting _SHDN                 =
             ↪  {RegisterAddresses::SYSTEM_CONTROL,
             ↪  0b00000010};
            const Setting _RESET                =
             ↪  {RegisterAddresses::SYSTEM_CONTROL,
             ↪  0b00000001};

            const Setting _PPG_ADC_RGE          =
             ↪  {RegisterAddresses::PPG_CONFIGURATION_1,
             ↪  0b11000000};
            const Setting _PPG_SR               =
             ↪  {RegisterAddresses::PPG_CONFIGURATION_1,
             ↪  0b00111100};
            const Setting _PPG_LED_PW           =
             ↪  {RegisterAddresses::PPG_CONFIGURATION_1,
             ↪  0b00000011};

            const Setting _SMP_AVE              =
             ↪  {RegisterAddresses::PPG_CONFIGURATION_2,
             ↪  0b00000111};

            const Setting _PROX_INT_THRESH      =
             ↪  {RegisterAddresses::PROX_INTERRUPT_THRESHOLD,
             ↪  0b11111111};

            const Setting _LED1_PA_CONF         =
             ↪  {RegisterAddresses::LED1_PA,
             ↪  0b11111111};
            const Setting _LED2_PA_CONF         =
             ↪  {RegisterAddresses::LED2_PA,
             ↪  0b11111111};

```

```
                const Setting _LED2_RGE          =
                 ↪  {RegisterAddresses::LED_RANGE,
                 ↪  0b00001100};
                const Setting _LED1_RGE          =
                 ↪  {RegisterAddresses::LED_RANGE,
                 ↪  0b00000011};

                const Setting _PILOT_PA          =
                 ↪  {RegisterAddresses::LED_PILOT_PA,
                 ↪  0b11111111};

                const Setting _ECG_ADC_CLK       =
                 ↪  {RegisterAddresses::ECG_CONFIGURATION_1,
                 ↪  0b00000100};
                const Setting _ECG_ADC_OSR       =
                 ↪  {RegisterAddresses::ECG_CONFIGURATION_1,
                 ↪  0b00000011};

                const Setting _PGA_ECG_GAIN      =
                 ↪  {RegisterAddresses::ECG_CONFIGURATION_3,
                 ↪  0b00001100};
                const Setting _IA_GAIN           =
                 ↪  {RegisterAddresses::ECG_CONFIGURATION_3,
                 ↪  0b00000011};
            }settings;

            /**
             * Read data from a specific register.
             *
             * To read data from a register, first a i2c write command has to
↪   be given, followed by the
             * register address. Then a repeated start condition has to be
↪   given. After that read command
             * is sent, no acknoledge bit is needed from the master. To end the
↪   read the stop condition
             * is given.
             * @param address the register to be read from.
             * @param read_buffer pointer to the buffer where data is stored.
             * @param repeat indicates the number of sequential read operations
↪   to perform.
             * @return returns a Max86150::ReturnCode.
             */
            Max86150::ReturnCode readRegister(Max86150::RegisterAddresses
             ↪  address, char* read_buffer, uint8_t repeat=1);

            /**
             * Write a byte of data to a specific register.
             *
             * The register that is written to cannot be in the
↪   REGISTER_READ_ONLY list.
```

```
         * @param address The register address to be written to.
         * @param data byte to be written to the register.
         * @return returns a Max86150::ReturnCode.
         */
        Max86150::ReturnCode writeRegister(Max86150::RegisterAddresses address, uint8_t data);

        /**
        * Determine the data to be written to a register for boolean interrupts and settings.
        *
        * This function determines the data to be sent to registers for interrupts and settings that can be enabled/disabled (i.e. boolean).
        * @param enable true to enable interrupt/setting, false to disable interrupt/setting.
        * @param bitmask The bitmask of the interrupt/setting within its respective register.
        * @param current_data The data currently present in the interrupt's/setting's register.
        * @return Data to be written to the interrupt's/setting's register.
        */
        uint8_t makeWriteRegisterData(bool enable, uint8_t bitmask, uint8_t current_data);

        /**
        * Return a Max86150::ReturnCode after attempting to write data to register.
        *
        * Max86150::ReturnCode::OK is returned if the data has been successfully written to the register and the RAM has been updated accordingly.
        * Max86150::ReurnCode::I2C_NO_ACK is returned if the data has not successfully been written to the register.
        * @param specific_setting The setting to be adjusted. For this function, only its register address is used.
        * @param data The data to be written to the register
        * @return returns a Max86150::ReturnCode.
        */
        Max86150::ReturnCode ackWriteRegister(Max86150::Setting specific_setting, uint8_t data);

        /**
        *
        * Write a boolean setting to the relevant register.
        *
        * Write the adjusted setting to its respective register.
        * This function can only be used with boolean settings, i.e. settings that can be enabled/disabled.
        * This function is called within the function that is used to change a setting.
        * @param value
        * @param specific_setting
        * @return returns a Max8150::ReturnCode.
        */
        Max86150::ReturnCode writeBoolSettingToRegister(bool value, Max86150::Setting specific_setting);

        /**
         * Configure the read_buffer pointers inside the data_element structs to point to the correct addresses
         * in the fifo_buffer variable.
         *
         * @return Max86150::ReturnCode::OK when successfull.
         */
        Max86150::ReturnCode setDataElementPointers(void);


        /**
         * Read the status of the Max86150 reset routine and update the class variable accordingly.
         *
         * @return true when still resetting, false if reset is finished.
         */
        bool isResetting();

    public:
        /**
         * Construct the driver.
         *
         * The I2C object is initialized, the SDA and SCL pins are allready defined for the target.
         * On the schematic from mbed-os site the different I2C pins indicate where extra slaves
         * can be connected.
         * The interrupt is also initialized, and the callback function is coupled to it.
         *
         * @param int_pin The pin connected to the interrupt signal.
         * @param interruptCallback pointer to the callback function of the interrupt.
         * @param scl_rate Set the clockrate for the I2C interface in kHz.
         * @param logger Pointer to AFLogger::logger object.
         * @param queue Pointer to the eventqueue.
         */
        Max86150(PinName int_pin, void (*interruptCallback)(void), uint16_t scl_rate, AFLogger::Logger *logger, EventQueue *queue);

```

```
            /**
             *  Reads the content of register's 1 and 2 and returns the cause
↪  of the interrupt.
             *  @return Max86150::InterruptStatus indicating the interrupt
↪  cause
             */
            Max86150::InterruptStatus getInterruptStatus();

            /**
            * Enable/disable interrupt setting.
            *
            * The following interrupt settings can be enabled/disabled:
            * A_FULL_EN;
            * PPG_RDY_EN;
            * ALC_OVF_EN;
            * PROX_INT_EN;
            * VDD_OOR_EN;
            * ECG_RDY_EN.
            * For elaboration on the interrupt settings, refer to enumeration
↪  InterruptEnableConf
            * @param enable true to enable this interrupt, false to disable
↪  this interrupt.
            * @param specific_interrupt The interrupt of interest to be
↪  enabled/disable.
            * @return returns a Max86150::ReturnCode.
            */
            Max86150::ReturnCode setInterrupt(bool enable,
             ↪  Max86150::InterruptEnableConf specific_interrupt);

            /**
             * Read the overflow count out of the register.
             *
             * When FIFO is full, any new samples result in new or old samples
↪  getting lost depending on FIFO_ROLLS_ON_FULL.
             * OVF_COUNTER counts the number of samples lost. It saturates at
↪  0x1F.
             *
             * @return Number of samples lost.
             */
            uint8_t getFifoOverflowCount();

            /**
             * Get the Fifo read pointer from the Max86150 register.
             *
             * @return Fifo address pointed to by read pointer.
             */
            uint8_t getFifoReadPointer();

            /**
```

```
         * @param address value between 0 - 31.
         */
        Max86150::ReturnCode setFifoReadPointer(uint8_t address);

        /**
         * Read all the samples that are available from the Fifo buffer.
         *
         * @return returns a Max86150::ReturnCode::OK on success, otherwise other value.
         */
        Max86150::ReturnCode readFifo();

        /**
         * 0: A_FULL interrupt does not get cleared by FIFO_DATA register read. It gets cleared by
         *       status register read.
         * 1: A_FULL interrupt gets cleared by FIFO_DATA register read or status register read
         *
         * @param value true for 1, false for 0.
         */
        Max86150::ReturnCode setFifoAlmostFullInterruptClear(bool value);

        /**
         * 0: The FIFO stops on full.
         * 1: The FIFO automatically rolls over on full, thereby overwriting previous data.
         *
         * @param value true for 1, false for 0.
         */
        Max86150::ReturnCode setFifoFullBehaviour(bool value);

        /**
         * 0: A_FULL interrupt gets asserted when the a_full condition is detected. It is cleared by status
         *       register read, but re-asserts for every sample if the a_full condition persists.
         * 1: A_FULL interrupt gets asserted only when the a_full condition is detected. The interrupt
         *       gets cleared on status register read, and does not re-assert for every sample until a new
         *       a-full condition is detected.
         *
         * @param value true for 1, false for 0.
         */
        Max86150::ReturnCode setFifoAlmostFullBehaviour(bool value);

        /**
         * Specifies the amount of free space on the FIFO at which the interrupt is triggered. Can
         * be between 0 - 15 samples.
         *
         * @param value integer between 0 - 15 indicating when the interrupt is triggered.
         */
        Max86150::ReturnCode setFifoAlmostFullValue(uint8_t value);

        /**
         * Configure the FIFO buffer data.
         *
         * The structure in which data is stored in the FIFO buffer is controlled by the
         * FIFO_DATA_CONTROL_REGISTER's. The FIFO buffer can hold samples from up to 4 different
         * sensors. These sensors are defined in Max86150::FifoDataConfig. The elements of the
         * FIFO buffer are defined in Max86150::FifoDataElement.
         * @param sensor The sensor which is assigned to the selected Fifo element.
         * @param data_register The element of the Fifo which to add the sensor to.
         * @return returns a Max86150::ReturnCode.
         *
         * @todo Set the fifo_element pointers to point to the correct position in the read buffer.
         */
        Max86150::ReturnCode setFifoDataElement(Max86150::FifoSensorsConf sensor, bool enable);

        /**
         * 0: Push to FIFO is disabled, but the read and write pointers and the data in the FIFO are all
         * held at their values before FIFO_EN is set to 0.
         * 1: The FIFO is enabled. When this bit is set the FIFO is flushed of all old data and the new
         * samples start loading from pointer zero
         *
         * @return OK on success, otherwise other ReturnCode
         */
        Max86150::ReturnCode setFifoEnable(bool enable);

        /**
         * 0: The part is in normal operation. No action taken
         * 1: The part can be put into a power-save mode by writing a '1' to this bit. While in this mode
         * all registers remain accessible and retain their data. ADC conversion data contained in the
         * registers are previous values. Writable registers also remain accessible in shutdown. All
         * interrupts are cleared. In this mode the oscillator is shutdown and the part draws minimum
         * current. If this bit is asserted during a active conversion then the conversion completes
         * before the part shuts down.
         *
         * @return OK on success, otherwise other ReturnCode
         */
        Max86150::ReturnCode sleep(bool enable);

        /**
         * 0: The part is in normal operation. No action taken.
         * 1: The part under-goes a forced power-on-reset sequence. All configuration, threshold and
         * data registers including distributed registers are reset to their power-on-state. This bit then
         * automatically becomes ‘0’ after the reset sequence is completed.
         *
         * @return OK on success, otherwise other ReturnCode
         */
        Max86150::ReturnCode reset();

        /**
         * Set the range of the PPG Analog to digital converter. Only fixed values are available
         * defined in Max86150::PpgAdcRangeConf.
         *
         * @return OK on success, otherwise other ReturnCode
         */
        Max86150::ReturnCode setPpgAdcRange(Max86150::PpgAdcRangeConf value);

        /**
         * Set the effectife sampling rate. With this the number of pulses per sample
         * can also be selected. The available configurations are stored in Max861150::PpgSampleRateConf.
         *
         * @return OK on success, otherwise other ReturnCode
         * @todo check if the selected samplerate is available (table from page 30 of datasheet).
         */
        Max86150::ReturnCode setPpgSampleRate(Max86150::PpgSampleRateConf value);

        /**
         * Set the pulse width of the ppg signal. The available configurations are stored in
         * Max86150::PpgPulseWidthConfg.
         *
         * @return OK on success, otherwise other ReturnCode
         */
        Max86150::ReturnCode setPpgPulseWidth(Max86150::PpgPulseWidthConf value);

        /**
         * To reduce the amount of data throughput, adjacent samples (in each individual channel) can be averaged and decimated
         * on the chip by setting this register.
         * These bits set the number of samples that are averaged on chip before being written to the FIFO.
         *
         * @return OK on success, otherwise other ReturnCode
         */
        Max86150::ReturnCode setPpgSampleAveraging(Max86150::SampleAveragingConf value);

        /**
         * Set the value of the IR ADC count at which the MAX86150 begins the PPG mode
         * specified in the FIFO data control register.
         *
         * PROX_INT_THRESH: Proximity Mode Interrupt Threshold*
         * This register sets the IR ADC count that triggers the beginning of the PPG mode specified in the FIFO Data Control
         * Register. The threshold is defined as the 8 MSB bits of the ADC count. For example, if PROX_INT_THRESH[7:0] = 0x01,
         * then an ADC value of 1023 (decimal) or higher triggers the PROX interrupt. If PROX_INT_THRESH[7:0] = 0xFF, then
         * only a saturated ADC triggers the interrupt.
         *
         * @param threshold the value which to write to the register
         * @return Max86150::ReturnCode::OK on succes.
         */
        Max86150::ReturnCode setProximityThreshold(uint8_t threshold);

        /**
         * Set the current pulse amplitude of the selected LED.
         *
         * This function set's both the LEDx PA register and the LED Range register so a current between
         * 0 and 102 mA can be selected. The LED PILOT PA register can also be set, however, when a
         * value exceeding the LEDx_RGE range is selected, the maximum value will be set for the PILOT
         * allowed by the current value of LEDx_RGE.
         *
```

```cpp
         * @param amplitude a value between 0 and 102 specifiy the pulse amplitude in mA.
         * @param led the Led of which to set the pulse amplitude.
         * @return Max86150::ReturnCode::OK on succes.
         */
        Max86150::ReturnCode setLedPulseAmplitude(uint8_t amplitude, Max86150::LedType led);

        /**
        * Set the ECG sample rate and the oversampling ratio, the sample rate is the product of the two.
        *
        * The eventual sample rate is the product of the base sample rate and the oversampling ratio.
        *
        * @param base_sample_rate the base sample rate to be used.
        * @param over_sampling_ratio the multiplication factor for the base sample rate.
        * @return Max86150::ReturnCode::OK on success.
        */
        Max86150::ReturnCode setEcgSampleRate(Max86150::EcgAdcSampleRateConf base_sample_rate, Max86150::OverSamplingRatioConf over_sampling_ratio);

        /**
         * Set the gain of the ECG progammable gain amplifier and the ECG instrumentation
         * amplifier.
         *
         * The equivalent input voltage is given by V_INPUT = ADC_CODE x 12.247V / IA_GAIN / PGA_GAIN.
         * only the setting PGA_GAIN=8, IA_GAIN=9.5 is trimmed to tight tolerence.
         * Note that the values for pga_gain and ia_gain must be specified explicitly everytime the function is called.

         * @param pga_gain The gain value to set the Programmable gain amplifier to.
         * @param ia_gain The gain value to set the Instrumentation amplifier to.
         * @return Max86150::ReturnCode::OK on success.
         */
        Max86150::ReturnCode setAmplifierGains(Max86150::PgaGainConf pga_gain, Max86150::InstrAmpGainConf ia_gain);

        /**
         * Get the new values from the specified sensor.
         *
```

```
             * @param sensor The sensor for which data is to be gathered.
             * @param write_buffer pointer to first location in array which to
 ↪  write the data to.
             * @return OK if succesfull, otherwise corresponding error
             */
            Max86150::ReturnCode getSensorData(FifoSensorsConf sensor, int32_t
             ↪  *write_buffer);

            uint8_t getPartId();
    };
}

#endif
```

[breaklines]

A.3. Max86150.cpp

```cpp
#include "max86150.h"
#include <cstdint>
#include <cstdio>

using namespace driver;

//============================PRIVATE_FUNCTIONS=================================
Max86150::ReturnCode Max86150::readRegister(Max86150::RegisterAddresses
 ↪  address, char* read_buffer, uint8_t repeat){
    if(resetting){
        if(isResetting()){
            return RESETTING;
        }
    }

    queue->call(printf, "This test for eventqueue");

    char data = address;
    uint8_t flgs[2];
    uint8_t i = 0;

    do {
        flgs[0] = i2c.write(SLAVE_ADDRESS, &data, 1, true);
        flgs[1] = i2c.read((SLAVE_ADDRESS|1), read_buffer, repeat, false);
        i++;

        // If both i2c transmissions where succesfull the loop exits
        if(((flgs[0] == 0) && (flgs[1] == 0)) ){
            break;
        }
    }while (i < I2C_RETRY_LIMIT);

    if(i >= I2C_RETRY_LIMIT){
        return ReturnCode::I2C_NO_ACK;
    }

    return Max86150::ReturnCode::OK;
}

Max86150::ReturnCode Max86150::writeRegister(Max86150::RegisterAddresses
 ↪  address, uint8_t data){

    if(resetting){
        if(isResetting()){
            return RESETTING;
        }
    }

    char data2[2] = {address, data};
    uint8_t flg;
    uint8_t i=0;
    do {
        flg = i2c.write(SLAVE_ADDRESS, data2, 2, true);
        i++;

        if(flg==0){
            break;
        }
    }while (i < I2C_RETRY_LIMIT);

    if(i >= I2C_RETRY_LIMIT){
        return ReturnCode::I2C_NO_ACK;
    }

    return ReturnCode::OK;
}

uint8_t Max86150::makeWriteRegisterData(bool enable, uint8_t bitmask, uint8_t current_data){
    uint8_t data;
    if(enable==true){
        data = (current_data | bitmask);
    }else{
        data = ~(~current_data | bitmask);
    }
    return data;
}

Max86150::ReturnCode Max86150::ackWriteRegister(Max86150::Setting specific_setting, uint8_t data){
    if(Max86150::writeRegister(specific_setting.address, data) == Max86150::ReturnCode::OK){
        Max86150::registerContents[specific_setting.address] = data;
        return Max86150::ReturnCode::OK;
    }

    logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to write    %x to register    %x", specific_setting.address, data);
    return Max86150::ReturnCode::I2C_NO_ACK;
}

Max86150::ReturnCode Max86150::setDataElementPointers(void){
    // Each sensor sample takes up 3 bytes in memory.
    // The first sensor will have its first sample in   fifo_buffer[0] - fifo_buffer[2]
    // The second sensor in                             fifo_buffer[3] - fifo_buffer[5]
    // ------------------------------------ETC---------------------------------------
    // The first sensor will have its second sample in fifo_buffer[0+active_data_elements*3] - fifo_buffer[2+active_data_elements*3]

    // Buffer can hold 32 samples for each sensor, so 32 pointers need to be constructed for each sensor.
    uint16_t index;          // must be 16 bit integer, otherwise an overflow will occur.
    // i is Sample number
    for(uint8_t i=0; i < FIFO_BUFFER_SIZE; i++){
        // n is Sensor number
        for(uint8_t n = 0; n < active_data_elements; n++){
            // The data_elements array holds pointers to the individual data elements.
            // pointers to structs can access the members using ->
            index = ELEMENT_SAMPLE_SIZE*n + ELEMENT_SAMPLE_SIZE*active_data_elements*i;
            data_elements[n]->read_buffer[i] = &fifo_buffer[index];
        }
    }
    return Max86150::OK;
}

Max86150::ReturnCode Max86150::writeBoolSettingToRegister(bool value, Max86150::Setting specific_setting){
    uint8_t current_data = Max86150::registerContents[specific_setting.address];
    uint8_t data = Max86150::makeWriteRegisterData(value, specific_setting.bitmask, current_data);
    return ackWriteRegister(specific_setting, data);
}

bool Max86150::isResetting(){
    // Send write slave id
    i2c.start();
    i2c.write(Max86150::SLAVE_ADDRESS);
    // Send register address
    i2c.write(settings._RESET.address);
    // Repeated start
    i2c.start();
    // Send read slave id (SLAVE_ADDRES bitwise_or 1)
    i2c.write(Max86150::SLAVE_ADDRESS|1);

    uint8_t data = i2c.read(false);
    data = data & settings._RESET.bitmask;
    if(data == 0){
        resetting = false;
```

```cpp
        return false;
    }else{
        return true;
    }

}

//===============================PUBLIC_FUNCTIONS=========================================
Max86150::Max86150(PinName int_pin, void (*interruptCallback)(void), uint16_t scl_rate, AFLogger::Logger *logger, EventQueue *queue):
    i2c(I2C_SDA, I2C_SCL),
    intb(int_pin),
    vdd_dig(PA_5),
    vdd_ana(PA_6),
    vdd_led(PB_5)
{
    logger->log(AFLogger::Logger::eLogLevel::NORMAL, "Initializing max86150");
    i2c.frequency(1000*scl_rate);
    logger->log(AFLogger::Logger::eLogLevel::TRACE, "I2C frequency set to %u kHz", scl_rate);
    //max86150Interrupt.fall(*interruptCallback);
    //logger->log(AFLogger::Logger::eLogLevel::TRACE, "Interrupt callback configured");
    this->logger = logger;
    this->queue = queue;

     // Start from a known state
    vdd_led = 0;
    vdd_dig = 0;
    vdd_ana = 0;

    ThisThread::sleep_for(1s);

    // Initialize power to Max chip
    vdd_ana = 1;
    logger->log(AFLogger::Logger::eLogLevel::NORMAL, "Turned on vdd_ana");
    ThisThread::sleep_for(500ms);
    vdd_dig = 1;
    logger->log(AFLogger::Logger::eLogLevel::NORMAL, "Turned on vdd_dig");
    ThisThread::sleep_for(500ms);
    vdd_led = 1;
    logger->log(AFLogger::Logger::eLogLevel::NORMAL, "Turned on vdd_led");

    while(intb.read()==1){
        logger->log(AFLogger::Logger::eLogLevel::NORMAL, "No interrupt received yet");
        ThisThread::sleep_for(1s);
    }

    uint8_t flgs[2];
    char data[3];
    char buff;
    data[0] = 0x00; // interrupt cause register
    data[1] = 0xff; // ID register

    // THIS IS THE HARDCODE WAY USING HIGHER LEVEL FUNCTIONS FROM I2C CLASS
    flgs[0] = i2c.write(SLAVE_ADDRESS, data, 1, true);
    flgs[1] = i2c.read(SLAVE_ADDRESS, &buff, 1);

    if(buff==1){
        logger->log(AFLogger::Logger::eLogLevel::NORMAL, "PWR_RDY interrupt received");
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "PWR_RDY interrupt not received, please check connections");
    }

    flgs[0] = i2c.write(SLAVE_ADDRESS, &data[1], 1, true);
    flgs[1] = i2c.read(SLAVE_ADDRESS, &buff, 1);

    if(buff==0x1E){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Max86150 succesfully initialized");
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Max86150 not correctly initialized");
    }
}

// Alex
Max86150::InterruptStatus Max86150::getInterruptStatus(){
    char data;
    InterruptStatus ret = NONE;
    // Read the first interrupt status register, this must be ISR safe
    if(readRegister(RegisterAddresses::INTERRUPT_STATUS_1, &data)==I2C_NO_ACK){
        //logger->log(AFLogger::Logger::eLogLevel::ERROR, "Could not read INTERRUPT_STATUS_1 register");
        return NONE;
    }
    if(data!=0){
        // Cast the value in data[0] to the type Max86150::InterruptStatus
        ret = static_cast<InterruptStatus>(data);
        return ret;
    }

    // Read the second interrupt status register
    if(readRegister(RegisterAddresses::INTERRUPT_STATUS_2, &data)==I2C_NO_ACK){
        //logger->log(AFLogger::Logger::eLogLevel::ERROR, "Could not read INTERRUPT_STATUS_2 register");
        return NONE;
    }
    if(data!=0){
        // Add 1 to indicate it is the second status register, cast to
         ↪ Max86150::InterruptStatus
        ret = static_cast<InterruptStatus>(data|1);
        //logger->log(AFLogger::Logger::eLogLevel::TRACE, "Interrupt cause is:
         ↪ %x", ret);
        return ret;
    }

    //logger->log(AFLogger::Logger::eLogLevel::WARNING, "No interrupt cause was
     ↪ found");
    // If all fails return NONE so the error can be handled
    return ret;
}

Max86150::ReturnCode Max86150::setInterrupt(bool enable,
 ↪ Max86150::InterruptEnableConf specific_interrupt){
    Max86150::Setting specific_setting;
    switch (specific_interrupt) {
    case A_FULL_EN:
        specific_setting = settings._A_FULL_EN;
        break;
    case PPG_RDY_EN:
        specific_setting = settings._PPG_RDY_EN;
        break;
    case ALC_OVF_EN:
        specific_setting = settings._ALC_OVF_EN;
        break;
    case PROX_INT_EN:
        specific_setting = settings._PROX_INT_EN;
        break;
    case VDD_OOR_EN:
        specific_setting = settings._VDD_OOR_EN;
        break;
    case ECG_RDY_EN:
        specific_setting = settings._ECG_RDY_EN;
        break;
    }

    uint8_t current_data =
     ↪ Max86150::registerContents[specific_setting.address];
    uint8_t data = Max86150::makeWriteRegisterData(enable,
     ↪ specific_setting.bitmask, current_data);
    if(ackWriteRegister(specific_setting, data)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully enabled
         ↪ interrupt %x", specific_setting.bitmask);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to enable interrupt %x", specific_setting.bitmask);
        return I2C_NO_ACK;
    }
}

uint8_t Max86150::getFifoOverflowCount(){
    char data;
    Max86150::readRegister(Max86150::RegisterAddresses::OVERFLOW_COUNTER, &data);
    // logger->log(AFLogger::Logger::eLogLevel::TRACE, "Fifo overflow count is: \t %u", data);
    return (uint8_t)data;
}

// Amar
uint8_t Max86150::getFifoReadPointer(){
    char data;
    Max86150::readRegister(Max86150::RegisterAddresses::FIFO_READ_POINTER, &data);
    // logger->log(AFLogger::Logger::eLogLevel::TRACE, "Fifo read pointer is: \t %u", data);
    return (uint8_t)data;
}

// Amar
Max86150::ReturnCode Max86150::setFifoReadPointer(uint8_t address){
    // address must be within the range of 0-31;
    if(address < 0 || address > 31 ){
        return ReturnCode::VALUE_OOR;
    }
    Max86150::Setting specific_setting = settings._FIFO_RD_PTR;
    if(ackWriteRegister(specific_setting, address)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set fifo read pointer to %x", address);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set fifo read pointer to %x", address);
        return I2C_NO_ACK;
    }
}

// Alex
Max86150::ReturnCode Max86150::readFifo(){
    // THIS FUNCTION IS AN INTERRUPT SERVICE ROUTINE FUNCTION, SO NO PRINTF

    char write_buffer[2];
    char status[3];
    char* write_ptr = &status[0];
    char* ovf_count = &status[1];;
    char* read_ptr = &status[2];;       // value of read_ptr is the address, the value of the register is *read_ptr
    uint8_t samples_in_fifo;



    // send start
    // send device address + write mode
    // send fifo_wr_ptr register address
    write_buffer[0] = RegisterAddresses::FIFO_WRITE_POINTER;
    if(i2c.write(SLAVE_ADDRESS, write_buffer, 1, true)){
        return ReturnCode::I2C_NO_ACK;
    }

    // repeated start
    // send device address + read mode
    // read fifo_wr_ptr
    // read ovf_acounter
    // read fifo_rd_ptr
    // stop
    i2c.read(SLAVE_ADDRESS, status, 3, false);

    // Check if there is an overflow
    if ((uint8_t)*ovf_count!=0) {
        samples_in_fifo = FIFO_BUFFER_SIZE;
    }else{
        if((uint8_t)*write_ptr <= (uint8_t)*read_ptr){
            // The write pointer has looped around
            samples_in_fifo = FIFO_BUFFER_SIZE - ((uint8_t)*read_ptr-(uint8_t)*write_ptr);
        }else {
            samples_in_fifo = ((uint8_t)*write_ptr-(uint8_t)*read_ptr);
        }
    }
    //logger->log(AFLogger::Logger::eLogLevel::TRACE, "Samples to read in Fifo: %u", samples_in_fifo);

    // start
    // send device address + write mode
    // send fifo_data register address
    write_buffer[0] = RegisterAddresses::FIFO_DATA_REGISTER;

    if(i2c.write(SLAVE_ADDRESS, write_buffer, 1, true)!=0){
        return ReturnCode::I2C_NO_ACK;
    }
    // repeated start
    // send device address + read mode
    // loop through the samples in fifo
    if(i2c.read(SLAVE_ADDRESS, fifo_buffer,
     ↪  (samples_in_fifo*active_data_elements*ELEMENT_SAMPLE_SIZE), false)!=0){
        // Reset read pointer to original position so read can be retried.
        if(setFifoReadPointer((uint8_t)*read_ptr)!=ReturnCode::OK){
            return ReturnCode::I2C_NO_ACK;
        }
    }

    return ReturnCode::OK;
}

// Amar
Max86150::ReturnCode Max86150::setFifoAlmostFullInterruptClear(bool value){
    Max86150::Setting specific_setting = settings._A_FULL_CLR;
    if(writeBoolSettingToRegister(value, specific_setting)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
         ↪  FIFO_ALMOST_FULL_INTTERUPT_CLEAR to %x", value);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set
         ↪  FIFO_ALMOST_FULL_INTTERUPT_CLEAR to %x", value);
        return I2C_NO_ACK;
    }
}

// Amar
Max86150::ReturnCode Max86150::setFifoAlmostFullBehaviour(bool value){
    Max86150::Setting specific_setting = settings._A_FULL_TYPE;
    if(writeBoolSettingToRegister(value, specific_setting)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
         ↪  FIFO_ALMOST_FULL_BEHAVIOUR to %x", value);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set
         ↪  FIFO_ALMOST_FULL_BEHAVIOUR to %x", value);
        return I2C_NO_ACK;
    }
}

// Amar
Max86150::ReturnCode Max86150::setFifoFullBehaviour(bool value){
    Max86150::Setting specific_setting = settings._FIFO_ROLLS_ON_FULL;
    if(writeBoolSettingToRegister(value, specific_setting)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
         ↪  FIFO_FULL_BEHAVIOUR to %x", value);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set
         ↪  FIFO_FULL_BEHAVIOUR to %x", value);
        return I2C_NO_ACK;
    }
}

Max86150::ReturnCode Max86150::setFifoAlmostFullValue(uint8_t value){
    if(value < 16){
        uint8_t current_settings = registerContents[FIFO_CONFIGURATION];
        // ABCD 1001
        current_settings = current_settings >> 4;
        // xxxx ABCD
        current_settings = current_settings << 4;
        // ABCD 0000
        uint8_t new_settings = current_settings | value;
        if(writeRegister(FIFO_CONFIGURATION, new_settings)==ReturnCode::OK){
            registerContents[FIFO_CONFIGURATION] = new_settings;
            logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
             ↪  FIFO_ALMOST_FULL_VALUE to %u", value);
            return ReturnCode::OK;
        }else {
            logger->log(AFLogger::Logger::eLogLevel::TRACE, "Failed to set
             ↪  FIFO_ALMOST_FULL_VALUE to %u", value);
            return ReturnCode::I2C_NO_ACK;
        }
    }else{
        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Invalid argument
         ↪  [%u]. value must be smaller than 16", value);
        return Max86150::ReturnCode::INVALID_DATA;
    }
}

// Alex
Max86150::ReturnCode Max86150::setFifoDataElement(Max86150::FifoSensorsConf
 ↪  sensor, bool enable){
    // Check if there is still space for an axtra sensor
    if(active_data_elements == FIFO_MAX_DATA_ELEMENTS){
        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Maximum number of
         ↪  FIFO_DATA_ELEMENTS allready reached");
        return ReturnCode::MAX_SENSORS_EXCEEDED;
    }

    // Check if the setting is allready correct, because then nothing has to be
     ↪  done.
    if (sensorStatus[sensor] == enable) {
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "This FIFO_DATA_ELEMENT
         ↪  is allready enabled");
        return ReturnCode::OK;
    }
```

```cpp

    // Set the sensorStatus map
    sensorStatus[sensor] = enable;

    // Reset the fd sensor values
    logger->log(AFLogger::Logger::eLogLevel::TRACE, "Resetting all data elements for re-designation");
    for (uint8_t i=0; i < FIFO_MAX_DATA_ELEMENTS; i++) {
        data_elements[i]->sensor = FifoSensorsConf::SENSOR_NONE;
    }

    // Next the data elements need to be assigned the correct sensor.
    // The ppg's come first and the ecg comes last, so construct a list with the correct sequence.
    // sensors
    uint8_t i = 0;
    if(sensorStatus[LED1_IR]){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "LED1_IR is set to data_element[%u]", i);
        data_elements[i]->sensor = FifoSensorsConf::LED1_IR;
        i++;
    }
    if(sensorStatus[LED2_RED]){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "LED2_RED is set to data_element[%u]", i);
        data_elements[i]->sensor = FifoSensorsConf::LED2_RED;
        i++;
    }
    if(sensorStatus[PILOT_LED1_IR]){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "PILOT_LED1_IR is set to data_element[%u]", i);
        data_elements[i]->sensor = FifoSensorsConf::PILOT_LED1_IR;
        i++;
    }
    if(sensorStatus[PILOT_LED2_RED]){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "PILOT_LED1_IR is set to data_element[%u]", i);
        data_elements[i]->sensor = FifoSensorsConf::PILOT_LED2_RED;
        i++;
    }
    if(sensorStatus[ECG]){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "ECG is set to data_element[%u]", i);
        data_elements[i]->sensor = FifoSensorsConf::ECG;
        i++;
    }

    // Keep track of the number of active sensors
    if(enable){
        active_data_elements++;
    }else {
        active_data_elements--;
    }

    // Write the correct values to the registers on the Max86150
    // Complete 8 bit value is constructed by shifting fd2 or fd4 4 bits to the
    ↪   left, and
    // or this with fd1 or fd3 respsectively.

    ↪   if(writeRegister(RegisterAddresses::FIFO_DATA_CONTROL_REGISTER_1,(fd2.sensor<<4)|(fd1
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully writen
        ↪   data to FIFO_DATA_ELEMENT_REGISTER_1");
    }else{
        logger->log(AFLogger::Logger::eLogLevel::WARNING, "failed to write data
        ↪   to FIFO_DATA_ELEMENT_REGISTER_1");
        return I2C_NO_ACK;
    }

    ↪   if(writeRegister(RegisterAddresses::FIFO_DATA_CONTROL_REGISTER_1,(fd4.sensor<<4)|(fd3
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully writen
        ↪   data to FIFO_DATA_ELEMENT_REGISTER_2");
    }else{
        logger->log(AFLogger::Logger::eLogLevel::WARNING, "failed to write data
        ↪   to FIFO_DATA_ELEMENT_REGISTER_2");
        return I2C_NO_ACK;
    }

    // Reconfigure the pointers to the read buffers inside the FifoDataElement
    ↪   structs
    setDataElementPointers();

    return ReturnCode::OK;
}

// Amar
Max86150::ReturnCode Max86150::setFifoEnable(bool enable){
    Max86150::Setting specific_setting = settings._FIFO_EN;
    if(writeBoolSettingToRegister(enable, specific_setting)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
        ↪   FIFO_ENABLE to %x", enable);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set
        ↪   FIFO_ENABLE to %x", enable);
        return I2C_NO_ACK;
    }
}
```

```
// Amar
Max86150::ReturnCode Max86150::sleep(bool enable){
    Max86150::Setting specific_setting = settings._SHDN;
    if(writeBoolSettingToRegister(enable, specific_setting)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set SLEEP
         ↪  to %x", enable);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set SLEEP to
         ↪  %x", enable);
        return I2C_NO_ACK;
    }
}

// Amar
Max86150::ReturnCode Max86150::reset(){
    if (resetting) {
        return RESETTING;
    }
    Max86150::Setting specific_setting = settings._RESET;
    if(writeBoolSettingToRegister(true, specific_setting)==OK){
        resetting = true;
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully RESET");
        return ReturnCode::OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to RESET");
        return I2C_NO_ACK;
    }
}

// Alex
Max86150::ReturnCode Max86150::setPpgAdcRange(Max86150::PpgAdcRangeConf value){
    // specific setting
    Setting ss = settings._PPG_ADC_RGE;
    uint8_t data = registerContents[ss.address];
    data = (~ss.bitmask) & data;
    data = (value << 6) | data;

    if(writeRegister(ss.address, data)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
         ↪  PPG_ADC_RANGE to %x", value);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set
         ↪  PPG_ADC_RANGE to %x", value);
        return I2C_NO_ACK;
    }
}

```

```cpp
// Alex
Max86150::ReturnCode Max86150::setPpgSampleRate(Max86150::PpgSampleRateConf value){
    Setting ss = settings._PPG_SR;
    uint8_t data = registerContents[ss.address];
    data = (~ss.bitmask) & data;
    data = (value << 4) | data;

    if(writeRegister(ss.address, data)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set PPG_SAMPLE_RATE to %x", value);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set PPG_SAMPLE_RATE to %x", value);
        return I2C_NO_ACK;
    }
}

// Alex
Max86150::ReturnCode Max86150::setPpgPulseWidth(Max86150::PpgPulseWidthConf value){
    Setting ss = settings._PPG_LED_PW;
    uint8_t data = registerContents[ss.address];
    data = (~ss.bitmask) & data;
    data = value | data;

    if(writeRegister(ss.address, data)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set PPG_PULSE_WIDTH to %x", value);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set PPG_PULASE_WIDTH to %x", value);
        return I2C_NO_ACK;
    }
}

// Amar
Max86150::ReturnCode Max86150::setPpgSampleAveraging(Max86150::SampleAveragingConf smp_ave){
    Max86150::Setting specific_setting = settings._SMP_AVE;
    uint8_t data = 0;
    data = data | smp_ave;

    logger->log(AFLogger::Logger::eLogLevel::TRACE, "Setting PPG sample averaging to %u", smp_ave);

    if(ackWriteRegister(specific_setting, data)){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
         ↪  PPG_SAMPLE_AVERAGING to %x", smp_ave);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set
         ↪  PPG_SAMPLE_AVERAGING to %x", smp_ave);
        return I2C_NO_ACK;
    }
}

// Amar
Max86150::ReturnCode Max86150::setProximityThreshold(uint8_t threshold){

     ↪  if(Max86150::writeRegister(Max86150::RegisterAddresses::PROX_INTERRUPT_THRESHOLD,
     ↪  threshold) == Max86150::ReturnCode::OK){
        Max86150::registerContents[PROX_INTERRUPT_THRESHOLD] = threshold;

        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
         ↪  PROXIMITY_THRESHOLD to %x", threshold);

        return Max86150::ReturnCode::OK;
    }
    logger->log(AFLogger::Logger::eLogLevel::ERROR, "Failed to set
     ↪  PROXIMITY_THRESHOLD to %x", threshold);
    return Max86150::ReturnCode::I2C_NO_ACK;
}

// Alex
Max86150::ReturnCode Max86150::setLedPulseAmplitude(uint8_t amplitude,
 ↪  Max86150::LedType led){
    /*
    the amplitude can take a value between 0 and 102 mA
    from 51 - 102 mA the rge register value must be set to '01'
    */

    // Check if the amplitude is within range
    if((amplitude < 0 || amplitude > 102)){
        logger->log(AFLogger::Logger::eLogLevel::WARNING, "%u mA is out of
         ↪  range. Pulse amplitude must be smaller than 102 mA", amplitude);
        return ReturnCode::VALUE_OOR;
    }

    uint8_t data;

    // Switch depending on the selected LED
    switch (led) {
        case INFRA_RED:{
            // Current settings in registers
            uint8_t range = (registerContents[settings._LED1_RGE.address] &&
             ↪  settings._LED1_RGE.bitmask);
```

```cpp

            if(amplitude>51){
                if(range == 0){
                    // First write new range value
                    if(ackWriteRegister(settings._LED1_RGE, 0b00000001)==ReturnCode::I2C_NO_ACK){
                        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set LED1_IR_RGE range to 100 mA");
                        return I2C_NO_ACK;
                    }
                    logger->log(AFLogger::Logger::eLogLevel::WARNING, "Succesfully set LED1_IR_RGE range to 100 mA");
                }
                // With the range set to 102 mA, the step size is 0.4 mA.
                // Cast the division of the value by 0.4 into uint8_t.
                data = (uint8_t) amplitude/0.4;
            }else{ // The value is smaller than 51
                if(range > 0){
                    // First write new range value so the value can be set more accurately
                    if(ackWriteRegister(settings._LED1_RGE, 0b00000000)==ReturnCode::I2C_NO_ACK){
                        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set LED1_IR_RGE range to 50 mA");
                      return I2C_NO_ACK;
                    }
                    logger->log(AFLogger::Logger::eLogLevel::WARNING, "Succesfully set LED1_IR_RGE range to 50 mA");
                }
                // With the range set to 51 mA the step size is 0.2 mA.
                // Cast the division of the value by 0.2 into uint8_t.
                data = (uint8_t) amplitude/0.2;
            }
            // Write new pulse amplitude value
            if(ackWriteRegister(settings._LED1_PA_CONF, data)==OK){
                logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set LED1_IR_PA range to %u mA", amplitude);
                return OK;
            }else{
                logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set LED1_IR_PA range to %u mA", amplitude);
                return I2C_NO_ACK;
            }
            break;
        }
        case RED:{
            // Current settings in registers
            uint8_t range = (registerContents[settings._LED2_RGE.address] && settings._LED2_RGE.bitmask);

            if(amplitude>51){
                if(range == 0){
                    // First write new range value
                    if(ackWriteRegister(settings._LED2_RGE, 0b00000100)==ReturnCode::I2C_NO_ACK){
                        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set LED2_IR_RGE range to 100 mA");
                        return I2C_NO_ACK;
                    }
                    logger->log(AFLogger::Logger::eLogLevel::WARNING, "Succesfully set LED1_IR_RGE range to 100 mA");
                }
                // With the range set to 102 mA, the step size is 0.4 mA.
                // Cast the division of the value by 0.4 into uint8_t.
                data = (uint8_t) amplitude/0.4;
            }else{ // The value is smaller than 51
                if(range > 0){
                    // First write new range value so the value can be set more accurately
                    if(ackWriteRegister(settings._LED2_RGE, 0b00000000)==ReturnCode::I2C_NO_ACK){
                        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set LED2_IR_RGE range to 50 mA");
                        return I2C_NO_ACK;
                    }
                    logger->log(AFLogger::Logger::eLogLevel::WARNING, "Succesfully set LED2_IR_RGE range to 50 mA");
                }
                // With the range set to 51 mA the step size is 0.2 mA.
                // Cast the division of the value by 0.2 into uint8_t.
                data = (uint8_t) amplitude/0.2;
            }
            // Write new pulse amplitude value
            if(ackWriteRegister(settings._LED2_PA_CONF, data)==OK){
                logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set LED2_IR_PA range to %u mA", amplitude);
                return OK;
            }else{
                logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set LED1_IR_PA range to %u mA", amplitude);
                return I2C_NO_ACK;
            }
            break;
        }
        case PILOT:{
            // The pilot pulse amplitude depends on which LED is used for the pilot
            // The step size also depends on the RGE setting of that specific LED
            // Fine control is not needed over the pilot pulse amplitude, so a RGE of 102 mA is assumed
            data = (uint8_t) amplitude/0.4;

            if(ackWriteRegister(settings._PILOT_PA, data)==OK){
                logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set PILOT_PA range to %u mA", amplitude);
                return OK;
            }else{
                logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set PILOT_PA range to %u mA", amplitude);
                return I2C_NO_ACK;
            }
            break;
        }
    }
}

// Alex
Max86150::ReturnCode Max86150::setEcgSampleRate(Max86150::EcgAdcSampleRateConf base_sample_rate, Max86150::OverSamplingRatioConf over_sampling_ratio){
    if(ackWriteRegister(settings._ECG_ADC_CLK, (base_sample_rate<<2))==I2C_NO_ACK){
        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set ECG_ADC_CLK to %x", base_sample_rate);
        return I2C_NO_ACK;
    }
    logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set ECG_ADC_CLK to %x", base_sample_rate);

    if(ackWriteRegister(settings._ECG_ADC_OSR, over_sampling_ratio)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set ECG_ADC_OSR to %x", over_sampling_ratio);
        return OK;
    }else{
        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set ECG_ADC_OSR to %x", over_sampling_ratio);
        return I2C_NO_ACK;
    }
}

// Amar
Max86150::ReturnCode Max86150::setAmplifierGains(Max86150::PgaGainConf pga_gain, Max86150::InstrAmpGainConf ia_gain){
    // Dummy example:
    // data:                    0000 0000   the 4 MSB are don't cares (check datasheet) and this initialization assumes that the pga_gain and ia_gain are explicitly specified each time the function is called
    // Bitwise shift right 4:   xxxx ABCD
    // Bitwise shift left 2:    xxAB CD00
    // Set pga_gain:            xxAB CDPP
    // Bitwise shift left 2:    ABCD PP00
    // Set ia_gain:             ACBD PPII
    Max86150::Setting specific_setting = settings._PGA_ECG_GAIN;  // here,
     ↪ specific_setting is only used for the register address when writing to
     ↪ the sensor, so either PGA_ECG_GAIN or IA_GAIN could be used
    uint8_t data = 0;
    data = data | pga_gain;
    data = data << 2;
    data = data | ia_gain;
    if(ackWriteRegister(specific_setting, data)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully set
         ↪ PA_GAIN to %x and IA_GAIN to %x", pga_gain, ia_gain);
        return OK;
    }else {
        logger->log(AFLogger::Logger::eLogLevel::WARNING, "Failed to set
         ↪ PA_GAIN to %x and IA_GAIN to %x", pga_gain, ia_gain);
        return I2C_NO_ACK;
    }
}

Max86150::ReturnCode Max86150::getSensorData(FifoSensorsConf sensor, int32_t
 ↪ *write_buffer){

    uint8_t setting = (registerContents[settings._PPG_ADC_RGE.address] &
     ↪ settings._PPG_ADC_RGE.bitmask);
    PpgAdcRangeConf range = (PpgAdcRangeConf) (setting >> 6);

    // The value to multiply by
    float mult;
    // The offset is used in the case of ECG, as the ADC outputs a bipolar
     ↪ integer.
    uint32_t offset = 0;
    // Cannot be unsigned, as ECG data can have a negative value.
    int32_t sample;
    uint32_t bitmask = 0;

    if (sensor != ECG) {
        // bitmask for PPG data
        bitmask = 0x07FFFF;
        // For the PPG the multiplier is determined by the ADC_RGE setting
         ↪ only
        switch (range) {
        case NANO_AMPERE_8192:
            mult = 7.8125;
            break;
        case NANO_AMPERE_4096:
            mult = 15.625;
            break;
        case NANO_AMPERE_16384:
            mult = 31.25;
            break;
        case NANO_AMPERE_32768:
            mult = 62.5;
            break;
        }
    }else{
        // bitmask for the ECG data
        bitmask = 0x03FFFF;
        offset = 65536;
        uint8_t ia_gain_bit = registerContents[settings._IA_GAIN.address] &
         ↪  settings._IA_GAIN.bitmask;
        float ia_gain;
        switch (ia_gain_bit) {
        case IA_GAIN_5:
            ia_gain = 5;
            break;
        case IA_GAIN_9_5:
            ia_gain = 9.5;
            break;
        case IA_GAIN_20:
            ia_gain = 20;
            break;
        case IA_GAIN_50:
            ia_gain = 50;
            break;
        default:
            ia_gain = 1;
        }

        uint8_t pga_gain_bit =
         ↪  (registerContents[settings._PGA_ECG_GAIN.address] &
         ↪  settings._PGA_ECG_GAIN.bitmask)>> 2;
        uint8_t pga_gain;
        switch (pga_gain_bit) {
        case PGA_GAIN_1:
            pga_gain = 1;
            break;
        case PGA_GAIN_2:
            pga_gain = 2;
            break;
        case PGA_GAIN_4:
            pga_gain = 4;
            break;
        case PGA_GAIN_8:
            pga_gain = 8;
            break;
        }

        // According to formula on page 17 of datasheet
        mult = 12.247 / (ia_gain*pga_gain);
    }

    // Check whether the sensor is enabled
    for (uint8_t i=0; i < active_data_elements; i++) {
        if(data_elements[i]->sensor == sensor){
            // Sensor is enabled, so process the fifo_buffer and write it to
             ↪  the array pointed to by write_buffer
            for (uint8_t j=0; j < data_elements[i]->new_samples; j++) {
                // Pointer to first byte of sample in the fifo_buffer
                char* base_pointer = data_elements[i]->read_buffer[j];
                sample =
                 ↪  (*(base_pointer)<<16)|(*(base_pointer+1)<<8)|*(base_pointer+2);
                sample = (sample & bitmask) - offset;
                // Store value to correct relative address of write_buffer
                *(write_buffer + j) = (sample)*mult;
            }
            return OK;
        }
    }

    return SENSOR_DISABLED;
}

uint8_t Max86150::getPartId(){
    char* data;
    InterruptStatus ret = NONE;
    // Read the first interrupt status register
    if(readRegister(RegisterAddresses::PART_ID, data)==OK){
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Succesfully read
         ↪  PART_ID as %x", *data);
        return (uint8_t)data[0];
    }else{
        logger->log(AFLogger::Logger::eLogLevel::TRACE, "Failed to read
         ↪  PART_ID");
        return 0;
    }
}
```

A.4. Max86150_test.cpp

```cpp
#include "max86150.h"
#include "logger.h"
#include "config.h"
#include <cstdint>

#if ACTIVE_FILE == DRIVER_TEST

void dummyCallbackFall(){
}

void dummyCallbackRise(){
}

void testall(){
    AFLogger::Logger logger(AFLogger::Logger::eLogLevel::TRACE);
    EventQueue queue;
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Started up everything===========================================================");
    driver::Max86150 test(PC_6, dummyCallbackFall, 100, &logger, &queue);

    // Calling every function one by one for testing
    test.getInterruptStatus();

    test.setInterrupt(true, driver::Max86150::InterruptEnableConf::A_FULL_EN);
    //test.setInterrupt(true, driver::Max86150::InterruptEnableConf::ALC_OVF_EN);
    //test.setInterrupt(true, driver::Max86150::InterruptEnableConf::ECG_RDY_EN);
    //test.setInterrupt(true, driver::Max86150::InterruptEnableConf::PPG_RDY_EN);
    //test.setInterrupt(true, driver::Max86150::InterruptEnableConf::PROX_INT_EN);
    //test.setInterrupt(true, driver::Max86150::InterruptEnableConf::VDD_OOR_EN);

    test.getFifoOverflowCount();
    test.getFifoReadPointer();
    test.setFifoReadPointer(14); // should work
    test.setFifoReadPointer(32); // address out of range, should return error
    // Might have to check whether sensors are activated
    test.setFifoAlmostFullInterruptClear(true); // simple bit write operation
    test.setFifoAlmostFullBehaviour(true);
    test.setFifoAlmostFullBehaviour(true);
    test.setFifoAlmostFullValue(12); // should work
    test.setFifoAlmostFullValue(17); // value out of range, should return error

    test.setFifoEnable(true); // should work
    test.sleep(true); // goes to sleep
    // A check should be implemented, as certain functions won't be accessible due to Max86150 sleeping, this still has to be implemented
    test.sleep(false);
    test.setPpgAdcRange(driver::Max86150::PpgAdcRangeConf::NANO_AMPERE_32768); // should work
    test.setPpgSampleRate(driver::Max86150::PpgSampleRateConf::SAMPLES_1600_PPS_1); // should work
    test.setPpgPulseWidth(driver::Max86150::PpgPulseWidthConf::MICRO_200); // should work
    test.setPpgSampleAveraging(driver::Max86150::SampleAveragingConf::SAMPLE_AVERAGE_2); // should work
    test.setProximityThreshold(144); // should work
    // Extensive testing
    test.setLedPulseAmplitude(10, driver::Max86150::LedType::INFRA_RED); // should work
    test.setLedPulseAmplitude(200, driver::Max86150::LedType::INFRA_RED); // value oor
    test.setLedPulseAmplitude(60, driver::Max86150::LedType::INFRA_RED); // changes range register
    test.setLedPulseAmplitude(10, driver::Max86150::LedType::RED); // should work
    test.setLedPulseAmplitude(200, driver::Max86150::LedType::RED); // value oor
    test.setLedPulseAmplitude(60, driver::Max86150::LedType::RED); // changes range register
    test.setLedPulseAmplitude(10, driver::Max86150::LedType::PILOT); // should work
    test.setLedPulseAmplitude(200, driver::Max86150::LedType::PILOT); // value oor
    test.setLedPulseAmplitude(60, driver::Max86150::LedType::PILOT); // does not change rge register, just divides by 0.4 and casts to int

    test.setEcgSampleRate(driver::Max86150::EcgAdcSampleRateConf::ECG_CLK_200, driver::Max86150::OverSamplingRatioConf::OSR_16);
    test.setAmplifierGains(driver::Max86150::PgaGainConf::PGA_GAIN_4, driver::Max86150::InstrumentationAmplifierGainConf::IA_GAIN_50);

    // Extensive test for the fifo data elements, most important part to get right.
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::LED1_IR, true); // should work
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::LED1_IR, true); // should return that the sensor is allready active
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::LED1_IR, false); // should work
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::LED2_RED, true); // should work
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::LED1_IR, true); // should work, and reorder the sensors properally
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::ECG, true); // should work, get's added to FD3
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::PILOT_LED1_IR, true);// should work, sensors get rearranged
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::PILOT_LED2_RED, true);//should not work, max sensors reached
    test.readFifo(); // Should try and read all available samples

    test.reset();
}

void testI2c(){
    InterruptIn intb(PC_6);
    DigitalOut led(LED1);
    DigitalOut vdd_dig(PA_5);
    DigitalOut vdd_ana(PA_6);
    DigitalOut vdd_led(PB_5);

    AFLogger::Logger logger(AFLogger::Logger::eLogLevel::TRACE);
    I2C i2c(I2C_SDA, I2C_SCL);
    i2c.frequency(100000);
    const uint8_t WRITE_ADDRESS = 0XBC;
    const uint8_t READ_ADDRESS = 0xBD;

    // Start from a known state
    led = 0;
    vdd_led = 0;
    vdd_dig = 0;
    vdd_ana = 0;

    intb.fall(&dummyCallbackFall);  // Setup interrupt callback
    ThisThread::sleep_for(1s);

    // Initialize power to Max chip
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Initializing max86150");
    vdd_ana = 1;
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Turned on vdd_ana");
    ThisThread::sleep_for(500ms);
    vdd_dig = 1;
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Turned on vdd_dig");
    ThisThread::sleep_for(500ms);
    vdd_led = 1;
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Turned on vdd_led");

    while(intb.read()==1){
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "No interrupt received
         ↪  yet");
        ThisThread::sleep_for(1s);
    }


    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Startup successfull");
    uint8_t flgs[2];
    char data[3];
    char buff[2];
    data[0] = 0x00; // interrupt cause register
    data[1] = 0xff; // ID register
    data[2] = 0;


    // THIS IS THE HARDCODE WAY USING HIGHER LEVEL FUNCTIONS FROM I2C CLASS
    flgs[0] = i2c.write(WRITE_ADDRESS, data, 1, true);
    flgs[1] = i2c.read(READ_ADDRESS, buff, 1);
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Received flags are: %x and
     ↪  %x", flgs[0], flgs[1]);
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Read data is: %x",
     ↪  buff[0]);


    flgs[0] = i2c.write(WRITE_ADDRESS, &data[1], 1, true);
    flgs[1] = i2c.read(READ_ADDRESS, buff, 1);
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Received flags are: %x and
     ↪  %x", flgs[0], flgs[1]);
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Read data is: %x",
     ↪  buff[0]);

    // Wait around, interrupts will interrupt this
    while(1){
    }
}

int main(){
    testall();
    //testI2c();
    while(1);
}

```

```
#endif
```

A.5. Test program output

```
NORMAL    :    Started up everything===============================================
NORMAL    :    Initializing max86150
TRACE     :    I2C frequency set to 100 kHz
NORMAL    :    Turned on vdd_ana
NORMAL    :    Turned on vdd_dig
NORMAL    :    Turned on vdd_led
NORMAL    :    PWR_RDY interrupt received
TRACE     :    Max86150 succesfully initialized
TRACE     :    Succesfully enabled interrupt 80
TRACE     :    Succesfully set fifo read pointer to e
TRACE     :    Succesfully set FIFO_ALMOST_FULL_INTTERUPT_CLEAR to 1
TRACE     :    Succesfully set FIFO_ALMOST_FULL_BEHAVIOUR to 1
TRACE     :    Succesfully set FIFO_ALMOST_FULL_BEHAVIOUR to 1
TRACE     :    Succesfully set FIFO_ALMOST_FULL_VALUE to 12
WARNING   :    Invalid argument [17]. value must be smaller than 16
TRACE     :    Succesfully set FIFO_ENABLE to 1
TRACE     :    Succesfully set SLEEP to 1
TRACE     :    Succesfully set SLEEP to 0
TRACE     :    Succesfully set PPG_ADC_RANGE to 3
TRACE     :    Succesfully set PPG_SAMPLE_RATE to 9
TRACE     :    Succesfully set PPG_PULSE_WIDTH to 2
TRACE     :    Succesfully set PPG_SAMPLE_AVERAGING to 1
TRACE     :    Succesfully set PROXIMITY_THRESHOLD to 90
TRACE     :    Succesfully set LED1_IR_PA range to 10 mA
WARNING   :    200 mA is out of range. Pulse amplitude must be smaller than 102 mA
WARNING   :    Succesfully set LED1_IR_RGE range to 100 mA
TRACE     :    Succesfully set LED1_IR_PA range to 60 mA
WARNING   :    Succesfully set LED2_IR_RGE range to 50 mA
TRACE     :    Succesfully set LED2_IR_PA range to 10 mA
WARNING   :    200 mA is out of range. Pulse amplitude must be smaller than 102 mA
WARNING   :    Succesfully set LED1_IR_RGE range to 100 mA
TRACE     :    Succesfully set LED2_IR_PA range to 60 mA
TRACE     :    Succesfully set PILOT_PA range to 10 mA
WARNING   :    200 mA is out of range. Pulse amplitude must be smaller than 102 mA
TRACE     :    Succesfully set PILOT_PA range to 60 mA
TRACE     :    Succesfully set ECG_ADC_CLK to 0
TRACE     :    Succesfully set ECG_ADC_OSR to 0
TRACE     :    Succesfully set PA_GAIN to 2 and IA_GAIN to 3
TRACE     :    Resetting all data elements for re-designation
TRACE     :    LED1_IR is set to data_element[0]
TRACE     :    Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_1
TRACE     :    Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_2
TRACE     :    This FIFO_DATA_ELEMENT is allready enabled
TRACE     :    Resetting all data elements for re-designation
TRACE     :    Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_1
TRACE     :    Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_2
TRACE     :    Resetting all data elements for re-designation
TRACE     :    LED2_RED is set to data_element[0]
TRACE     :     Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_1
TRACE     :     Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_2
TRACE     :     Resetting all data elements for re-designation
TRACE     :     LED1_IR is set to data_element[0]
TRACE     :     LED2_RED is set to data_element[1]
TRACE     :     Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_1
TRACE     :     Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_2
TRACE     :     Resetting all data elements for re-designation
TRACE     :     LED1_IR is set to data_element[0]
TRACE     :     LED2_RED is set to data_element[1]
TRACE     :     ECG is set to data_element[2]
TRACE     :     Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_1
TRACE     :     Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_2
TRACE     :     Resetting all data elements for re-designation
TRACE     :     LED1_IR is set to data_element[0]
TRACE     :     LED2_RED is set to data_element[1]
TRACE     :     PILOT_LED1_IR is set to data_element[2]
TRACE     :     ECG is set to data_element[3]
TRACE     :     Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_1
TRACE     :     Succesfully writen data to FIFO_DATA_ELEMENT_REGISTER_2
WARNING   :     Maximum number of FIFO_DATA_ELEMENTS allready reached
TRACE     :     Succesfully RESET
```

A.6. Max86150_recording.cpp

```
#include "max86150.h"
#include "logger.h"
#include "config.h"
#include <cstdint>
#include <cstdio>

#if ACTIVE_FILE == RECORDING_TEST

#define PPG_SAMPLE_RATE     100
#define RECORDING_LENGTH    30
// PPG_ADC_RGE= 32µA, PPG_SR = 100Hz, PPG_LED_PW = 400s

void intCallback();

uint32_t rec_samples = PPG_SAMPLE_RATE*RECORDING_LENGTH;
uint32_t current_samples = 0;
int32_t recording_IR[PPG_SAMPLE_RATE * RECORDING_LENGTH+100];
int32_t recording_RED[PPG_SAMPLE_RATE * RECORDING_LENGTH+100];

bool interrupt;
AFLogger::Logger logger(AFLogger::Logger::eLogLevel::WARNING);
driver::Max86150::InterruptStatus cause;
EventQueue queue(32*EVENTS_EVENT_SIZE);
Thread important(osPriorityRealtime7);
InterruptIn intb(D10);
driver::Max86150 test(&intb, 100, &logger, &queue);

void retrieveIRSensorData(){
    if(current_samples < rec_samples){
        current_samples +=
        ↪ test.getSensorData(driver::Max86150::FifoSensorsConf::LED1_IR,
        ↪ &recording_IR[current_samples]);
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Total read IR samples:
        ↪ %d", current_samples);
        test.getSensorData(driver::Max86150::FifoSensorsConf::LED2_RED,
        ↪ &recording_RED[current_samples]);
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Total read RED
        ↪ samples: %d", current_samples);
    }else{
        test.sleep(true);
        for(uint32_t i=0;i<rec_samples;i++){
            printf("%d, ", recording_IR[i]);
        }
        printf("\n\n");
        for(uint32_t i=0;i<rec_samples;i++){
            printf("%d, ", recording_RED[i]);
        }
        exit(0);
    }
}

void interruptHandler(void){
    cause = test.getInterruptStatus();

    switch (cause) {
    case driver::Max86150::InterruptStatus::A_FULL:
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Cause: A_FULL");
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "readFifo response:
         ↪  %x", test.readFifo());
        retrieveIRSensorData();
        break;
    case driver::Max86150::InterruptStatus::PWR_RDY:
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Cause: PWR_RDY");
        break;
    case driver::Max86150::InterruptStatus::ECG_RDY:
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Cause: ECG_RDY");
        break;
    case driver::Max86150::InterruptStatus::NONE:
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Cause: NONE");
        break;
    case driver::Max86150::InterruptStatus::PPG_RDY:
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Cause: PPG_RDY");
        break;
    case driver::Max86150::InterruptStatus::PROX_INT:
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Cause: PROX_INT");
        break;
    case driver::Max86150::InterruptStatus::VDD_OOR:
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Cause: VDD_OOR");
        break;
    case driver::Max86150::InterruptStatus::ALC_OVF:
        logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Cause: ALC_OVF");
        break;
    }
}

void intCallback(void){
    //queue.call(printf, "Interrupt detected\n");
    queue.call(interruptHandler);
}

void fallingCallback(void){
    queue.call(printf, "Interrupt removed\n");
}


int main(){
    // Start event queue
    logger.log(AFLogger::Logger::eLogLevel::NORMAL, "Testing");

    test.setFifoAlmostFullInterruptClear(true);
    test.setFifoAlmostFullBehaviour(false);
    test.setFifoAlmostFullValue(15);

    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::LED2_RED, true);
    test.setFifoDataElement(driver::Max86150::FifoSensorsConf::LED1_IR, true);
    test.writeFifoDataElements();
    test.setFifoEnable(true);
    //test.setFifoDataElement(driver::Max86150::FifoSensorsConf::PILOT_LED1_IR,
     ↪  true);

     ↪  //test.setFifoDataElement(driver::Max86150::FifoSensorsConf::PILOT_LED2_RED,
     ↪  true);


    test.setLedPulseAmplitude(5, driver::Max86150::LedType::INFRA_RED);
    test.setLedPulseAmplitude(5, driver::Max86150::LedType::RED);
    //test.setLedPulseAmplitude(80, driver::Max86150::LedType::PILOT);
    test.setPpgPulseWidth(driver::Max86150::PpgPulseWidthConf::MICRO_50);
    test.setPpgAdcRange(driver::Max86150::PpgAdcRangeConf::NANO_AMPERE_32768);

    //test.setProximityThreshold(128);
    //test.setInterrupt(true,
     ↪  driver::Max86150::InterruptEnableConf::PROX_INT_EN);
    test.setInterrupt(true, driver::Max86150::InterruptEnableConf::A_FULL_EN);
    //test.setInterrupt(true,
     ↪  driver::Max86150::InterruptEnableConf::ALC_OVF_EN);
    //test.setInterrupt(true,
     ↪  driver::Max86150::InterruptEnableConf::VDD_OOR_EN);
    //test.setInterrupt(false,
     ↪  driver::Max86150::InterruptEnableConf::PPG_RDY_EN);

    important.start(callback(&queue, &EventQueue::dispatch_forever));
    intb.fall(intCallback);
    //intb.rise(fallingCallback);
    queue.call(printf, "Queue test\n");
    interruptHandler();

     ↪  test.setPpgSampleRate(driver::Max86150::PpgSampleRateConf::SAMPLES_200_PPS_1);

     ↪  test.setPpgSampleAveraging(driver::Max86150::SampleAveragingConf::SAMPLE_AVERAGE_2);
    while(true){
    }
}

#endif
```

B

PCB

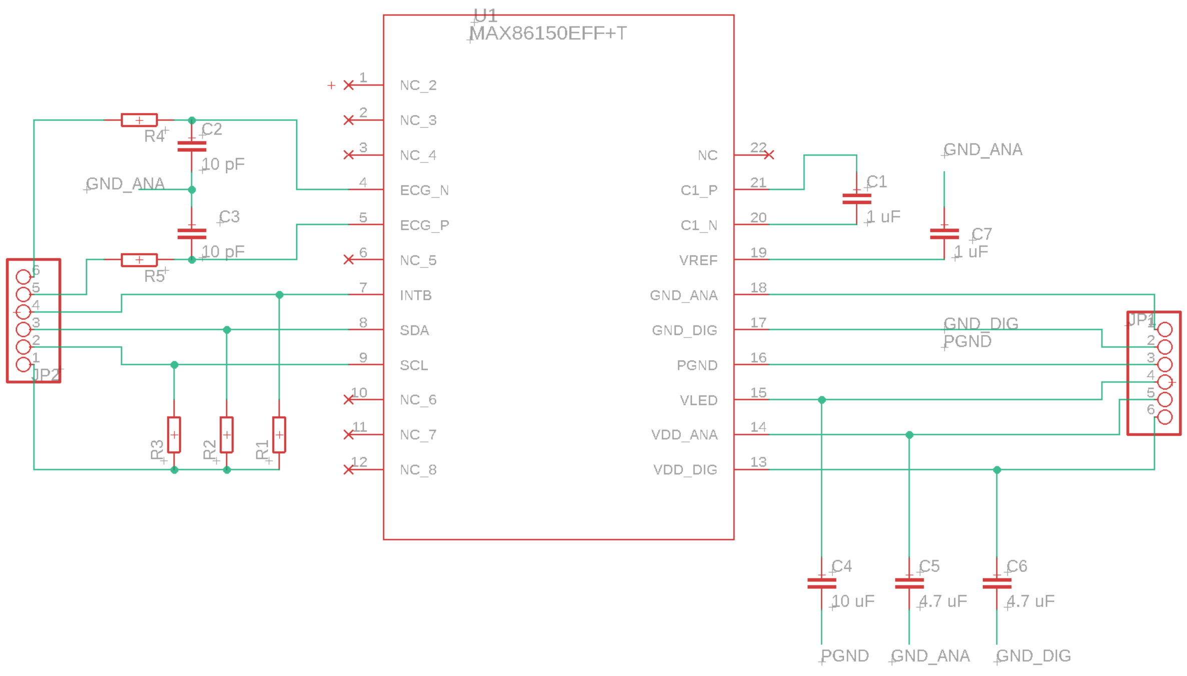


Figure B.1: The schematic of the PCB circuit.

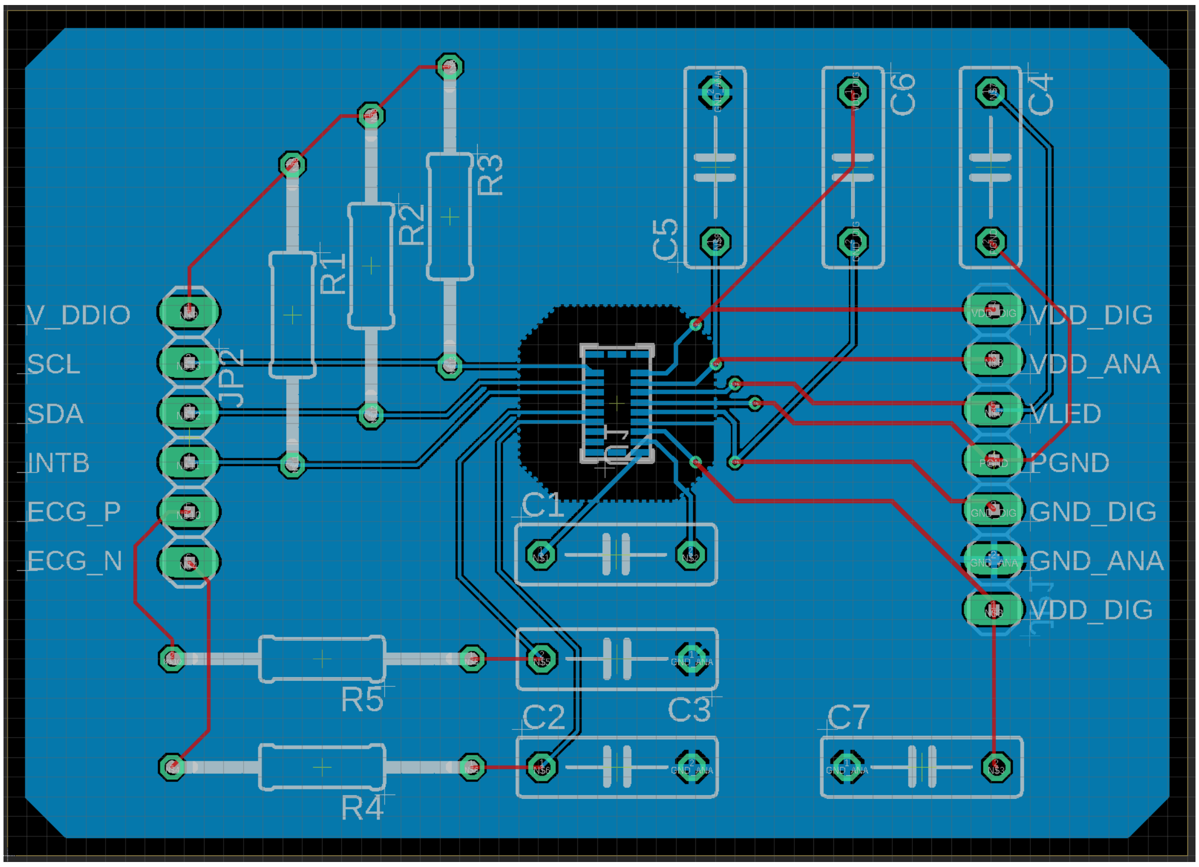


Figure B.2: Top view of the designed PCB layout.

C
Sensor Recordings

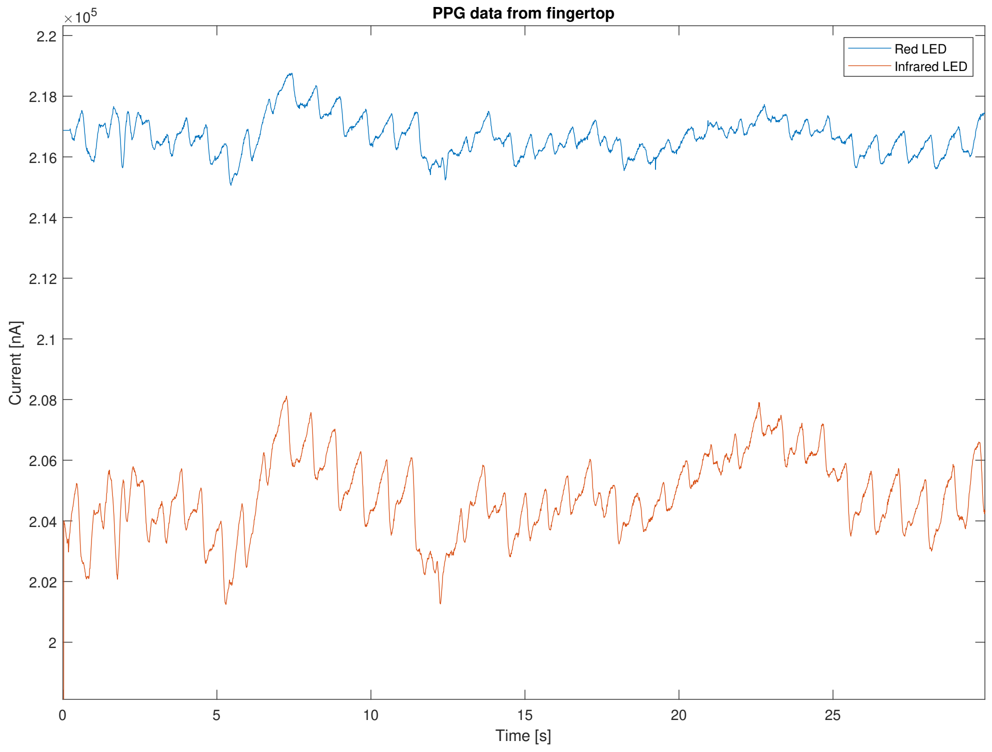


Figure C.1: PPG recording at the tip of the middle finger with a red and infrared LED.

Bibliography

[1] G. Lippi, F. Sanchis-Gomar, and G. Cervellin, “Global epidemiology of atrial fibrillation: An increasing epidemic and public health challenge,” *International Journal of Stroke*, vol. 16, no. 2, pp. 217–221, 2021, pMID: 31955707. [Online]. Available: https://doi.org/10.1177/1747493019897870

[2] B. A. Schoonderwoerd, M. D. Smit, L. Pen, and I. C. Van Gelder, “New risk factors for atrial fibrillation: causes of ‘not-so-lone atrial fibrillation’,” *EP Europace*, vol. 10, no. 6, pp. 668–673, 05 2008. [Online]. Available: https://doi.org/10.1093/europace/eun124

[3] A. Aizer, J. M. Gaziano, N. R. Cook, J. E. Manson, J. E. Buring, and C. M. Albert, “Relation of vigorous exercise to risk of atrial fibrillation,” *The American Journal of Cardiology*, vol. 103, no. 11, pp. 1572–1577, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0002914909005499

[4] M. Myrstad, W. Nystad, S. Graff-Iversen, D. S. Thelle, H. Stigum, M. Aarønæs, and A. H. Ranhoff, “Effect of years of endurance exercise on risk of atrial fibrillation and atrial flutter,” *The American Journal of Cardiology*, vol. 114, no. 8, pp. 1229–1233, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0002914914015082

[5] E. Guasch, B. Benito, X. Qi, C. Cifelli, P. Naud, Y. Shi, A. Mighiu, J.-C. Tardif, A. Tadevosyan, Y. Chen, M.-A. Gillis, Y.-K. Iwasaki, D. Dobrev, L. Mont, S. Heximer, and S. Nattel, “Atrial fibrillation promotion by endurance exercise,” *Journal of the American College of Cardiology*, vol. 62, no. 1, pp. 68–77, 2013. [Online]. Available: https://www.jacc.org/doi/abs/10.1016/j.jacc.2013.01.091

[6] A. Jahangir, V. Lee, P. Friedman, J. Trusty, D. Hodge, S. Kopecky, D. M.D, S. Hammill, W.-K. Shen, and B. Gersh, “Long-term progression and outcomes with aging in patients with lone atrial fibrillation: A 30-year follow-up study,” *Circulation*, vol. 115, pp. 3050–6, 07 2007.

[7] S. Barold, “Willem einthoven and the birth of clinical electrocardiography a hundred years ago,” *Cardiac electrophysiology review*, vol. 7, pp. 99–104, 02 2003.

[8] AliveCor, “Kardiamobile,” 2005, accessed on: June 15, 2021. [Online]. Available: https://store.alivecor.com/products/kardiamobile

[9] Fitbit, “Sense,” accessed on: June 15, 2021. [Online]. Available: https://www.fitbit.com/global/us/products/smartwatches/sense

[10] A. Inc., “Using apple watch for arrhythmia detection,” 2020, accessed on: June 15, 2021. [Online]. Available: https://www.apple.com/uk/healthcare/docs/site/Apple_Watch_Arrhythmia_Detection.pdf

[11] L. Savvides and S. Scott, “Fitbit sense review: An ambitious smartwatch that’s getting better with time,” CNET, accessed on: June 18, 2021. [Online]. Available: https://www.cnet.com/news/fitbit-sense-review-ambitious-smartwatch-fitness-tracker-getting-better-with-time/

[12] D. Phelan, "Apple watch series 4 ecg heart rate feature now live: Here's all you need to know," Forbes, accessed on: June 18, 2021. [Online]. Available: https://www.forbes.com/sites/davidphelan/2018/12/06/apple-watch-series-4-ecg-heart-rate-feature-now-live-heres-all-you-need-to-know/?sh=5fcf048f32c5

[13] Y. Firth and P. Errico, "Low-power, low-voltage ic choices for ecg system requirements," *Analog Dialogue*, vol. 29, no. 3, pp. 9–10, 1995.

[14] T. Dobreva and T. Stoyanov, "High-resolution front-end for ecg signal processing," 01 2007.

[15] A. K. Y. Wong, K.-P. Pun, Y.-T. Zhang, and K. N. Leung, "A low-power cmos front-end for photoplethysmographic signal acquisition with robust dc photocurrent rejection," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 2, no. 4, pp. 280–288, 2008.

[16] *SFH 7050 - Photoplethysmography Sensor Application Note*, OSRAM Opto Semiconductors, June 2014. [Online]. Available: https://dammedia.osram.info/media/resource/hires/osram-dam-2496553/SFH%207050%20-%20Photoplethysmography%20Sensor.pdf

[17] SecondsCount, "What is atrial fibrillation (afib or af)?" 2015, accessed on: June 14, 2021. [Online]. Available: http://www.secondscount.org/heart-condition-centers/info-detail-2/what-is-atrial-fibrillation-afib-af#.YMcXBuFR2Uk

[18] H. Calkins, K.-H. Kuck, R. Cappato, J. Brugada, J. Camm, S.-A. Chen, H. Crijns, R. Damiano, D. Davies, J. Dimarco, J. Edgerton, K. Ellenbogen, M. Ezekowitz, D. Haines, M. Haissaguerre, G. Hindricks, Y. Iesaka, W. Jackman, J. Jalife, and D. Wilber, "2012 hrs/ehra/ecas expert consensus statement on catheter and surgical ablation of atrial fibrillation: recommendations for patient selection, procedural techniques, patient management and follow-up, definitions, endpoints, and research trial design," *Europace : European pacing, arrhythmias, and cardiac electrophysiology : journal of the working groups on cardiac pacing, arrhythmias, and cardiac cellular electrophysiology of the European Society of Cardiology*, vol. 14, pp. 528–606, 03 2012.

[19] R. W. Troughton, C. R. Asher, and A. L. Klein, "The role of echocardiography in atrial fibrillation and cardioversion," *Heart*, vol. 89, no. 12, pp. 1447–1454, 2003. [Online]. Available: https://heart.bmj.com/content/89/12/1447

[20] A. Weymann, S. Ali-Hasan-Al-Saegh, A. Sabashnikov, A.-F. Popov, S. J. Mirhosseini, T. Liu, M. Lotfaliani, M. P. B. de Oliveira Sá, W. L. Baker, S. Yavuz *et al.*, "Prediction of new-onset and recurrent atrial fibrillation by complete blood count tests: a comprehensive systematic review with meta-analysis," *Medical science monitor basic research*, vol. 23, pp. 179 – 222, 2017.

[21] I. Beaulieu-Boire, N. Leblanc, L. Berger, and J.-M. Boulanger, "Troponin elevation predicts atrial fibrillation in patients with stroke or transient ischemic attack," *Journal of Stroke and Cerebrovascular Diseases*, vol. 22, no. 7, pp. 978–983, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1052305712000134

[22] C. Selmer, J. B. Olesen, M. L. Hansen, J. Lindhardsen, A.-M. S. Olsen, J. C. Madsen, J. Faber, P. R. Hansen, O. D. Pedersen, C. Torp-Pedersen, and G. H. Gislason, "The spectrum of thyroid disease and risk of new onset atrial fibrillation: a large population cohort study," *BMJ*, vol. 345, 2012. [Online]. Available: https://www.bmj.com/content/345/bmj.e7895

[23] A. Tarniceriu, J. Harju, Z. Rezaei Yousefi, A. Vehkaoja, J. Parák, A. Yli-Hankala, and I. Korhonen, "The accuracy of atrial fibrillation detection from wrist photoplethysmography. a study

on post-operative patients," in *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, vol. 2018, 07 2018.

[24] Y. Shen, M. Voisin, A. Aliamiri, A. Avati, A. Hannun, and A. Ng, "Ambulatory atrial fibrillation monitoring using wearable photoplethysmography with deep learning," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1909–1916. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/3292500.3330657

[25] S. K. Bashar, D. Han, E. Ding, C. Whitcomb, D. D. McManus, and K. H. Chon, "Smartwatch based atrial fibrillation detection from photoplethysmography signals," in *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2019, pp. 4306–4309.

[26] S. K. Bashar, D. Han, S. Hajeb, E. Ding, C. Whitcomb, D. McManus, and K. Chon, "Atrial fibrillation detection from wrist photoplethysmography signals using smartwatches," *Scientific Reports*, vol. 9, 10 2019.

[27] P. Cheng, Z. Chen, Q. Li, Q. Gong, J. Zhu, and Y. Liang, "Atrial fibrillation identification with ppg signals using a combination of time-frequency analysis and deep learning," *IEEE Access*, vol. 8, pp. 172 692–172 706, 2020.

[28] J. Selder, T. Proesmans, L. Breukel, O. Dur, W. Gielen, A. van Rossum, and C. Allaart, "Assessment of a standalone photoplethysmography (ppg) algorithm for detection of atrial fibrillation on wristband-derived data," *Computer Methods and Programs in Biomedicine*, vol. 197, p. 105753, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169260720315868

[29] J. R. Hampton and J. Hampton, *The ECG made easy.* Elsevier, 2019.

[30] J. Heuser, "Atrial fibrillation," Wikipedia, 2005, accessed on: June 14, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Atrial_fibrillation

[31] R. Alcaraz and J. J. Rieta, "Adaptive singular value cancelation of ventricular activity in single-lead atrial fibrillation electrocardiograms," *Physiological Measurement*, vol. 29, no. 12, pp. 1351–1369, October 2008. [Online]. Available: https://doi.org/10.1088/0967-3334/29/12/001

[32] X. Fan, Q. Yao, Y. Cai, F. Miao, F. Sun, and Y. Li, "Multiscaled fusion of deep convolutional neural networks for screening atrial fibrillation from single lead short ecg recordings," *IEEE Journal of Biomedical and Health Informatics*, vol. 22, no. 6, p. 1744—1753, November 2018. [Online]. Available: https://doi.org/10.1109/JBHI.2018.2858789

[33] F. Castells, C. Mora, J. Rieta, D. Moratal-Pérez, and J. Millet, "Estimation of atrial fibrillatory wave from single-lead atrial fibrillation electrocardiograms using principal component analysis concepts," *Medical & Biological Engineering & Computing*, vol. 43, no. 5, p. 557—560, September 2005. [Online]. Available: https://doi.org/10.1007/bf02351028

[34] E. Svennberg, M. Stridh, J. Engdahl, F. Al-Khalili, L. Friberg, V. Frykman, and M. Rosenqvist, "Safe automatic one-lead electrocardiogram analysis in screening for atrial fibrillation," *EP Europace*, vol. 19, no. 9, pp. 1449–1453, 10 2016. [Online]. Available: https://doi.org/10.1093/europace/euw286

[35] R. Duarte, A. Stainthorpe, J. Mahon, J. Greenhalgh, M. Richardson, S. Nevitt, E. Kotas, A. Boland, H. Thom, T. Marshall, M. Hall, and Y. Takwoingi, "Lead-i ecg for detecting atrial

fibrillation in patients attending primary care with an irregular pulse using single-time point testing: A systematic review and economic evaluation," *PLOS ONE*, vol. 14, no. 12, pp. 1–17, 12 2019. [Online]. Available: https://doi.org/10.1371/journal.pone.0226671

[36] D. R. Seshadri, B. Bittel, D. Browsky, P. Houghtaling, C. K. Drummond, M. Y. Desai, and A. M. Gillinov, "Accuracy of apple watch for detection of atrial fibrillation," *Circulation*, vol. 141, no. 8, pp. 702–703, 2020. [Online]. Available: https://www.ahajournals.org/doi/abs/10.1161/CIRCULATIONAHA.119.044126

[37] N. Saghir, A. Aggarwal, N. Soneji, V. Valencia, G. Rodgers, and T. Kurian, "A comparison of manual electrocardiographic interval and waveform analysis in lead 1 of 12-lead ecg and apple watch ecg: A validation study," *Cardiovascular Digital Health Journal*, vol. 1, no. 1, pp. 30–36, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666693620300062

[38] Z. C. HABERMAN, R. T. JAHN, R. BOSE, H. TUN, J. S. SHINBANE, R. N. DOSHI, P. M. CHANG, and L. A. SAXON, "Wireless smartphone ecg enables large-scale screening in diverse populations," *Journal of Cardiovascular Electrophysiology*, vol. 26, no. 5, pp. 520–526, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/jce.12634

[39] M. R. F. Gropler, A. S. Dalal, G. F. Van Hare, and J. N. A. Silva, "Can smartphone wireless ecgs be used to accurately assess ecg intervals in pediatrics? a comparison of mobile health monitoring to standard 12-lead ecg," *PLOS ONE*, vol. 13, no. 9, pp. 1–9, 09 2018. [Online]. Available: https://doi.org/10.1371/journal.pone.0204403

[40] B. B. Winter and J. G. Webster, "Driven-right-leg circuit design," *IEEE Transactions on Biomedical Engineering*, vol. BME-30, no. 1, pp. 62–66, 1983.

[41] N. V. Thakor and J. G. Webster, "Ground-free ecg recording with two electrodes," *IEEE Transactions on Biomedical Engineering*, vol. BME-27, no. 12, pp. 699–704, 1980.

[42] A. Zompanti, A. Sabatini, S. Grasso, G. Pennazza, G. Ferri, G. Barile, M. Chello, M. Lusini, and M. Santonico, "Development and test of a portable ecg device with dry capacitive electrodes and driven right leg circuit," *Sensors*, vol. 21, no. 8, 2021. [Online]. Available: https://www.mdpi.com/1424-8220/21/8/2777

[43] A. Samol, K. Bischof, B. Luani, D. Pascut, M. Wiemer, and S. Kaese, "Recording of bipolar multichannel ecgs by a smartwatch: Modern ecg diagnostic 100 years after einthoven," *Sensors*, vol. 19, no. 13, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/13/2894

[44] S. Yao and Y. Zhu, "Nanomaterial-Enabled Dry Electrodes for Electrophysiological Sensing: A Review," *JOM - Journal of the Minerals, Metals and Materials Society*, vol. 68, no. 4, pp. 1145–1155, 2016. [Online]. Available: https://doi.org/10.1007/s11837-016-1818-0

[45] M. Chawla, "A comparative analysis of principal component and independent component techniques for electrocardiograms," *Neural Computing and Applications*, vol. 18, no. 6, p. 539–556, 09 2009.

[46] I. Romero, "Pca and ica applied to noise reduction in multi-lead ecg," in *2011 Computing in Cardiology*, 2011, pp. 613–616.

[47] M. Raya and L. Sison, "Adaptive noise cancelling of motion artifact in stress ecg signals using accelerometer," in *Proceedings of the Second Joint 24th Annual Conference and the Annual Fall*

Meeting of the Biomedical Engineering Society] [Engineering in Medicine and Biology, vol. 2, 2002, pp. 1756–1757 vol.2.

[48] S. Yoon, S. D. Min, Y. Yun, S. Lee, and M. Lee, "Adaptive motion artifacts reduction using 3-axis accelerometer in e-textile ecg measurement system," *Journal of Medical Systems*, vol. 32, pp. 101–106, 05 2008.

[49] D. Tong, K. Bartels, and K. Honeyager, "Adaptive reduction of motion artifact in the electrocardiogram," in *Proceedings of the Second Joint 24th Annual Conference and the Annual Fall Meeting of the Biomedical Engineering Society] [Engineering in Medicine and Biology*, vol. 2, 2002, pp. 1403–1404 vol.2.

[50] P. Hamilton, M. Curley, R. Aimi, and C. Sae-Hau, "Comparison of methods for adaptive removal of motion artifact," in *Computers in Cardiology 2000. Vol.27 (Cat. 00CH37163)*, 2000, pp. 383–386.

[51] S. Kim, R. F. Yazicioglu, T. Torfs, B. Dilpreet, P. Julien, and C. Van Hoof, "A 2.4µa continuous-time electrode-skin impedance measurement circuit for motion artifact monitoring in ecg acquisition systems," in *2010 Symposium on VLSI Circuits*, 2010, pp. 219–220.

[52] A. B. Hertzman, "The blood supply of various skin areas as estimated by the photoelectric plethysmograph," *American Journal of Physiology-Legacy Content*, vol. 124, no. 2, pp. 328–340, 1938. [Online]. Available: https://doi.org/10.1152/ajplegacy.1938.124.2.328

[53] M. Elgendi, *PPG Signal Analysis: An Introduction Using MATLAB*. CRC Press, Taylor & Francis Group, 2021.

[54] S. Das, S. Pal, and M. Mitra, "Real time heart rate detection from ppg signal in noisy environment," in *2016 International Conference on Intelligent Control Power and Instrumentation (ICICPI)*, 2016, pp. 70–73.

[55] N. Pinheiro, R. Couceiro, J. Henriques, J. Muehlsteff, I. Quintal, L. Gonçalves, and P. Carvalho, "Can ppg be used for hrv analysis?" in *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, 2016, pp. 2945–2949.

[56] E. Gil, M. Orini, R. Bailón, J. Vergara, L. Mainardi, and P. Laguna, "Photoplethysmography pulse rate variability as a surrogate measurement of heart rate variability during non-stationary conditions," *Physiological measurement*, vol. 31, pp. 1271–90, 09 2010.

[57] A. C. Podaru, V. David, and O. M. Asiminicesei, "Determination and comparison of heart rate variability and pulse rate variability," in *2018 International Conference and Exposition on Electrical And Power Engineering (EPE)*, 2018, pp. 0551–0554.

[58] E. Mejía-Mejía, K. Budidha, T. Y. Abay, J. M. May, and P. A. Kyriacou, "Heart rate variability and multi-site pulse rate variability for the assessment of autonomic responses to whole-body cold exposure," in *2020 42nd Annual International Conference of the IEEE Engineering in Medicine Biology Society (EMBC)*, 2020, pp. 2618–2621.

[59] H. Shin, "Ambient temperature effect on pulse rate variability as an alternative to heart rate variability in young adult," *Journal of Clinical Monitoring and Computing*, vol. 30, pp. 939 – 948, 2015.

[60] J. G. Webster, *Design of Pulse Oximeters*. Taylor & Francis Group, 1997.

[61] J. Allen, “Photoplethysmography and its application in clinical physiological measurement,” *Physiological Measurement*, vol. 28, pp. R1–39, 04 2007.

[62] M. R. Ram, K. V. Madhav, E. H. Krishna, N. R. Komalla, and K. A. Reddy, “A novel approach for motion artifact reduction in ppg signals based on as-lms adaptive filter,” *IEEE Transactions on Instrumentation and Measurement*, vol. 61, no. 5, pp. 1445–1457, 2012.

[63] G. Joseph, A. Joseph, G. Titus, R. M. Thomas, and D. Jose, “Photoplethysmogram (ppg) signal analysis and wavelet de-noising,” in *2014 Annual International Conference on Emerging Research Areas: Magnetics, Machines and Drives (AICERA/iCMMD)*, 2014, pp. 1–5.

[64] T. Tamura, “Current progress of photoplethysmography and spo2 for health monitoring,” *Biomedical Engineering Letters*, vol. 9, 02 2019.

[65] A. Reisner, P. A. Shaltis, D. McCombie, and H. H. Asada, “Utility of the photoplethysmogram in circulatory monitoring,” *Anesthesiology*, vol. 108, no. 5, p. 950—958, May 2008. [Online]. Available: https://doi.org/10.1097/ALN.0b013e31816c89e1

[66] T. Tamura, Y. Maeda, M. Sekine, and M. Yoshida, “Wearable photoplethysmographic sensors—past and present,” *Electronics*, vol. 3, no. 2, pp. 282–302, 2014. [Online]. Available: https://www.mdpi.com/2079-9292/3/2/282

[67] S. K. Longmore, G. Y. Lui, G. Naik, P. P. Breen, B. Jalaludin, and G. D. Gargiulo, “A comparison of reflective photoplethysmography for detection of heart rate, blood oxygen saturation, and respiration rate at various anatomical locations,” *Sensors*, vol. 19, no. 8, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/8/1874

[68] R. Wang, G. Blackburn, M. Desai, D. Phelan, L. Gillinov, P. Houghtaling, and M. Gillinov, “Accuracy of Wrist-Worn Heart Rate Monitors,” *JAMA Cardiology*, vol. 2, no. 1, pp. 104–106, 01 2017.

[69] S. Lee, H. Shin, and C. Hahm, “Effective ppg sensor placement for reflected red and green light, and infrared wristband-type photoplethysmography,” in *2016 18th International Conference on Advanced Communication Technology (ICACT)*, 2016, pp. 556–558.

[70] Y. Lee, H. Shin, J. Jo, and Y.-K. Lee, “Development of a wristwatch-type ppg array sensor module,” in *2011 IEEE International Conference on Consumer Electronics -Berlin (ICCE-Berlin)*, 2011, pp. 168–171.

[71] B. Bent, B. Goldstein, W. Kibbe, and J. Dunn, “Investigating sources of inaccuracy in wearable optical heart rate sensors,” *npj Digital Medicine*, vol. 3, p. 18, 02 2020.

[72] D. Phan, L. Y. Siong, P. N. Pathirana, and A. Seneviratne, “Smartwatch: Performance evaluation for long-term heart rate monitoring,” in *2015 International Symposium on Bioelectronics and Bioinformatics (ISBB)*, 2015, pp. 144–147.

[73] M. Wallen, S. Gomersall, S. Keating, U. Wisloff, and J. Coombes, “Accuracy of heart rate watches: Implications for weight management,” *PloS one*, vol. 11, p. e0154420, 05 2016.

[74] H. Kinnunen, A. Rantanen, T. Kenttä, and H. Koskimäki, “Feasible assessment of recovery and cardiovascular health: Accuracy of nocturnal hr and hrv assessed via ring ppg in comparison to medical grade ecg,” *Physiological Measurement*, vol. 41, 03 2020.

[75] Oura, “Oura ring heritage - silver,” accessed on: June 18, 2021. [Online]. Available: https://ouraring.com/product/heritage-silver/step1

[76] Y. Maeda, M. Sekine, T. Tamura, A. Moriya, T. Suzuki, and K.-I. Kameyama, "Comparison of reflected green light and infrared photoplethysmography," *2008 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Conference*, vol. 2008, pp. 2270–2, 02 2008.

[77] Y. Maeda, M. Sekine, and T. Tamura, "The advantages of wearable green reflected photoplethysmography," *Journal of medical systems*, vol. 35, pp. 829–34, 10 2011.

[78] R. R. Anderson and J. A. Parrish, "The optics of human skin," *Journal of Investigative Dermatology*, vol. 77, no. 1, pp. 13–19, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022202X15461251

[79] V. Vizbara, "Comparison of green, blue and infrared light in wrist and forehead photoplethysmography," *BIOMEDICAL ENGINEERING*, vol. 17, no. 1, pp. 78–81, 2013.

[80] J. Lee, K. Matsumura, K.-i. Yamakoshi, P. Rolfe, S. Tanaka, and T. Yamakoshi, "Comparison between red, green and blue light reflection photoplethysmography for heart rate monitoring during motion," *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Conference*, vol. 2013, pp. 1724–1727, 07 2013.

[81] W. Cui, L. Ostrander, and B. Lee, "In vivo reflectance of blood and tissue as a function of light wavelength," *IEEE Transactions on Biomedical Engineering*, vol. 37, no. 6, pp. 632–639, 1990.

[82] N. Kollias, "The spectroscopy of human melanin pigmentation," *Melanin: Its Role in Human Photoprotection*, pp. 31–38, 01 1995.

[83] G. Zonios, J. Bykowski, and N. Kollias, "Skin melanin, hemoglobin, and light scattering properties can be quantitatively assessed in vivo using diffuse reflectance spectroscopy," *Journal of Investigative Dermatology*, vol. 117, no. 6, pp. 1452–1457, 2001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022202X15414836

[84] B. Fallow, T. Tarumi, and H. Tanaka, "Influence of skin type and wavelength on light wave reflectance," *Journal of clinical monitoring and computing*, vol. 27, 02 2013.

[85] R. Hailu, "Fitbits and other wearables may not accurately track heart rates in people of color," *Stat*, July 2019. [Online]. Available: https://www.statnews.com/2019/07/24/fitbit-accuracy-dark-skin/

[86] M. Zbrog, "Heart rate fitness trackers may be less reliable for darker skin tones," *The Swaddle*, August 2019. [Online]. Available: https://theswaddle.com/are-heart-rate-trackers-accurate-for-dark-skin-maybe-not/

[87] ——, "Health equity 101: The racial biases of smartwatches & other healthcare tech," *HealthcareDegree*, October 2020. [Online]. Available: https://www.healthcaredegree.com/blog/biases-in-wearable-health-tech

[88] A. Boxall, "Wearables don't work the same on dark skin. it's time to change that," *Digital Trends*, February 2021. [Online]. Available: https://www.digitaltrends.com/mobile/does-skin-tone-affect-ppg-heart-rate-sensor-accuracy/

[89] B. Sañudo, M. De Hoyo, A. Muñoz-López, J. Perry, and G. Abt, "Pilot study assessing the influence of skin type on the heart rate measurements obtained by photoplethysmography with the apple watch," *Journal of medical systems*, vol. 43, no. 7, p. 195, May 2019. [Online]. Available: https://doi.org/10.1007/s10916-019-1325-2

[90] M. P. Wallen, S. R. Gomersall, S. E. Keating, U. Wisløff, and J. S. Coombes, "Accuracy of heart rate watches: Implications for weight management," *PLOS ONE*, vol. 11, no. 5, pp. 1–11, 05 2016. [Online]. Available: https://doi.org/10.1371/journal.pone.0154420

[91] A. Shcherbina, C. M. Mattsson, D. Waggott, H. Salisbury, J. W. Christle, T. Hastie, M. T. Wheeler, and E. A. Ashley, "Accuracy in wrist-worn, sensor-based measurements of heart rate and energy expenditure in a diverse cohort," *Journal of Personalized Medicine*, vol. 7, no. 2, 2017. [Online]. Available: https://www.mdpi.com/2075-4426/7/2/3

[92] J. Emery, "Skin pigmentation as an influence on the accuracy of pulse oximetry," *Journal of perinatology: official journal of the California Perinatal Association*, vol. 7, no. 4, pp. 329–330, 1987.

[93] A. L. Ries, L. M. Prewitt, and J. J. Johnson, "Skin color and ear oximetry," *Chest*, vol. 96, no. 2, pp. 287–290, 1989. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0012369215459161

[94] P. E. Bickler, J. R. Feiner, and J. W. Severinghaus, "Effects of Skin Pigmentation on Pulse Oximeter Accuracy at Low Saturation," *Anesthesiology*, vol. 102, no. 4, pp. 715–719, 04 2005. [Online]. Available: https://doi.org/10.1097/00000542-200504000-00004

[95] P. Bothma, G. Joynt, J. Lipman, H. Hon, B. Mathala, J. Scribante, and J. Kromberg, "Accuracy of pulse oximetry in pigmented patients," *South African medical journal*, vol. 86, pp. 594–596, 06 1996.

[96] J. Adler, L. Hughes, R. Vivilecchia, and C. Camargo, "Effect of skin pigmentation on pulse oximetry accuracy in the emergency department," *Academic emergency medicine : official journal of the Society for Academic Emergency Medicine*, vol. 5, no. 10, p. 965—970, October 1998. [Online]. Available: https://doi.org/10.1111/j.1553-2712.1998.tb02772.x

[97] E. E. Foglia, R. K. Whyte, A. Chaudhary, A. Mott, J. Chen, K. J. Propert, and B. Schmidt, "The effect of skin pigmentation on the accuracy of pulse oximetry in infants with hypoxemia," *The Journal of Pediatrics*, vol. 182, pp. 375–377, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022347616312720

[98] R. Shriram, M. Sundhararajan, and N. Daimiwal, "Effect of change in intensity of infrared led on a photoplethysmogram," in *2014 International Conference on Communication and Signal Processing*, 2014, pp. 1064–1067.

[99] R. Yousefi, M. Nourani, and I. Panahi, "Adaptive cancellation of motion artifact in wearable biosensors," in *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2012, pp. 2004–2008.

[100] H.-W. Lee, J. Lee, W.-G. Jung, and G.-K. Lee, "The periodic moving average filter for removing motion artifacts from ppg signals," *International Journal of Control Automation and Systems*, vol. 5, pp. 701–706, 2007.

[101] H. Asada, H.-H. Jiang, and P. Gibbs, "Active noise cancellation using mems accelerometers for motion-tolerant wearable bio-sensors," in *The 26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, vol. 1, 2004, pp. 2157–2160.

[102] M. Boloursaz Mashhadi, E. Asadi, M. Eskandari, S. Kiani, and F. Marvasti, "Heart rate tracking using wrist-type photoplethysmographic (ppg) signals during physical exercise with simultaneous accelerometry," *IEEE Signal Processing Letters*, vol. 23, no. 2, pp. 227–231, 2016.

[103] S. H. Kim, D. W. Ryoo, and C. Bae, “Adaptive noise cancellation using accelerometers for the ppg signal from forehead,” in *2007 29th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 2007, pp. 2564–2567.

[104] M. Boloursaz Mashhadi, E. Asadi, M. Eskandari, S. Kiani, and F. Marvasti, “Heart rate tracking using wrist-type photoplethysmographic (ppg) signals during physical exercise with simultaneous accelerometry,” *IEEE Signal Processing Letters*, vol. 23, no. 2, pp. 227–231, 2016.

[105] Y. Ye, Y. Cheng, W. He, M. Hou, and Z. Zhang, “Combining nonlinear adaptive filtering and signal decomposition for motion artifact removal in wearable photoplethysmography,” *IEEE Sensors Journal*, vol. 16, no. 19, pp. 7133–7141, 2016.

[106] M. R. Ram, K. V. Madhav, E. H. Krishna, N. R. Komalla, and K. A. Reddy, “A novel approach for motion artifact reduction in ppg signals based on as-lms adaptive filter,” *IEEE Transactions on Instrumentation and Measurement*, vol. 61, no. 5, pp. 1445–1457, 2012.

[107] B. Kim and S. Yoo, “Motion artifact reduction in photoplethysmography using independent component analysis,” *IEEE Transactions on Biomedical Engineering*, vol. 53, no. 3, pp. 566–568, 2006.

[108] R. Yousefi, M. Nourani, S. Ostadabbas, and I. Panahi, “A motion-tolerant adaptive algorithm for wearable photoplethysmographic biosensors,” *IEEE Journal of Biomedical and Health Informatics*, vol. 18, no. 2, pp. 670–681, 2014.

[109] Y. Zhang, S. Song, R. Vullings, D. Biswas, N. Simões-Capela, N. van Helleputte, C. van Hoof, and W. Groenendaal, “Motion artifact reduction for wrist-worn photoplethysmograph sensors based on different wavelengths,” *Sensors*, vol. 19, no. 3, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/3/673

[110] J. Lee, M. Kim, H.-K. Park, and I. Y. Kim, “Motion artifact reduction in wearable photoplethysmography based on multi-channel sensors with multiple wavelengths,” *Sensors*, vol. 20, no. 5, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/5/1493

[111] A. A. Silverio, J. E. Asilo, S. L. D. Balagat, A. A. Pineda, J. A. Tangi, D. T. P. Tuazon, and B. M. B. Piedra, “Micromotion artefact reduction of a wrist worn ppg sensor using green light ppg and surface emg,” in *2020 IEEE 8th R10 Humanitarian Technology Conference (R10-HTC)*, 2020, pp. 1–5.

[112] Y. Yamaguchi, S. Itami, H. Watabe, K. Yasumoto, Z. A. Abdel-Malek, T. Kubo, F. Rouzaud, A. Tanemura, K. Yoshikawa, and V. J. Hearing, “Mesenchymal-epithelial interactions in the skin: increased expression of dickkopf1 by palmoplantar fibroblasts inhibits melanocyte growth and differentiation,” *J Cell Biol*, vol. 165, no. 2, pp. 275–285, Apr 2004.

[113] M. Integrated, “Max86150 integrated photoplethysmogram and electrocardiogram biosensor module for mobile health.” [Online]. Available: https://datasheets.maximintegrated.com/en/ds/MAX86150.pdf

[114] F. Semiconductor, “Xtrinsic mma8451q 3-axis, 14-bit/8-bit digital accelerometer,” accessed on: June 18, 2021. [Online]. Available: https://docs.rs-online.com/8f5f/0900766b814beed7.pdf

[115] arm, “Arm cortex m7 processor datasheet,” accessed on: June 18, 2021. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm-Cortex-M7-Processor-Datasheet.

pdf?revision=35233191-7cdb-429e-9170-e07ad9d87c7a&la=en&hash=5C37DFAA43B532B8BD2DB277685F76007FCFEAE5

[116] STM, "Stm32 nucleo-144 development board with stm32h743zi mcu, supports arduino, st zio and morpho connectivity," accessed on: June 18, 2021. [Online]. Available: https://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-mpu-eval-tools/stm32-mcu-mpu-eval-tools/stm32-nucleo-boards/nucleo-h743zi.html

[117] P. M. M. M. Adrianus and S. Herman, "Two-wire bus-system comprising a clock wire and a data wire for interconnecting a number of stations," Patent 4 689 740, 1987. [Online]. Available: https://patentimages.storage.googleapis.com/bd/83/50/99109ed29b7c19/US4689740.pdf

[118] SFUPTOWNMAKER, "I2c basic address and data frames," accessed on: June 11, 2021. [Online]. Available: https://cdn.sparkfun.com/assets/learn_tutorials/8/2/I2C_Basic_Address_and_Data_Frames.jpg

[119] mbed, "An introduction to arm mbed os 6," accessed on: June 18, 2021. [Online]. Available: https://os.mbed.com/docs/mbed-os/v6.11/introduction/index.html

[120] A. Limited, "I2c class reference," accessed on: June 14, 2021. [Online]. Available: https://os.mbed.com/docs/mbed-os/v6.11/mbed-os-api-doxy/classmbed_1_1_i2_c.html

[121] N.-C. Lee, *Reflow soldering processes.* Newnes, 2002.

[122] Zithan, "Rss components of a thermal profile," accessed on: June 16, 2021. [Online]. Available: https://commons.wikimedia.org/wiki/File:RSS_Components_of_a_Profile.svg