

UNIVERSITÀ
DEGLI STUDI
DI PADOVA



DEPARTMENT OF INFORMATION ENGINEERING
MASTER DEGREE IN ELECTRONIC ENGINEERING

Development of a wearable insole for heart rate monitoring

Supervisor

Prof. Giorgi Giada

Author

Jahreis Thomas

ACCADEMIC YEAR 2024-2025

Graduation date 10/07/2025

Sommario

Questa tesi si propone di sviluppare un sistema di acquisizione per la rilevazione del segnale cardiaco dalla pianta del piede. La misurazione della frequenza cardiaca da questa zona rappresenta un ambito di ricerca ancora poco esplorato, ma con potenzialità interessanti. In particolare, l'integrazione del sistema all'interno di una soletta apre nuove prospettive per la realizzazione di calzature intelligenti, utili sia in ambito sportivo, per il monitoraggio del benessere degli atleti, sia in ambito lavorativo, per la tutela della salute dei lavoratori.

Le principali sfide affrontate hanno riguardato la progettazione di un sistema in grado di acquisire campioni in modo affidabile e tempestivo, nonché l'individuazione dei punti più idonei per la misurazione sulla pianta del piede. La complessità anatomica di quest'area, dovuta alla presenza di numerosi muscoli, vene, vasi sanguigni e tendini, rende particolarmente difficile l'acquisizione di segnali PPG (Photoplethysmographic). Per questo motivo, l'attenzione è stata rivolta in particolare al lato della pianta del piede, una zona per la quale non si riscontrano precedenti studi in letteratura.

Il lavoro ha previsto la prototipazione del sistema di misura, lo sviluppo della logica di acquisizione e raccolta dei segnali, la verifica del codice implementato e l'acquisizione sperimentale di forme d'onda da diversi soggetti, al fine di condurre uno studio preliminare sulla fattibilità della misurazione.

Abstract

This thesis presents the development of an acquisition system for measuring heart rate signals from the sole of the foot. Heart rate monitoring from this area represents a largely unexplored field of research, yet one with significant potential. In particular, the integration of such a system into an insole opens the door to innovative smart footwear, applicable in sports for athlete monitoring as well as in occupational settings for worker health assessment.

The main challenges addressed include the design of a system capable of reliably and promptly acquiring signal samples, along with the identification of suitable measurement sites on the foot. The anatomical complexity of the sole—due to the presence of numerous muscles, veins, blood vessels, and tendons—makes the acquisition of PPG (Photoplethysmographic) signals particularly demanding. For this reason, the focus was placed on the side of the foot sole, a region for which no prior measurement attempts were found in the literature.

The work involved the prototyping of the measurement system, the development of acquisition and data collection logic, validation of the implemented code, and experimental signal acquisition from multiple subjects to conduct a preliminary feasibility study.

Contents

1	Introduction	1
1.1	Wearable devices	1
1.2	Optical sensors	2
1.3	Photoplethysmography	2
1.3.1	Operating principle	2
1.4	Foot anatomy	4
2	The Measurement System	7
2.1	Adafruit Feather nRF52840 Sense	7
2.2	SparkFun Pulse Oximeter and Heart Rate Sensor	8
2.2.1	The MAX30101	8
2.2.2	The MAX32664	10
2.3	The MAX30102	12
2.4	Final Prototype	13
3	Software design and analysis	17
3.1	Programming Language Selection	17
3.2	Tools	17
3.3	Data Acquisition	19
3.4	User interface	22
3.5	Data Acquisition Logic	25
3.5.1	Polling-Based Methods	26
3.6	Testing of the Code	28
3.6.1	Methodology	29
3.6.2	Results and Discussion	29
3.6.3	Power consumption	31
3.6.4	Sample Reliability	33
3.6.5	fetchData() Function	36

3.6.6	sendData() Function	38
3.6.7	Implementation of the sendData() Function	39
4	Python client	43
4.1	Main Program Execution	43
4.2	Acquisition from serial port	44
4.3	Future Enhancements	45
5	System testing	47
5.1	Measurement setup	47
5.2	Waveforms analysis	49
6	Conclusions	55
	Bibliography	57

List of Figures

1.1	Hemoglobin absorbance spectra	4
1.2	Peripheral vascularization of the foot [8]	5
2.1	The Adafruit Feather nRF52840 Sense board	7
2.2	The SparkFun Pulse Oximeter and Heart Rate Sensor	8
2.3	Schematic of the MAX30101	9
2.4	Channel slot timing for multi-LED mode	11
2.5	Pulse width setting and timing information	11
2.6	Tables showing the available settings for the sensor, left table for red+IR LEDs and right table for only red LED	12
2.7	Application circuit for the MAX32664	13
2.8	MAX30102 front and back view	14
2.9	MAX30102 package internal circuit	15
2.10	Final prototype	15
3.1	Acquisition system logical scheme	19
3.2	<code>fetchData()</code> alternatives	21
3.3	Acquisition system alternatives	22
3.4	Complete loop cycle in case of new data found in the sensor's FIFO	32
3.5	Complete loop cycle in case of no data found in the sensor's FIFO for the check logic ($t_{loop_data_check}$)	33
3.6	Times required by the I2C to start	34
3.7	Logic transition times of the I2C, $t_{SU,DAT}$ is being directly measured by the oscilloscope	35
3.8	Power consumption measured with an oscilloscope. The blue line is the voltage on the resistor while the pink line is the interrupt pin showing the duration of a loop cycle	36
3.9	System power consumption measured with the power profiler kit	37
5.1	Acquisition setup of a subject	49

5.2	Signal from the finger before filtering	50
5.3	Signal from the finger after filtering	51
5.4	Filtered waveform of a heartbeat from the foot	52
5.5	Filtered waveform from the foot not exhibiting a heartbeat	53
5.6	Lateral acquisition from the foot	53
5.7	Heart rate acquired from the side of the foot	54

Chapter 1

Introduction

1.1 Wearable devices

The rapid advancements in ubiquitous sensor technologies are significantly transforming the landscape of Internet of Things (IoT) applications. These sensors enable seamless and wireless data collection, paving the way for a future where healthcare services are not bound by time or location constraints and can meet virtually any need. Beyond personal healthcare, these innovations are driving progress in domains such as home automation, safety, and convenience. Moreover, they play a pivotal role in large-scale endeavors, including the development of smart cities, advancements in smart agriculture, and the creation of energy-efficient industrial systems. Their applications extend to critical areas like water conservation, supply chain optimization, transportation, and logistics [1].

Among these advancements, wearable devices have emerged as transformative tools in delivering health and safety solutions. Recent technological breakthroughs have enabled the design of sensors that are smaller, more intelligent, and, crucially, more cost-effective. This has spurred a surge in the popularity of wearable technologies as their decreasing costs are matched by increasing reliability and efficiency. Wearable devices are being utilized in diverse applications, ranging from monitoring water pollution through sensors embedded in divers' suits to tracking glucose levels via skin patches. These devices are both flexible and non-invasive, offering innovative approaches to everyday challenges. Biosensors, in particular, have emerged as a cornerstone in sensor development due to their ability to analyze atmospheric and biological data. This category encompasses a wide array of devices, from electronic noses to DNA-functionalized sensors capable of detecting viruses, which are revolutionizing public health and point-of-care testing. With the advent of smartwatches that detect irregularities in heart rhythms or instances of sleep apnea, constant manual health checkups may soon become a thing of the past.

1.2 Optical sensors

Optical sensors form the foundation of contemporary sensing technologies, leveraging the properties of light to measure and analyze physical, chemical, or environmental conditions. These sensors operate by converting light—whether it is transmitted, reflected, absorbed, or emitted—into electrical signals for measurement. They are highly valued for their precision, their ability to operate without physical contact, and their functionality in challenging environments, such as extreme temperatures or corrosive atmospheres. Optical sensors come in many forms, including photodiodes, phototransistors, and fiber-optic sensors, and are applied across various industries, such as manufacturing, healthcare, environmental monitoring, and consumer electronics.

A defining feature of optical sensors is their sensitivity to light across a wide spectrum, enabling them to detect even the most subtle changes in intensity, wavelength, or polarization. They are versatile, capable of not only measuring light but also detecting motion, distance, and material properties. The scalability of optical sensors allows for their incorporation into devices as small as smartphones and systems as large as industrial automation equipment.

1.3 Photoplethysmography

Among the numerous applications of optical sensing technology, photoplethysmography (PPG) sensors stand out as a critical innovation, especially in the healthcare sector. In 2007, the World Health Organization recognized the pulse oximeter as an essential device for reducing operative complications and included it in its Surgical Safety Checklist [2]. Photoplethysmography, a non-invasive optical method, enables the measurement of various physiological parameters by producing waveforms that represent circulatory volume changes in skin tissue. First described in the 1930s, this technique laid the foundation for significant advancements in medical monitoring systems. The Hewlett-Packard 47201A ear oximeter, the first commercially available continuous arterial oxyhemoglobin saturation monitor, was developed in the early 1970s. However, early models were cumbersome and required heating the earlobe, which could lead to burns. This issue was resolved with the advent of pulse oximetry, an innovative technique introduced by Dr. Takuo Aoyagi of Nihon Kohden in 1972, which was commercialized by the mid-1970s. Today, the modern pulse oximeter represents the most prominent clinical application of PPG, benefiting from further advancements made in 1991 [3].

1.3.1 Operating principle

PPG sensors operate by using light to measure variations in blood volume within tissue, offering a non-invasive method to monitor heart rate, oxygen saturation, and other physiological

parameters. By analyzing the interaction between light absorption and blood flow, PPG sensors provide accurate and unobtrusive measurements of vital signs, ensuring minimal disruption to the individual. These sensors rely on principles such as the Lambert-Beer law to detect blood flow dynamics.

$$I = I_0 \cdot e^{-\varepsilon \cdot c \cdot l} \quad (1.1)$$

The relationship between light absorption and solution concentration is mathematically expressed as follows:

- l represents the geometrical path length of the light through the medium (measured in cm),
- ε denotes the molar absorptivity or extinction coefficient (expressed in $L \cdot mol^{-1} \cdot cm^{-1}$),
- c signifies the concentration of the analyte within the solution ($mol \cdot L^{-1}$),
- I_0 refers to the source intensity of the light emitted by the LED,
- I represents the intensity of the light detected after passing through the medium.

This formulation enables the quantification of analyte concentration by correlating light absorption to the solution's properties. By carefully selecting the wavelength of the incident light, it is possible to determine the concentration of a specific analyte. For a PPG sensor, it is sufficient to detect variations in the analyte concentration, which manifest as changes in the intensity of light measured by the photodiode, provided the optical path length remains constant.

PPG sensors utilize light across various wavelengths, including red ($620 \div 750 \text{ nm}$), infrared ($780 \text{ nm} \div 1 \text{ mm}$), green ($495 \div 570 \text{ nm}$), and blue ($450 \div 495 \text{ nm}$). Each wavelength interacts distinctly with oxygenated and deoxygenated hemoglobin due to their differing molar absorption coefficients. This variability forms the basis of PPG functionality. Figure 1.1 illustrates these differences in molar absorption coefficients.

PPG sensors are classified into two primary categories based on their form factor: transmission-type and reflection-type. Transmission-type PPG sensors emit light on one side of the measurement site and detect the light that passes through the body part, such as a finger. These sensors are widely used in clinical settings due to their ability to provide reliable measurements. This reliability stems from the light passing through veins and arteries, allowing for enhanced detection accuracy. However, their main limitation is a low signal-to-noise ratio (SNR), as other tissues like bones, skin, and fat can interfere with the signal.

In contrast, reflection-type PPG sensors detect light on the same side as its emission. This design results in a higher SNR but makes these sensors more susceptible to motion artifacts.

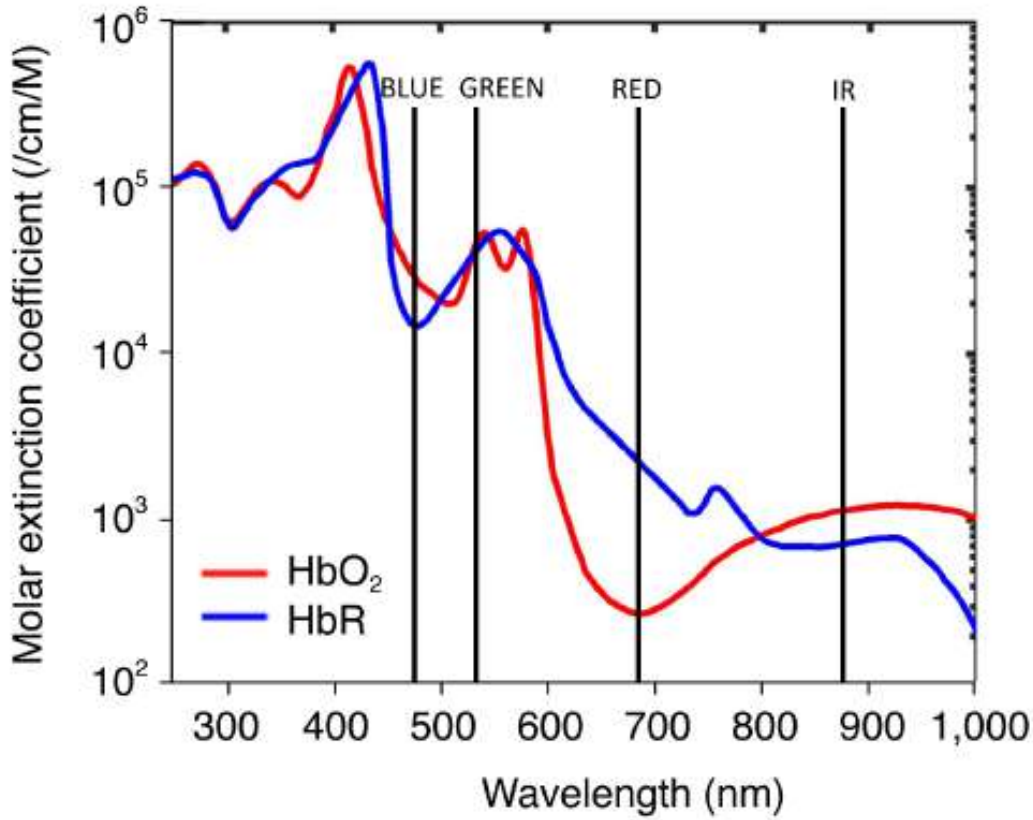


Figure 1.1: Hemoglobin absorbance spectra

The choice between these two types typically depends on the location of the sensor application. Transmission-type sensors are suited for thinner parts of the body, while reflection-type sensors are preferred for areas where the body is too thick for light to pass through [4][5].

1.4 Foot anatomy

The feet serve as a notable example of an application site for PPG sensors, providing both accessibility and unique circulatory properties. This makes them highly effective for acquiring accurate data in clinical and sports-related scenarios, such as smart shoes designed for monitoring diabetic patients [6]. Interestingly, the foot is sometimes referred to as the "second heart," as it aids blood return to the heart through two structures: the superficial "sole of Lejars" and the internal "sole of Bourgeret." These anatomical features facilitate upward blood flow by leveraging the rhythmic compression of the sole during movement.

The foot's vascular system, designed to maintain healthy circulation, is divided into two regions: the dorsal (top) and plantar (sole) sides. On the dorsal side, blood is delivered primarily by the arcuate artery and the dorsal metatarsal artery, which branch from the larger dorsal artery

of the foot. These main arteries create a network of smaller branches, ensuring the surrounding tissues are well-supplied with blood. Each toe receives nourishment from smaller vessels along its sides, and the dorsal system is interconnected with the sole's vessels to maintain consistent blood flow.

The plantar side of the foot is supported by the plantar arch, a central vessel located close to the bones. This arch extends across the sole and branches into four main arteries, which run between the toes and split into smaller arteries to supply blood to both sides of each toe. Dedicated vessels also support the big toe and the smallest toe. Connections between the plantar and dorsal vessels ensure efficient circulation throughout the foot [7].

Particularly, the inner side of the foot, below the ankle, offers an excellent site for studying PPG signals. This area is densely vascularized and experiences reduced motion during walking, minimizing the impact of movement artifacts, hopefully. Moreover, the presence of the great saphenous vein ensures substantial blood flow, enhancing the signal quality for PPG applications (Figure 1.2).

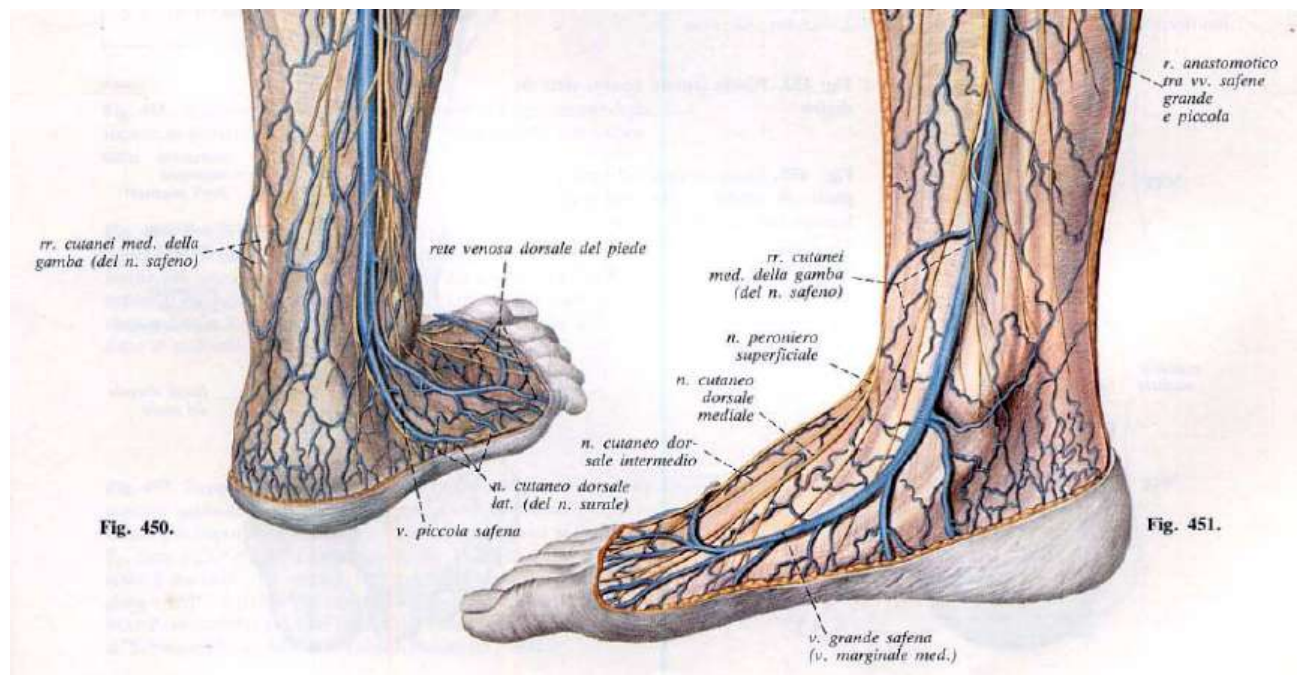


Figure 1.2: Peripheral vascularization of the foot [8]

Chapter 2

The Measurement System

The measurement system was developed with the goal of acquiring heart rate data from the sole of the foot. With this in mind, it was realized using a microcontroller board and a photoplethysmography (PPG) sensor. The board used was the Adafruit Feather nRF52840 Sense, which served to acquire data from the sensor and communicate with the computer's serial bus. Two possible sensors were studied, one was the SparkFun Pulse Oximeter and Heart Rate Sensor which is composed by the MAX30101 PPG sensor and the MAX32664 microcontroller, the other was the MAX30102, a PPG sensor without an integrated microcontroller in the package.

2.1 Adafruit Feather nRF52840 Sense

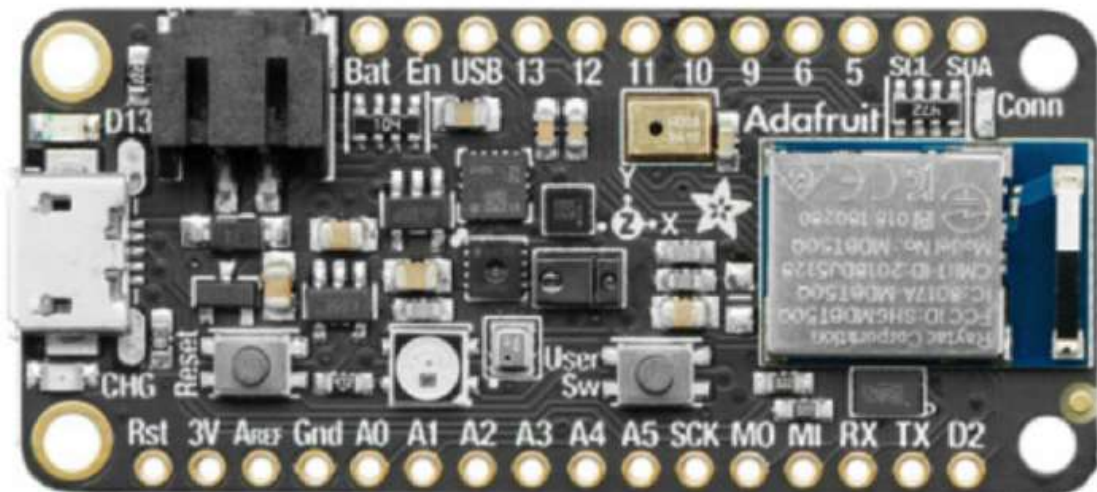


Figure 2.1: The Adafruit Feather nRF52840 Sense board

The Adafruit Feather nRF52840 Sense (Figure 2.1) is a versatile microcontroller board pow-

ered by an ARM Cortex M4F processor running at 64 MHz. It features 1MB flash memory and 256KB SRAM, which is significantly larger than typical Arduino boards (the ATmega2560 has 264KB flash memory and 8KB SRAM). It includes a native USB stack with a UF2 bootloader for easy programming using CircuitPython and Bluetooth Low Energy. The board also comes equipped with a variety of environmental and motion sensors, such as an accelerometer and gyroscope (LSM6DS3TR-C), a proximity, light, color, and gesture sensor (APDS9960), and a temperature and barometric sensor (BMP280). The board is lightweight and compact, weighing 6g and measuring 5.08 cm x 2.29 cm x 0.71 cm, making it ideal for space-constrained solutions. It has a built-in attachment for a 3.7V LIPO battery and 21 GPIO pins [9]. Among these, the 3.3V reference voltage and the ground pin were used to power the sensor, while the two I2C pins were used for communication between the sensor and the board. All communication between the board and the computer was facilitated through serial communication using a USB micro cable.

2.2 SparkFun Pulse Oximeter and Heart Rate Sensor

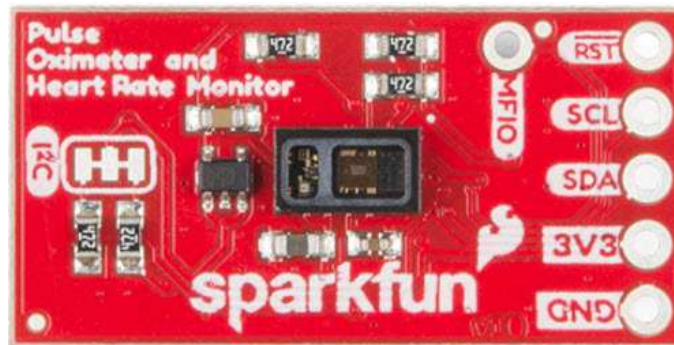


Figure 2.2: The SparkFun Pulse Oximeter and Heart Rate Sensor

The SparkFun Pulse Oximeter and Heart Rate Sensor is a compact, I2C-based biometric sensor designed to measure heart rate and SpO₂. It utilizes the MAX30101 pulse oximeter and heart rate sensor, which employs photoplethysmography (PPG) to detect light absorption by the blood. Additionally, it features the MAX32664 biometric sensor hub, a small M4 Cortex processor pre-programmed with all necessary algorithms for heart rate and SpO₂ measurement, including calculation, filtering, and automatic gain control.

2.2.1 The MAX30101

Produced by Maxim Integrated (now Analog Integrated since its acquisition in 2021 [10]), the MAX30101 has been available since 2018 and is the first in the MAX3010X series. These

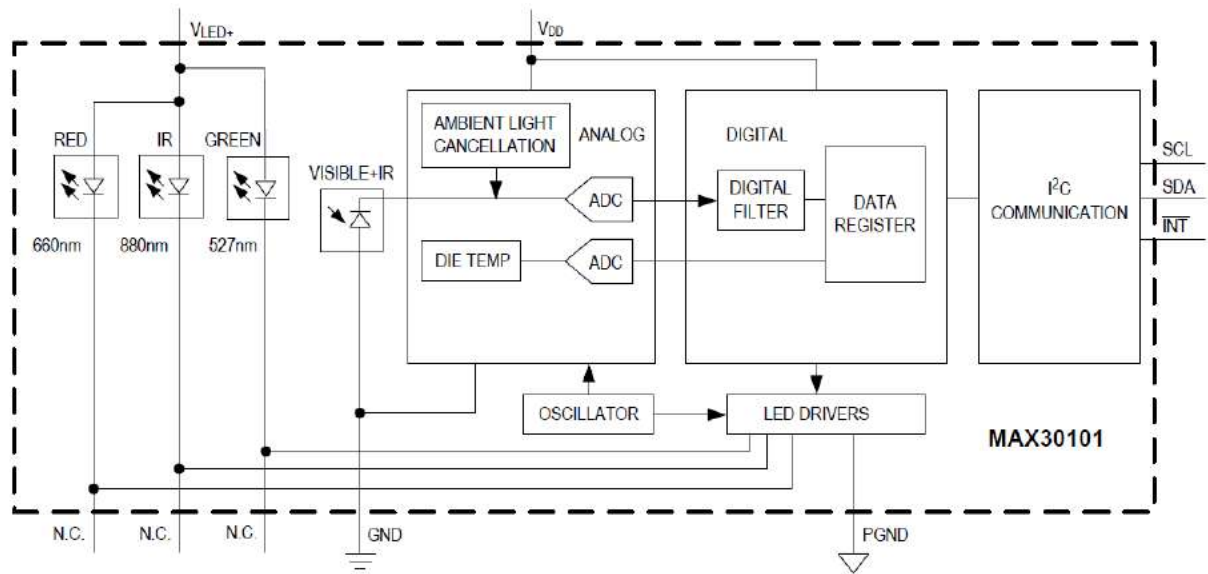


Figure 2.3: Schematic of the MAX30101

sensors are designed to provide accurate measurements of heart rate and SpO₂ using the photoplethysmography technique, except for the MAX30105, which is intended for use as a particle sensor in smoke detectors. Measuring only 5.6 mm x 3.3 mm x 1.55 mm, the MAX30101 is a small and compact sensor ideal for fitness, smartwatch, and other wearable biomedical applications. It can operate in a wide range of external temperatures (-40°C to +85°C). The sensor is powered by a single 3.3V pin and a ground pin and communicates via the two I²C pins SDA and SCL. While the SCL pin shares a common clock signal between the master (e.g., the board) and the slave (e.g., the sensor), the SDA pin handles all communication, synchronized by the common clock signal. Although the sensor features two distinct pins (V_{LED+} and V_{DD} in Figure 2.3), which supply power to the LEDs and the rest of the sensor, respectively, its implementation in the SparkFun Pulse Oximeter, as shown in Figure 2.2, eliminates the use of V_{LED+} , leaving the user with the option to power only the entire sensor through a single supply.

The sensor has three different LEDs: green (λ 537 nm), red (λ 660 nm), and infrared (λ 880 nm) (see Figure 2.3). Utilizing one of these different wavelengths, it is possible to accurately determine the heart rate of the subject. It should be noted, however, that the possible light combinations are limited. The available configurations are: 1 = Red only, 2 = Red + IR, and 3 = Red + IR + Green. The library provided with the sensor [11] allows for direct enabling and disabling of individual LEDs, but the potential consequences of doing so are not disclosed. Other useful programmable parameters include the ADC scale range, LED power supply voltage, the number of samples averaged per measurement, sample rate, and LED pulse width.

The ADC has clock frequency of 10.24 MHz and is realized with a continuous-time oversampling sigma-delta converter with 18-bit resolution. The programmable full-scale range varies

from 2 μA to 16 μA , this translates to a sensitivity of 7.81 pA per LSB for the 2048 nA range and a sensitivity of 62.5 pA per LSB for the 16384 nA range, as explained in the datasheet [12].

The power level regulates the power supplied to the LEDs and influences presence detection's range, a useful feature for developing a system to monitor a subject's health that can turn on and off based on the presence of the subject. Presence detection is done making use of the difference in the intensity of the light measured from the photodiode, samples without a finger present are in the order of the hundreds in a well lit environment while they become in the order of the tens of thousands when an object is placed in front of the sensor. Actual implementation of presence detection, such as setting thresholds, interpreting the data and triggering specific actions, needs to be manually coded [13]. However, this feature was not explored in this master thesis.

Sample rate and pulse width are mutually dependent parameters; selecting a high sample rate and a large pulse width would result in a total time exceeding a second, rendering it impractical. Consider, for example, to select a pulse width of $411\mu\text{s}$ and a sample rate of 3200 Hz, this results in a duty cycle of 1.3 ($411\mu\text{s} \cdot 3200\text{ Hz}$), therefore greater than 1, which is absolutely illogical. Available sample rates are 50, 200, 400, 800, 1000, 1600, and 3200 sps (samples per second), while available pulse widths are 69, 118, 225, and 411 μs . For each pulse width, there is a corresponding resolution of the sample acquired from the ADC, ranging from 15 bits for the 69 μs pulse width to 18 bits for the 411 μs pulse width.

The number of active LEDs is also an important variable that affects the combinations of sample rates and pulse widths, as the board must take a measurement for each LED with the same pulse width and sample rate for the other LEDs. The sensor keeps one led on for the duration specified by the pulse width, then waits for an amount of time dependent on the pulse width and then turns on the next LED. In this way, the sensor can measure the amount of reflected light for each LED without being influenced by the previous one. An example is shown in figure 2.4 which shows the timing for a measurement with sample rate of 1000 sps, pulse width of $69\mu\text{s}$ and red+IR LEDs active. Figure 2.5 contains a table that reports the wait time between LEDs turn on depending on the pulse width selected. The interconnection between sample rate, available pulse width and resulting number of bits for each measurement is summarized in Figure 2.6.

2.2.2 The MAX32664

The MAX32664 is a pre-programmed microcontroller with firmware drivers and algorithms for heart rate estimation and power management. Combined with the appropriate sensor devices, the MAX32664 acts as a sensor hub to provide processed data to a host device. This solution seamlessly enables customers to receive raw and/or calculated data from Maxim's optical sensor solutions. The tiny form factor (1.6 mm x 1.6 mm 16-bump WLP) allows for integration into

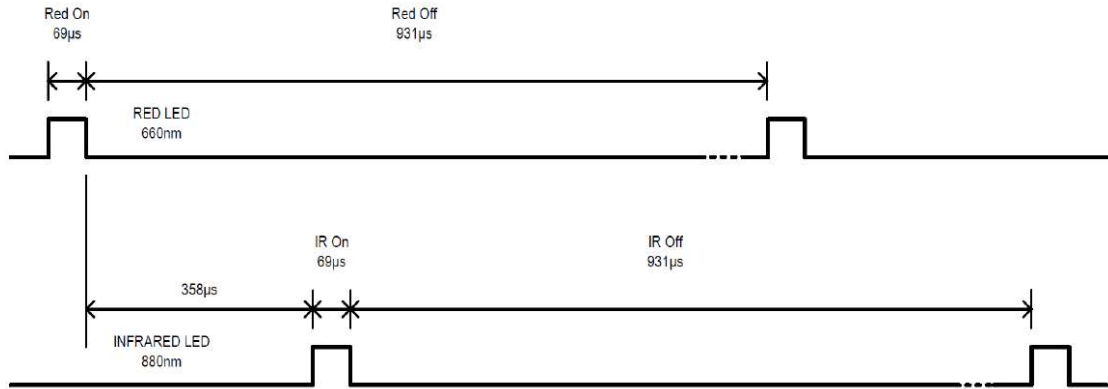


Figure 2.4: Channel slot timing for multi-LED mode

PULSE-WIDTH SETTING (μs)	CHANNEL SLOT TIMING (TIMING PERIOD BETWEEN PULSES) (μs)	CHANNEL-CHANNEL TIMING (RISING EDGE-TO-RISING EDGE) (μs)
69	358	427
118	407	525
215	505	720
411	696	1107

Figure 2.5: Pulse width setting and timing information

extremely small applications. The MAX32664 provides a fast-mode I2C slave interface to a microcontroller host. A second I2C interface is dedicated to communicating with sensors. These two different I2C connections prevent direct communication with the MAX30101 PPG sensor, necessitating the use of the MAX32664 as an intermediary as illustrated in Figure 2.7, where it is clear that the two I2C connections are separate.

In this thesis, the aim was to develop a system that directly acquires the PPG signal from the sole of the foot without previous processing. The integrated algorithms in the sensor were then deemed unnecessary, so we decided to bypass them. To deactivate all the processing done by the microcontroller the functions ‘agcAlgoControl’ and ‘maximFastAlgoControl’ were used [11]. The Automatic Gain Control algorithm regulates the the gain of the sensor to ensure the signals are strong enough to be analyzed as well as not too strong to avoid distortions or saturations. The MAXIM FAST algorithm instead is a proprietary technology of Maxim integrated which elaborates Biometric data to provide fast and accurate estimations of heart rate and SpO2 measurements. Disabling the algorithms allowed us to retrieve the raw PPG data, achieving our objective. To retrieve data from the FIFO the method `readSensorBpm()` was used, which returns a `bioHUB` object which contains the values of the red and, eventually, the IR LED. This code was then tested placing a finger on the sensor with the arm in a resting position and applying a small constant pressure. Unfortunately, during the testing phase, it was discovered that the

SAMPLES PER SECOND	PULSE WIDTH (μ s)			
	69	118	215	411
50	○	○	○	○
100	○	○	○	○
200	○	○	○	○
400	○	○	○	○
800	○	○	○	
1000	○	○		
1600	○			
3200				
Resolution (bits)	15	16	17	18

SAMPLES PER SECOND	PULSE WIDTH (μ s)			
	69	118	215	411
50	○	○	○	○
100	○	○	○	○
200	○	○	○	○
400	○	○	○	○
800	○	○	○	○
1000	○	○	○	○
1600	○	○	○	
3200	○			
Resolution (bits)	15	16	17	18

Figure 2.6: Tables showing the available settings for the sensor, left table for red+IR LEDs and right table for only red LED

MAX32664 still had subroutines working even when the algorithms were deactivated. This was deduced from the fact that the sensor reset the sampling rate every time it detected the presence of a human finger and even the slightest movement seemed to trigger this effect.

To overcome this problem and guaranteeing a full control of the sensor settings, we decided to connect directly a PPG sensor, in order to collect raw data under well-controlled configurations.

2.3 The MAX30102

The main difference between the MAX30102 and the MAX30101 is the absence of the green LED from this sensor. As explained in the introduction, the green LED is only one of the possible wavelengths used to estimate the heart rate so this absence did not affect our project. The other difference between the MAX30101 and the MAX30102 are two reserved registers, specifically the 0x0E and 0x0F registers. These registers are used by the MAX30101 to set the current level for the green LED, the reason why the green LED has two slots is unclear as it was not disclosed by the manufacturer. The sensor comes in a package that handles the internal voltage reference when powered and delivers the right voltage for the sensor as well as the voltage bias for the LEDs, full circuit of the package is shown in figure 2.9. The circuit accepts an input voltage, usually between 3.3 and 5.5V, and adjusts it to properly power the sensor. This adjustment is done through components such as voltage regulators, which convert the voltage to the necessary amount, typically 3.3V. Additionally, the circuit includes capacitors to stabilize the voltage and prevent unwanted fluctuations, as well as resistors to limit the current or divide the voltage where necessary. Finally, there are connection pins, which allow the sensor to be connected to an external system for communication and power, to exchange data with other devices. When

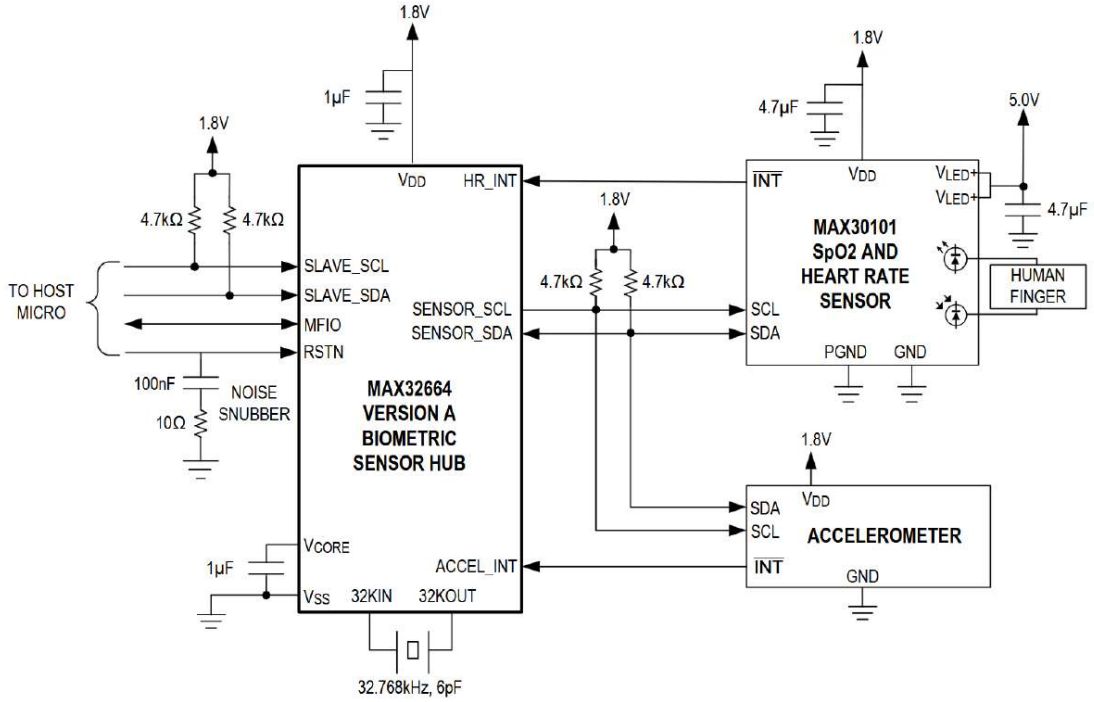


Figure 2.7: Application circuit for the MAX32664

tested as the previous one, this sensor returned data from the measured light intensity without changing the parameters set before the acquisition.

2.4 Final Prototype

The measurement system developed in the end comprised the Adafruit Feather nRF52840 Sense board and the MAX30102 PPG sensor connected by I2C bus. Following the recommendations in the MAX30102 datasheet for sensor implementation, a $1\mu F$ filter capacitor was placed between the LEDs driver pins and the ground pin to prevent fluctuations in the power delivered to the LEDs. Specifically, the capacitor was soldered to the sensor pins and is located on the underside of the MAX30102 package. The capacitor under the sensor created stability problems for resting a finger on solid surfaces. On the insole, thanks to its soft material, the capacitor was absorbed by the foam and did not interfere with measurements. Another suggestion in the datasheet was to place pull-up resistors on the I2C connections. However, considering the encumbrance generated by the capacitor, we decided to postpone the addition of the resistors unless communication issues arose. I2C signal levels were tested in the lab with a DSO (Digital Signal Oscilloscope) to ensure correct voltage levels. Since no issues were observed during testing, the resistors were never implemented, keeping the system as compact as possible. The sensor was powered through the 3V pin on the board, which delivered 3.3V to the Vin pin of the

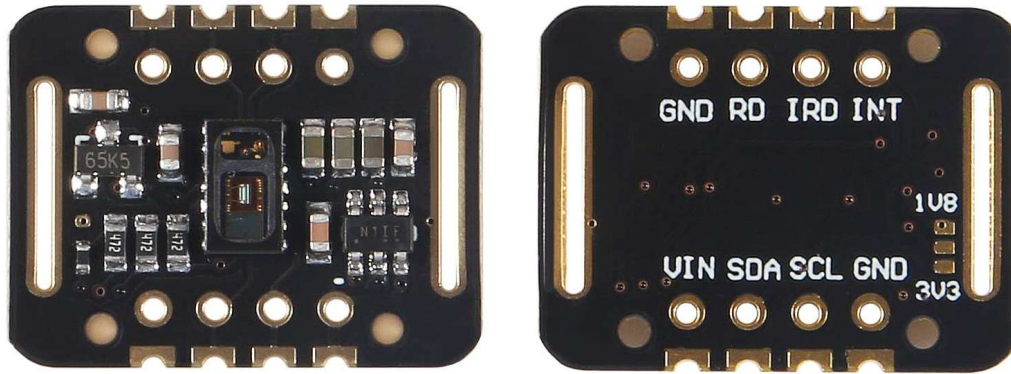


Figure 2.8: MAX30102 front and back view

sensor's package and the ground pin was connected to the two ground pins on the package. The two sensor pins are for the power supply ground and the analog ground for the LEDs drivers. Due to the sensor's package design, it was impossible to distinguish between the two, so they were connected together, creating a single ground connection.

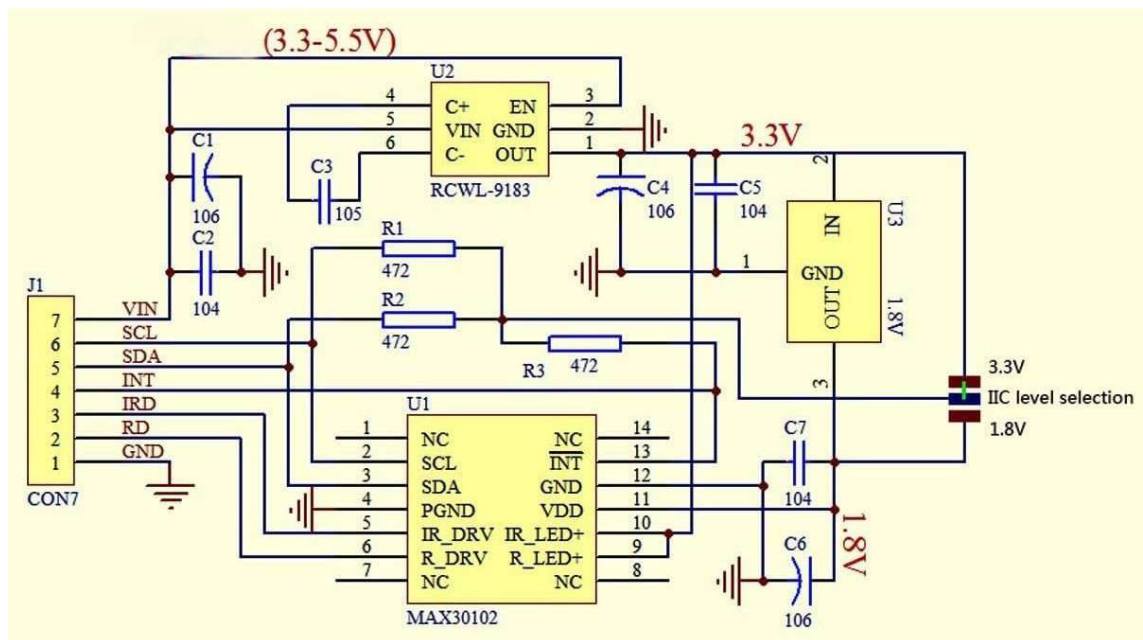


Figure 2.9: MAX30102 package internal circuit

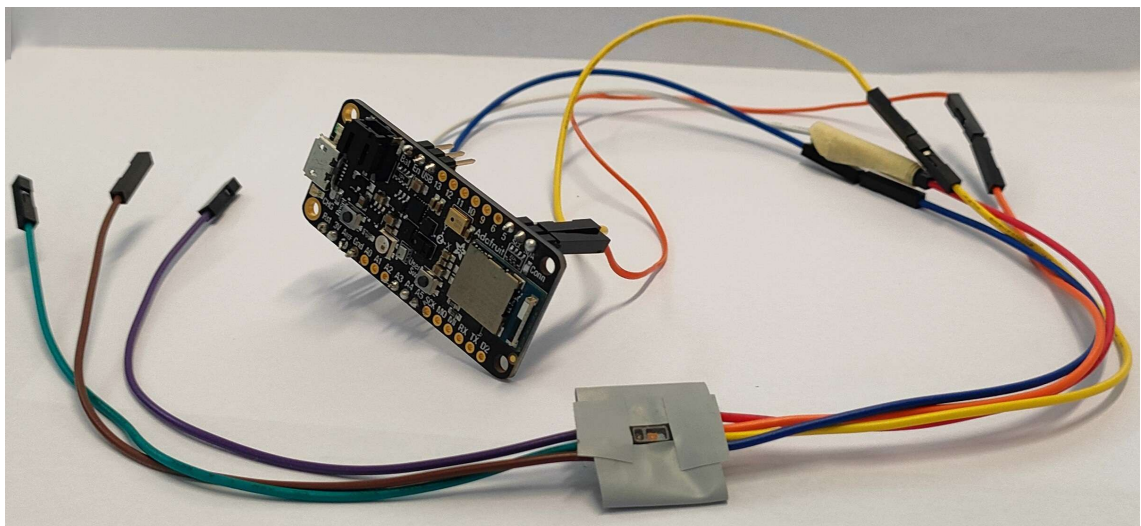


Figure 2.10: Final prototype

Chapter 3

Software design and analysis

3.1 Programming Language Selection

The programming of the Adafruit board could be implemented using two supported languages: Arduino or CircuitPython. The selection was informed by prior work conducted on the same board [14]. Arduino, being a compiled language that blends C and C++, offers significantly faster execution speeds. This feature was essential for achieving the primary objective: to guarantee reliable and uninterrupted code performance.

3.2 Tools

Before writing the code to communicate with the sensor, various preliminary components (software and development environment) had to be installed. The first step involved installing the Arduino IDE (Integrated development environment), it's the software platform where one can write, compile, and upload code to Arduino boards. It provides an interface to edit sketches (Arduino programs), a compiler to turn code into machine-readable instructions, and a tool to upload the code to the microcontroller. The program can be downloaded from the Arduino main site [15] by going to the software page and selecting the installer for the operating system of the machine in which the user wants to install the program. The version used was the Arduino IDE 2.3.4, after having installed the development environment it was necessary to install a board manager for the Adafruit Feather nRF52840 Sense. This can be done by going to the board manager tab on the left side of the Arduino IDE and searching for the name of the board, in this case the board manager is called 'Adafruit nRF52' which handles all the Adafruit feather boards. The board manager is a tool of the Arduino IDE that connects to a remote repository in which board definitions are stored. The information stored are microcontroller specs, pin configurations, drivers and, eventually, additional software. This allow the Arduino IDE to recognize

and establish communication with a board. The board manager is responsible for compiling the Arduino code so that it can execute on the board's specific architecture. The Adafruit board is based on the ARM Cortex-M4F architecture, which differs from the classical Arduino boards. This difference necessitated the installation of the specific board manager to ensure compatibility. Although the Arduino Nano 33 BLE utilizes the same processor, and thus shares the same architecture, small differences in implementation could have potentially caused functionality issues. For this reason, it was deemed preferable to use the dedicated compiler available for the Adafruit board.

Sensor-specific libraries were installed. Library installation in the Arduino IDE can be done by either going to the library manager tab on the left side of the Arduino IDE and searching for the name of the library that the user wants to install. Or it can be installed by downloading the library from a remote repository (e.g. GitHub) where the code of the library is provided and placing the folder containing the library in the Library folder in the Arduino folder that was automatically created after the installation of the IDE. When the user wants to use a specific library, it just needs to add a `#include`, followed by the name of the file in which the library is written, at the top of the sketch. Arduino libraries are often made by two files, one in C and one in C++ programming languages. Usually the library in C is just a library that defines and automatically calls the C++ file. The user then just needs to call the C library to also include the C++ one.

Methods of the library are called by creating an object specific for the class which the library includes and adding to it the name of the method (e.g. `bioHub.readBpm()`, in this case the `bioHub` is the object specific for the SparkFun Biosensor Hub Library and the method `readBpm()` is a method that returns the BPM of the subject).

The SparkFun Biosensor Hub Library [11], developed by SparkFun Electronics, was used to allow communication between the MAX32664 sensor hub and the board. This library provides the necessary functions to configure and acquire data from the sensor.

The MAX30102 sensor, on the other hand, was controlled using the SparkFun MAX3010x Pulse and Proximity Sensor Library [16]. This library supports a great variety of sensors in the MAX3010x series. However, due to its broad compatibility, careful consideration was required to avoid executing unsupported commands, such as activating the green LED, which is absent in the MAX30102 sensor. The consequences of issuing unsupported commands were not explicitly stated by the manufacturer, making caution essential. Once the required libraries were installed, the next step was developing the code for signal acquisition.

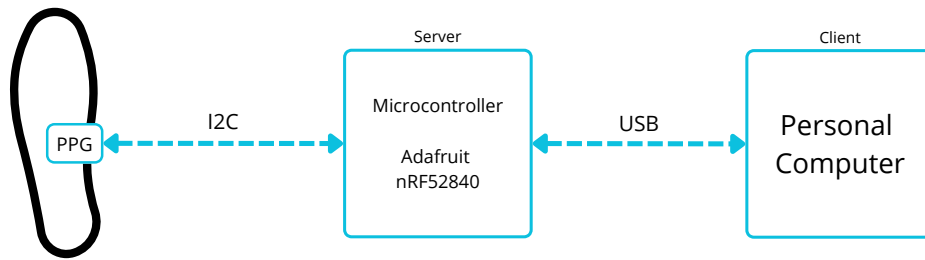


Figure 3.1: Acquisition system logical scheme

3.3 Data Acquisition

The logic for data acquisition was divided into two parts: (1) a server that retrieves samples from the sensor and transmits them to the PC via a serial interface and (2) a client that reads the serial port on the PC and stores the retrieved data in a file (Figure 3.1).

The server has the purpose to setup the PPG sensor to begin the acquisition as well as to collect data from the sensor which will be temporarily stored in the board's flash memory. The code can be divided in two main parts as in all Arduino programs, the setup function and the loop function. The setup is a function that is executed only one time when the program is started, as the name suggests, its purpose is to initialize the board and the other components connected to it. In this case the function has the objective to initialize the PPG sensor for the acquisition. When all parameters have been set, the function is finished and the program passes to the loop function. This section of the code was designed to reproduce a, sort of, user interface (UI). This decision was motivated by the fact that the sensor has multiple parameters that can be set that influence the acquisition. This solution avoids having to change hard-coded parameters when there is the need to change the sensor configuration. Without this solution, the code would have to be loaded every time it becomes necessary to change the sensor parameters. As it will be detailed more clearly later, the setup then asks through the serial port if the user wants to modify the preset parameters and, eventually, accepts available values for the parameters that need to be modified.

The loop function, differently from the setup, is continuously executed while the program is working. The objective is to continuously check the samples stored in the sensor FIFO and retrieve them when new are available. A second purpose is to send the acquired data from the sensor to the computer connected via USB which will store and elaborate the data. To achieve these objectives two different functions were developed, `fetchData()` and `sendData()`, which respectively collect and send data from the sensor and to the computer.

Two main strategies could be performed by the `fetchData()` function, it could either call the `sendData()` function for each new sample acquired from the sensor or it could accumulate them inside the Adafruit board flash memory and send them when full. When the board sends samples on the serial port the program is interrupted and needs to wait for the `print()` function to finish before continuing (see Figure 3.2 on the left side). This can require non negligible times, especially if one considers that each sample is made of 3 bytes plus an identifier. The identifier allows the client to differentiate the samples sent by the server (two bytes) and the end characters, employed to inform that the sample is finished (one byte). This would mean that each data sent through the serial port would require to send at least 6 bytes, considering only one new sample produced by the sensor when the function is called, and up to 184 bytes if the sensor had his FIFO full. This means that the program would be slow and this would limit the maximum sample rate.

The second approach instead would acquire samples in a timely manner but the amount of consecutive samples obtained would be limited by the flash memory of the Adafruit board (Figure 3.2 right side). Thankfully the board has 1MB of flash memory which can contain 50000 consecutive samples before needing to be emptied. This means 50 seconds of acquisition at 1KHz and up to almost 17 minutes of continuous samples obtained. The second approach was deemed the most suitable, as it would facilitate the validation of the sensor reliability in estimating the heart rate.

The `sendData()` function had a straightforward development after having decided how to treat the data inside the server. Since samples are accumulated inside the memory of the Adafruit board, the function needs only to empty the data array inside the board and reset its write pointer. Regarding the strategy which determines when to request data from the sensor, various strategies to recognize the presence of new data were tested and the one which proved to be the most reliable was selected.

A second program was developed to acquire data from the serial port of the pc. The client was developed assuming that it would always listen for new data sent from the Adafruit board and then store the samples received inside a csv file. This format was chosen to improve compatibility as this type of file can be read by multiple software and can be easily used by other programming languages (such as MATLAB for example), it is also compact and lightweight. One can think of the interactions between the Adafruit board and the pc as a server-client interaction, where the Adafruit board acts as a server sending packets of samples to the client (the pc). Usually this transaction happens after the client requests the packets but we can imagine that the pc is constantly requesting new data from the server.

Conformably to the client-server architecture, a common communication protocol must be established between the microcontroller and the pc. This is needed because we need to ensure that

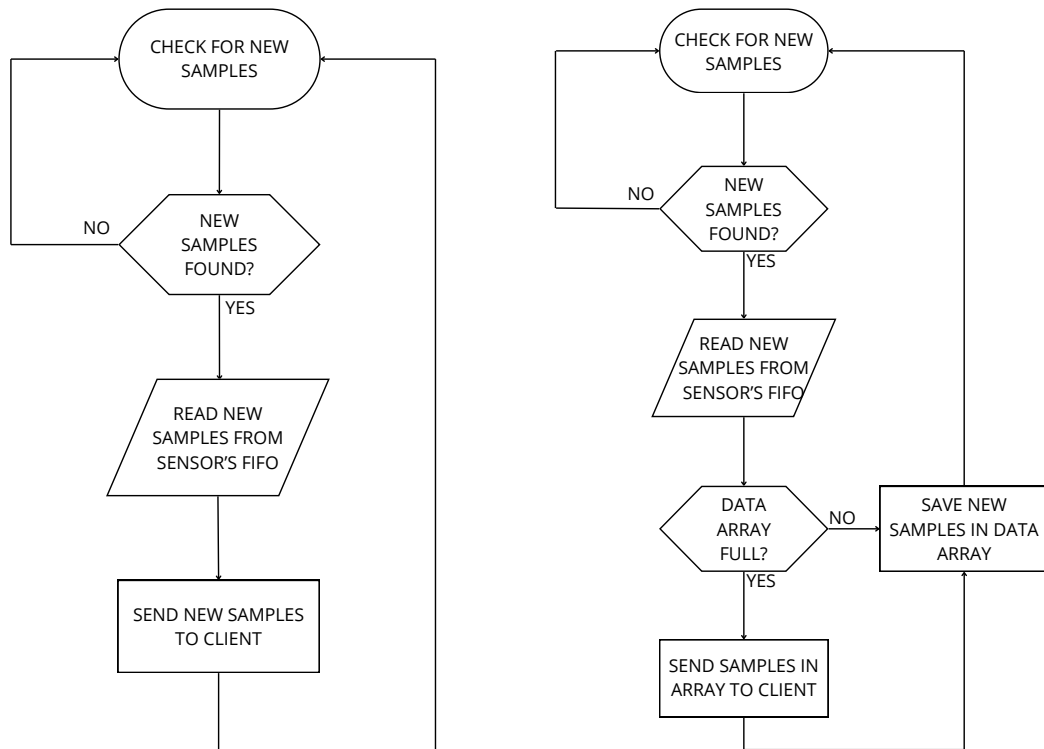


Figure 3.2: `fetchData()` alternatives

the pc correctly understands what the board is sending. The client doesn't know the settings of the sensor and in particular the number of LED active. To differentiate between samples obtained by the red LED and the infrared LED a communication standard was established. Before each sample an identifier letter is sent in the Serial port, "R" for the red LED samples and "I" for the infrared LED samples. In this way, the program in the pc can understand if a sample is from a LED or the other and store the samples correctly. This protocol is also needed as the program which collects data is different from the one used to initialize the sensor for the acquisition. Only one program can access the serial port at a time, this means that the user needs to first configure the sensor using the serial monitor in the Arduino IDE, then close the serial monitor and then collect data from the microcontroller.

Another way could have been to make the Data collector program in the pc communicate to the Adafruit board that it's ready to collect samples, then the microprocessor would start collecting data from the PPG sensor. In this way we could also have the microcontroller tell the data collector program which LED have been activated but this would require additional components in the loop function, slowing down the continuous check for new data possibly reducing the amount of samples collected at medium-high sample rates.

Alternatively, the Data collector program could be used to communicate with the sensor, using the serial port, and so the setup of the sensor and the data collection could be both handled by the python program. A visual example is reported in Figure 3.3, to be noted that the Arduino IDE is always needed to upload the sketch to the microcontroller board for each modification of the program. This second approach would be easier to use as it wouldn't require the user to switch from a program to another after the acquisition is started. However, implementing the ability of sending information to the microcontroller from the python program would require an additional loop in the python code to continuously check for response request by the Adafruit board. Additionally, the program would need to be able to differentiate the information request from the data sent through the USB port, increasing the time needed to convert the data from binary to ASCII code and decimal system. One could argue that, once the acquisition has started, the program could focus on only converting and storing the data from the USB port. Although true, the board could experience fluctuations in the power supply that, if low enough, could reset the program needing a new setup of the acquisition variables. Also, knowing that this design is aimed at an expert user who knows deeply the program and the components used, simplifications for an unexperienced users were not deemed necessary.

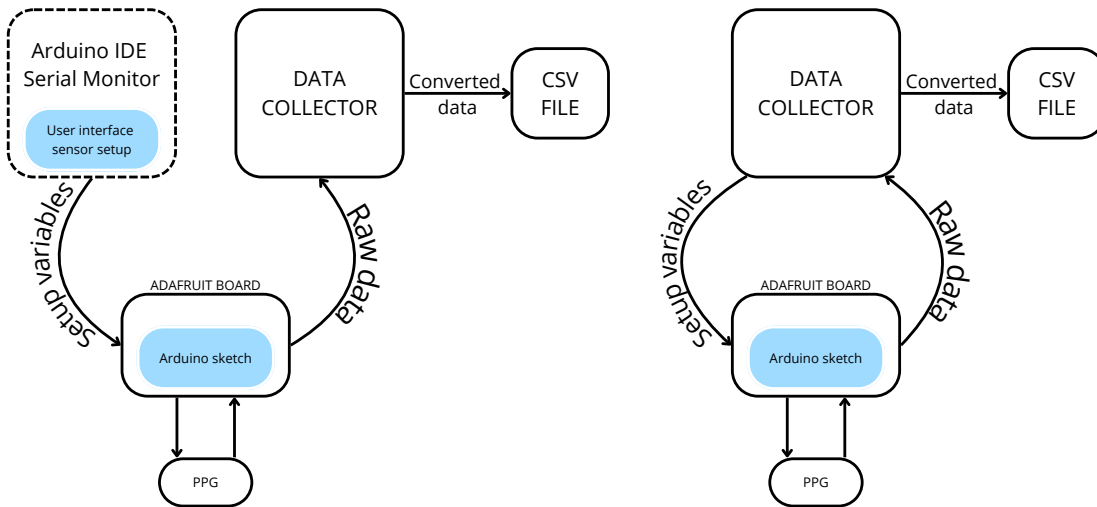


Figure 3.3: Acquisition system alternatives

3.4 User interface

Before beginning the measurement session, the sensor parameters must be properly configured to best suit the measurements the user wants to make. The adjustable parameters are the following:

- **Power Level:** This sets the current supplied to the LED, ranging from 0 mA (so the LED is always off) to 51 mA. The power level is programmed using a integer value between 0 and 255, corresponding to the minimum and maximum current. The stronger the power level the more depth can the light reach inside the skin. This can also become a problem because too much reflected light can saturate the ADC as well as increase the amount of light reflected by muscles and other tissues surrounding the veins.
- **Sample Average:** The number of samples averaged for each measurement. Available options are 1, 2, 4, 8, 16, or 32. If the number selected is greater than 1 (e.g. 4), then the sensor first collects the samples (so 4 samples as per the example before) and then does the average which will be saved in the sensor's FIFO. Averaging multiple samples can reduce noise in the collected signal.
- **LED Mode:** This determines the number of LEDs active during measurements, either 1 (red LED only) or 2 (both red and IR LEDs). For heart rate estimation only one light is sufficient while two are needed for SpO2 measurements.
- **Sample Rate:** The frequency of data collection, expressed in samples per second. Higher sample rates would produce more defined curves allowing a better appreciation of the evolution over time of the reflected light. That said, given the limited amount of samples collected consecutively before the need to interrupt the collection of samples to send the data to the client. This could cause artifacts in the resulting waveform since data collection could stop and restart in different moments as well as different ambient light generating fictitious peaks.
- **Pulse Width:** Defines the time period during which the LED remains active in each measurement cycle. Options include 69 μs , 118 μs , 215 μs , and 411 μs . Longer light emissions produce higher definition for the ADC, starting with 15 bits for 69 μs , up to 18 bits for 411 μs , increasing one bit for each longer emission time. This obviously comes with the drawback of reducing the maximum sample rate available, as the longer the emission time the longer is the time needed for each measurement taking a bigger portion of a second.
- **ADC Range:** The resolution and reference voltage of the sensor's ADC. This determines the smallest and largest signal amplitudes measurable, with each range linked to specific current values for the least significant bit (LSB). Specific values are shown in table 3.1.

The user interface, for configuring these parameters, proceeds to ask the user if it wants to change the value of each parameter one after the other. For example, the power level setup begins

ADC full scale [nA]	15 bits [pA]	16 bits [pA]	17 bits [pA]	18 bits [pA]
2048	62.5	31.25	15.63	7.81
4096	125	62.5	31.25	15.63
8192	250	125	62.5	31.25
16384	500	250	125	62.5

Table 3.1: ADC full scale range and corresponding value for LSB for each resolution

as follows: The code prompts the user with the default value, valid range and an explanation of the parameter's meaning.

```
Serial.println("Default power level is 0x1F=6.2mA, minimum value is 0x00
corressponding to 0mA, maximum value is 0xFF
corresponding to 51mA. Higher values will increase the detection range.
Would you like to modify it? (y or n)");
[...]
```

If the user opts to modify the value, it is required to input a valid number.

```
[...]
} else if (answer == "y") {
    Serial.println("Please provide a value fot the led Power level");
    while (true) {
        // Check if data is available to read
        if (Serial.available() > 0) {
            // Read the incoming data as a string
            String receivedString = Serial.readStringUntil('\n');
            [...]
        }
    }
}
```

Invalid inputs are caught and, in this case, the user is asked re-enter the value.

```
[...]
if (newLevel >= 0 && newLevel <= 255) {
    uint8_t newpowerLevel = (uint8_t)newLevel;
    powerLevel = newpowerLevel;
    break;
} else {
    Serial.println("Invalid power level entered. Please enter a
value between 0 and 255.");
[...]
```

Additional parameters, such as the Sample Rate and Pulse Width, include conditional checks to ensure logical and compatible settings. As specified in the sensor's datasheet, certain pulse widths restrict available sample rates based on the LED mode. Below is an example for the setup of the pulse width value:

```
[...]
if (ledMode = 1) {
    if (newWidth < 118) {
        pulseWidth = newWidth;
        break;
    } else if (newWidth > 69 && newWidth < 411) {
        //check before if the sample rate is too high otherwise do nothing
        if (sampleRate > 1600) {
            sampleRate = 1600;
            Serial.println("Sample rate set at 1600");
        }
        pulseWidth = newWidth;
        break;
    } else if [...]
```

As observed, for the configuration of the pulse width, which represents the final interconnected variable of the setup, an initial verification of the `ledMode` is performed. Subsequently, once a valid value has been received, an additional check is conducted on the `sampleRate`. If the selected `sampleRate` exceeds the permissible range, the highest available sample rate is automatically set, and the user is notified of this adjustment.

3.5 Data Acquisition Logic

The acquisition logic represents the critical part of the system, designed as fast as possible, for continuous collection of sensor data. Two primary approaches were initially explored: leveraging the hardware interrupt on the sensor or employing a polling method.

Using the hardware interrupt involved monitoring the state of the corresponding pin, which should change from high (3.3V) to low (0V) when an interrupt is issued or the FIFO is read. However, testing revealed that the interrupt pin's behavior was inconsistent, regardless of whether the FIFO almost full or data-ready interrupts were enabled. The code developed for the hardware interrupt relied on a system interrupt that would be called when the analog pin, set as an input and connected to the interrupt pin of the sensor, detected a change in the state of said

pin. The analog pin must be configured as an input to recognize external changes in the state of the sensor pin. When the interrupt occurred, the program would call the `fetchData()` function to acquire samples from the sensor. The system was left running for several minutes and no data was acquired since no changes in the state of the interrupt pin of the sensor was noticed using the oscilloscope with a probe connected to the interrupt pin of the sensor. Due to these issues, the interrupt-based approach was discarded, and focus shifted to polling methods.

3.5.1 Polling-Based Methods

Three distinct polling methods were examined:

1. Using the `check()` method from the MAX30102 library.
2. Using the **FIFO Almost Full Interrupt** logic.
3. Using the **Data Ready Interrupt** logic.

The `check()` Method

The `check()` method, from the SparkFun MAX3010x Pulse and Proximity Sensor Library [16], is specifically designed to poll the sensor for new data. Its implementation is explained in the following:

First, the method retrieves the `readPointer` and `writePointer` from the FIFO data register. These pointers contain the address in the FIFO of where the last unread sample is and where to write the next sample. If the pointers are unequal, then new data is available. The method calculates the number of unread samples by subtracting the `readPointer` from the `writePointer`, adjusting for wrap conditions if necessary.

Subsequently, the method calculates the total number of bytes to read, based on the number of active LEDs, and retrieves the data in chunks, respecting platform-specific I2C buffer constraints. The retrieved data is then converted into a 32-bit integer format and stored in the appropriate fields of the `sense` structure. The head pointer is updated and wrapped when needed. The `sense` structure is an object specific of the MAX3010x library which is used to temporarily store the samples acquired from the sensor in the board's RAM.

If additional LEDs (e.g. IR or Green) are active, the method processes and stores their data in the same manner. Finally, the method returns the number of newly acquired samples, enabling other parts of the program to collect the data which are stored in LED specific arrays (e.g. `sense.red` for the red LED).

To ensure continuous data acquisition, the `loop()` function repeatedly calls the `check()` method. The implementation of the `loop()` function is as follows:

```

void loop() {
newSamples=particleSensor.check();
if(newSamples!=0){
  for(int i=0; i<newSamples;i++){
    if(tail==array_size){
      sendData();
    }
    //the array saves the values obtained from the LEDs, one for each new sample;
    redVals[tail]=particleSensor.getFIFOred();
    particleSensor.nextSample();
    tail++;
  }
  newSamples=0;
}
}

```

In this implementation, the `loop()` function begins by invoking the `check()` method. If new samples are detected, a `for` loop processes each sample. If the data array is full, the `sendData()` function is called to transmit the data, emptying the data array before saving new data inside. Otherwise, the red LED measurements are stored using the `getFIFOred()` method, and the `tail` pointer is incremented. After processing all new samples, the `newSamples` variable is reset, ensuring the loop continues to poll the sensor for additional data.

The FIFO Almost Full Interrupt

This logic aims to trigger data collection when the FIFO contains a user-specified number of unread samples. The loop function, combined with the `fetchData()` function, checks for an interrupt signal:

```

void loop() {
  if (particleSensor.readRegister8(0x57, INTERRUPT_STATUS_1) & 0x80) {
    fetchData();
  }
}

```

In the setup function of the sketch the interrupt is enabled allowing it's use in the loop function reported above. As said before, this interrupt can be programmed to activate when a specific

number of unread samples is present in the FIFO, during testing the number was kept to the default value of 32, which is the maximum amount of samples present in the FIFO. However, testing revealed this logic to be unreliable. The FIFO Almost Full flag never activated, even if the number of samples necessary for the interrupt trigger was set to 1, likely due to sensor issues. Consequently, this method was excluded from further development.

The Data Ready Interrupt

This logic triggers data collection whenever at least one new sample is available. Similar in implementation to the FIFO Almost Full method, it differs in the interrupt flag used:

```
void loop() {  
  if (particleSensor.readRegister8(0x57, INTERRUPT_STATUS_1) & 0x40) {  
    fetchData();  
  }  
}
```

The `INTERRUPT_STATUS_1` in the code above corresponds to the interrupt register where all the sensor's hardware interrupts are present, with the exception of the temperature interrupt. This last interrupt is issued by an optional temperature sensor that can be connected to the PPG sensor. Since in this implementation of the project is not present, it wasn't considered in the development of the code. The main difference of this code from the previous one that used an hardware interrupt as well, is the mask applied to the byte returned by the `readRegister8()` function, which is the content of the interrupt register 1. This mask (`0x40`) zeroes out all the bits except for the the one referred to the Data ready interrupt. This interrupt is issued when at least one new sample is present in the sensor's FIFO. The Data Ready Interrupt proved more reliable than the FIFO Almost Full flag, making it a viable alternative.

3.6 Testing of the Code

To develop a code capable of ensuring data acquisition, an analysis of execution times was conducted as part of the testing phase. The objective was to assess the speed of the code and its capacity to handle different sample rates without interruptions. For this purpose, specific modifications were introduced into the code. For measuring execution times, the program was altered to toggle the state of one or two analog pins at specific points, thus measuring execution time. As an example, one can think of having a pin always in the HIGH state while in idle and changing it's state to LOW when the `loop` function starts and turning it again in HIGH state when the function ends. In this way it is possible to correctly measure the duration of the loop

cycle during the execution of the program using an oscilloscope with a probe connected to said pin.

3.6.1 Methodology

During testing:

- An analog pin's state transitioned from high to low when the acquisition of sensor data began and reverted to high when the acquisition concluded. This pin essentially acted as an indicator for the start and end of a loop cycle.
- The program automatically adjusted the sample rate every minute, progressing incrementally from 50 to 1000 samples per second. This approach allowed for the assessment of both the sensor and code under varying data acquisition speeds.
- A mixed signal oscilloscope was connected to the analog pin, the SCL and SDA pins and the ground pin. This allowed us to analyze the correct behavior of the board during sample acquisition as well as the power consumption of the sensor.

3.6.2 Results and Discussion

From this analysis, two significant results were observed:

1. The FIFO_ALMOST_FULL interrupt-based acquisition logic failed to function correctly. Despite extensive debugging, no issues in the code were identified. This result, combined with the consistent malfunctioning of the interrupt pin under the same conditions, suggests a potential sensor-related fault. However, due to time constraints, this approach was discarded.
2. The `check()` method exhibited similar acquisition capabilities with respect to the DATA_RDY interrupt code. Both logic are capable of collecting all the samples produced by the PPG sensor but the latter has a faster execution of the loop. This was expected, due to the fact that the `check()` method requires two read from the sensor's registers when no data is found, and up to 34 if the FIFO is found full. Conversely, the DATA_RDY interrupt requires only one read from the sensor if no data is found, otherwise it executes the `check()` method when the `fetchData()` is called. In other words: if no samples are found the DATA_RDY interrupt is faster than the `check()` method and slightly slower if data are found.

Results are reported in table 3.2, where $t_{loop_noData_check}$ is the duration of a loop cycle in case of no data found in the sensor's FIFO for the check method, with $t_{loop_noData_DR}$ being its correspondent for the DATA_RDY interrupt. While $t_{loop_data_check}$ and $t_{loop_data_DR}$ are the durations of the loop cycle if samples are found in the sensor's FIFO for each logic respectively.

LED Mode	$t_{loop_noData_check}$ [μ s]	$t_{loop_data_check}$ [ms]	$t_{loop_noData_DR}$ [μ s]	$t_{loop_data_DR}$ [ms]
Red LED	906	1.5	456	2
Red+IR LED	906	1.8	456	2.3

Table 3.2: Acquisition times for the acquisition logic

Considering a worst case scenario, wich corresponds of two LED at a 1kHz sample rate, both programs are able to retrieve all the samples produced by the sensor. In fact, reading a single measurement for 2 LED (so 1 sample for the red and 1 for the infrared LED) takes $2.3ms$ ($t_{loop_data_DR}$ in case of 2 LEDs active) for the DATA_RDY interrupt logic and $1.8ms$ ($t_{loop_data_check}$) for the check() logic and, at 1kHz, each sample is produced every millisecond. It is clear that both programs initially lag behind the sensor's sample production. In each iteration of the loop, all the samples, accumulated in the internal FIFO of the PPG sensor since the previous reading operation, are acquired. Each value corresponding to a given LED is stored in 3 bytes, thus requiring about $270\mu s$ to be transferred, from sensor to board, thanks to an I2C bus with a clock line configured at $100kHz$. Means that the check() logic will retrieve 3 or 2 samples for each iteration while the DATA_RDY logic will retrieve between 4 and 3 samples for each iteration of the loop. However, at smaller sampler rates and with only one LED, the DATA_RDY logic, configured for faster execution, is capable of acquiring samples with a smaller delay respect to the check() logic.

Additional measurements were taken, in Figure 3.4 a loop cycle when new data is found in the sensor's FIFO is represented, while in Figure 3.5 we can see a complete iteration of a loop cycle for the check() logic when no data is found. Additionally, logical transition times for the I2C were measured, as it can be seen in figure 3.6 and 3.7. All measures are reported on tables 3.3 and 3.4, where:

- t_{LP} is the time required to check if the DATA_RDY interrupt has been activated
- t_{FP} is the time required to obtain the first pointer in the FIFO (read pointer)
- t_{SP} is the time required to obtain the second pointer in the FIFO (write pointer)
- t_{FR} is the time needed to request the unread samples from the FIFO
- t_{RS} is the time required to obtain the sample from the sensor via I2C

- t_{SDA} is the time between the start of a loop cycle and the start signal sent by the SDA pin
- t_{SCL} is the time between the start of a loop cycle and the beginning of the I2C communication signaled by a low state of the SCL after a HIGH to LOW transition on the SDA
- t_{LOW} is the duration of a LOW logic state of the SCL
- $t_{HD,DAT}$ is the time between the transition to "not listening" state of the SCL and the subsequent change of the state of the SDA to the next value
- $t_{SU,DAT}$ is the time between the transition to the next logical state of the SDA and the return to the "listening" state of the SCL

Measurement	Value [μs]
t_{LP}	456
t_{FP}	451
t_{SP}	454
t_{FR}	240
t_{RS}	402

Table 3.3: Acquisition steps recorded times

Measurement	Value [μs]
t_{SDA}	10
t_{SCL}	20
t_{LOW}	5,2
$t_{HD,DAT}$	0,54
$t_{SU,dat}$	4,7

Table 3.4: I2C logical transitions times

3.6.3 Power consumption

To estimate the power consumption of the system a 10Ω resistor was placed between the ground pin of the sensor and the one of the board. Then, an oscilloscope probe was placed on the positive terminal of the resistor, in this way, we knew the other side was automatically referred to zero (Figure 3.8). From this measurement we obtained the mean voltage of the sensor which is $43mV$ and the peak voltage which is $101,5mV$. The peaks in the voltage are due to the sensor's turn on of the LED for a measurement which increases the current absorbed by the device. From the mean and peak voltages we obtained the mean and peak current absorbed which are $4,3mA$

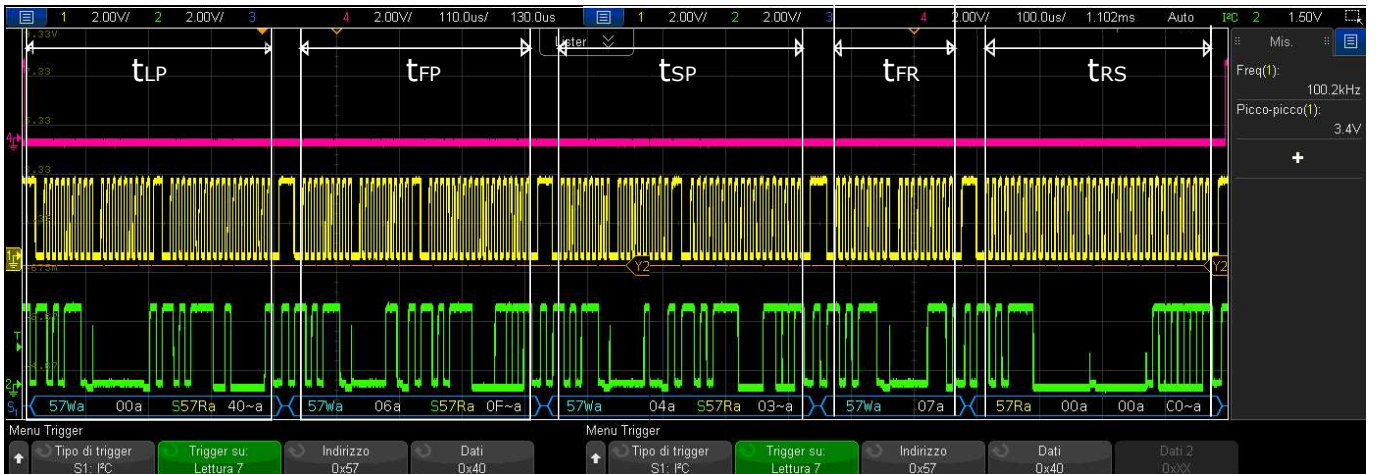


Figure 3.4: Complete loop cycle in case of new data found in the sensor's FIFO

and $10,15mA$ respectively, this means a mean power absorbed by the sensor of $0,18mW$ and a peak power of $1,03mW$.

To estimate the total power consumption of the device, instead of an oscilloscope we used a power profiler kit (PPK) produced by Nordic semiconductors. This device (specifically the Power profiler kit II) has a voltage controlled generator which was set to $3.3V$ and connected to the $3.3V$ pin of the board and the sensor's in parallel. Connecting a voltage generator to the $3.3V$ pin of the Adafruit board is a viable way to power the board but not recommended, this pin is not protected from overvoltages or overcurrents and so it should be done only if the user is sure that the voltage generator is stable. Connected to the voltage controlled generator in the PPK, there is an amperometer which measures the current absorbed by the system connected to the PPK. With this configuration we could precisely measure the total power consumption of the system (Figure 3.9). Because the board was not connected to a pc, a program which automatically configured the sensor for operation was previously uploaded and the sensor had the following configuration:

- `powerLevel = 31` meaning $6.4mA$ per LED
- `sampleAverage = 1`
- `ledMode = 1`
- `sampleRate = 400Hz`
- `pulseWidth = 69\mu s`
- `adcRange = 16384` meaning $62.5pA$ per LSB



Figure 3.5: Complete loop cycle in case of no data found in the sensor's FIFO for the check logic ($t_{loop_data_check}$)

We obtained a mean current of $8,97mA$ and a peak current of $20,92mA$, which, considering a $3,3V$ imposed by the voltage controlled generator, gives us a mean power absorbed of $29,6mW$ and a peak power of $69mW$. It has to be noted that the mean power absorbed measured by the PPK takes into consideration the peaks in the absorbed current while the mean power absorbed by the sensor's does not.

Lastly, an interesting phenomenon was observed during the measurement, every ~ 2 minutes the current absorbed by the system grew by an offset of $\sim 1,5mA$. This phenomenon was re-conducted to the transmission of the system of the acquired data on the serial port, even if it was not connected to the pc during testing. As it will be detailed later, we estimated a time to send the data of ~ 2 seconds which is consistent with the results obtained in the Sample reliability tests regarding the real serial communication speed of $\sim 245kB$.

3.6.4 Sample Reliability

To test the sample rate reliability a new version of the code was developed as a test bench of sorts. This version of the code was without the user interface to select the sensor parameters and instead was focused on acquiring samples from the sensor for a fixed interval. When the time interval, starting from the beginning of the acquisition, reached it's end, it would trigger a call to the `sendData()` function to send all the data acquired during the interval. The acquisition



Figure 3.6: Times required by the I2C to start

interval was selected to be 30 seconds such to never overflow the data array for all measurements up to 1kHz sensor's sampling frequency. A python program was used to log the serial communications. Based on the predefined sample rate and time interval, the expected number of samples was calculated as the sensor's sampling frequency times the acquisition interval. The results are shown in Table 3.5 and Table 3.6.

Sample Rate [samples/s]	Expected Samples	Acquired Samples	Received Samples
50	1500	1503	1503
100	3000	3007	3007
200	6000	6011	6011
400	12000	11993	11993
800	24000	23930	23930
1000	30000	29880	29880

Table 3.5: Sample counts for the first acquisition logic (DATA_READY method).

From this analysis it appears clearly that the sample rate of the sensor isn't perfect, up to 200 Hz the sensors collects slightly more data than expected and above this sampling frequency data are slightly lower than expected. The result is not critical, as deviation from the expected sample rate are up to 0.4% for 1kHz and lower for smaller sample rates. Additionally, the analysis showed that the serial communication between server and client is successful as no data is lost.

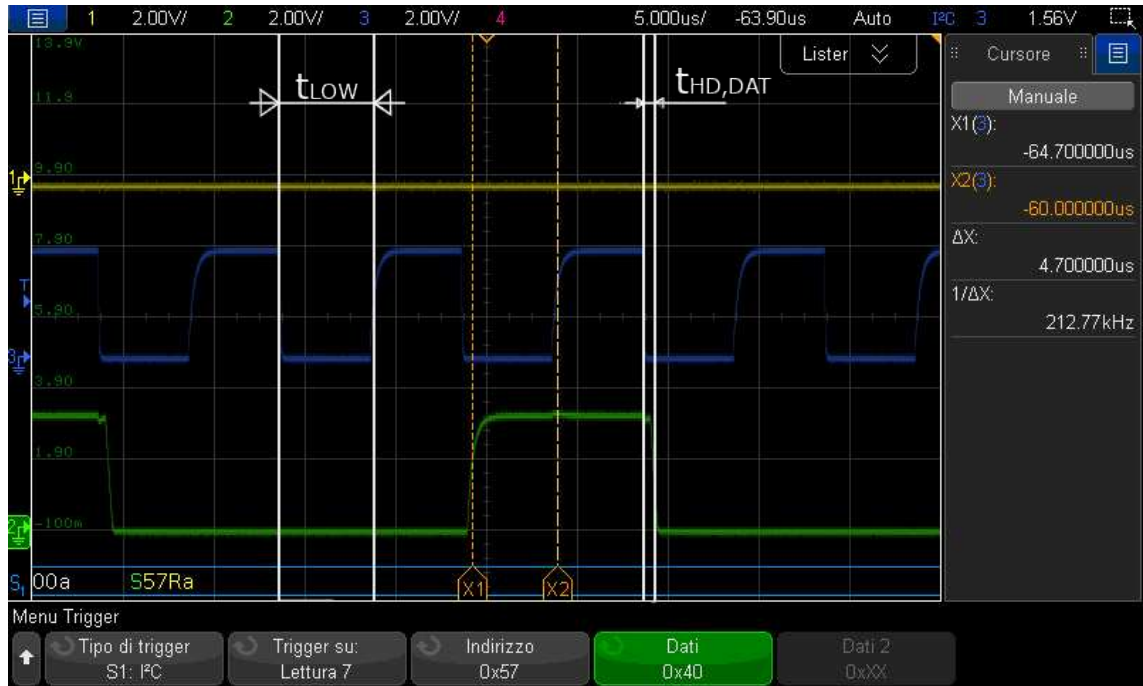


Figure 3.7: Logic transition times of the I2C, $t_{SU,DAT}$ is being directly measured by the oscilloscope

Sample Rate [samples/s]	Expected Samples	Acquired Samples	Received Samples
50	1500	1503	1503
100	3000	3007	3007
200	6000	6011	6011
400	12000	11992	11992
800	24000	23928	23928
1000	30000	29879	29879

Table 3.6: Sample counts for the second acquisition logic (check() interrupt).

Lastly, realizing in the previous tests that the serial communication speed was higher than the baud rate set at the beginning of the setup function, a serial communication test was held. This behavior is due to the fact that the Adafruit board has a native USB 2.0 which overwrites the baud rate set by the user. The objectives of the test were two: ensuring that all the samples stored in the data array were received by the client; and estimating the correct serial communication speed. To verify the correct reception of samples the `sendData()` function was modified again. The pointer to the next unfilled space in the data array (tail) was sent before the data, while data were sent without the identifiers. A python code on the pc would then open the serial port and wait for packets of data, once recognized the initial int sent (the tail pointer) it would then store the following bytes until the board finished sending the packet of samples, identified with the reception of an "end" word. The collected bytes would be divided by 4 (the number of bytes of

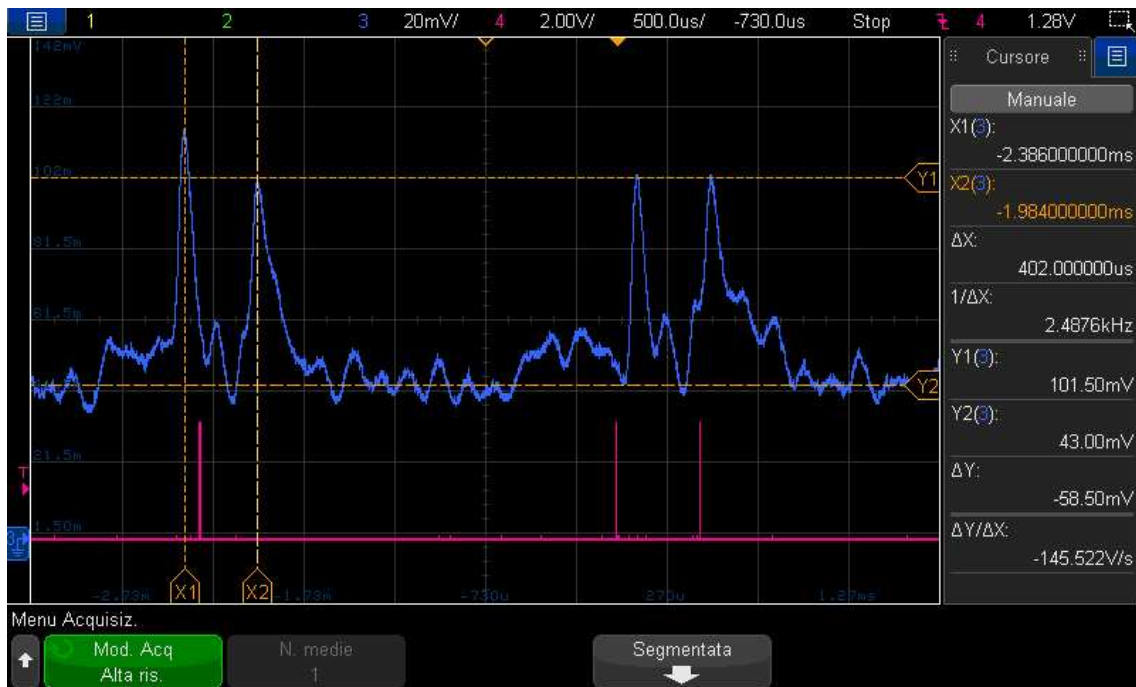


Figure 3.8: Power consumption measured with an oscilloscope. The blue line is the voltage on the resistor while the pink line is the interrupt pin showing the duration of a loop cycle

which a sample is composed) and the two amount would be compared. In this way we could not only asses the fact that all samples are received by the client but also if the sample rate set was reliable or not. For the serial communication speed a simple test was prepared, the board would collect 2500 samples (10000 Bytes) and it would send them to the client after sending the amount of samples that would be sent. A python program on the pc would then start a timer when the amount of samples to be sent would reach the serial port of the pc and time elapsed from the beginning of the reception of samples to the end would be taken, dividing the amount of bytes obtained by the time required to receive them gave us an estimation of the serial communication speed which is around $240kB/s$.

3.6.5 fetchData() Function

The 'fetchData()' function was developed with the scope to collect data from the sensor ensuring that no samples would be lost, as well as calling the sendData function when the data array is (or would be after the additions of a sample) full. The function begins by checking the number of samples currently available in the sensor's FIFO buffer and stores this value in the 'newSamples' variable. If 'newSamples' is greater than zero, the function checks whether adding these samples would exceed the board's array capacity. If so, it calls the 'sendData()' function which will send the stored samples to the client. After that, it retrieves and stores data for the red LED. If dual LED mode is enabled, the function also retrieves data for the infrared

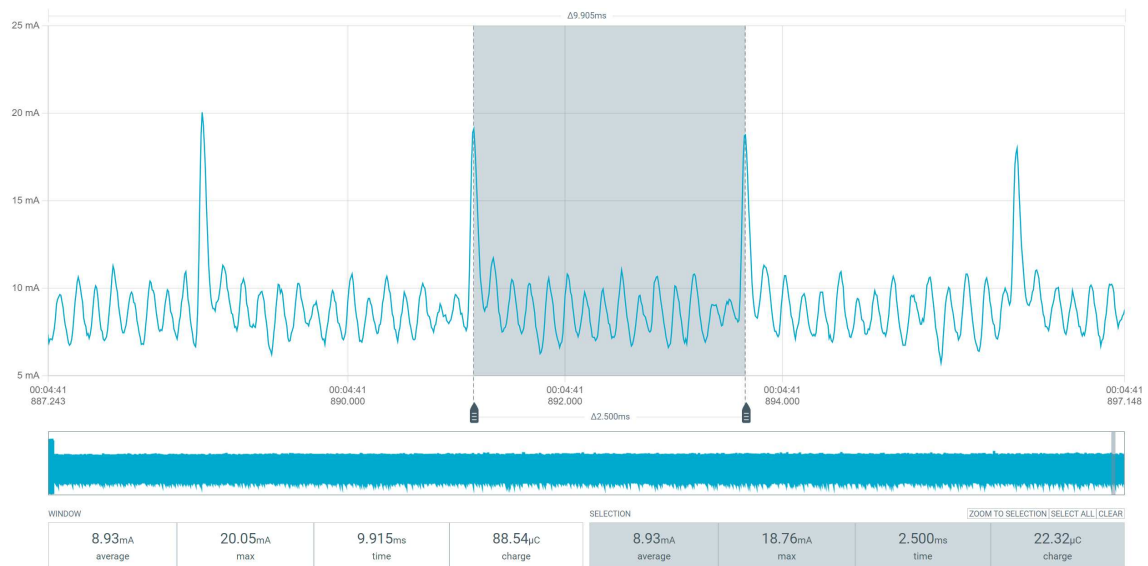


Figure 3.9: System power consumption measured with the power profiler kit

(IR) LED. For data retrieval, the ‘getFIFOred()’ and ‘getFIFOIR()’ methods are utilized. These two methods return the last unread sample in the sense array either from the red values array or IR values. Unlike ‘getRED()’ and ‘getIR()’, these methods avoid introducing delays. The getRED() method, when called, calls the safeCheck() method, which polls the sensor for new data like the check() method but tries repeatedly up to 250 ms before interrupting if no data is found during that time interval. Same happens for the getIR() method. After storing the data, the tail pointer in the sensor’s FIFO is updated using the ‘nextSample()’ method. This last action is necessary because when the getFIFOred() or getFIFOIR() are called they return the least unread sample collected from the FIFO without advancing the pointer to the next unread sample in the temporary sense array. Without calling for nextSample() the getFIFOred() method and the other would constantly return the same sample and data would be lost. Below is the implementation:

```
void fetchData() {
    newSamples = particleSensor.check();
    if (newSamples > 0) {
        for (int i = 0; i < newSamples; i++) {
            if (tail + (ledMode == 2 ? 2 : 1) > array_size) {
                sendData();
                break;
            }
        }

        // Salva i dati dal LED rosso
    }
}
```

```

ledVals[tail] = particleSensor.getFIFOred();
tail++;

// Se in modalità doppio LED, salva anche IR
if (ledMode == 2) {
    ledVals[tail] = particleSensor.getFIFOIR();
    tail++;
}

particleSensor.nextSample();
}
}

```

3.6.6 sendData() Function

The `sendData()` function is a pivotal component of the system, enabling communication between the board and the system tasked with data analysis. During the evaluation of potential communication solutions, two main approaches were considered:

- The first approach involved leveraging the Bluetooth Low Energy (BLE) functionality integrated into the board. However, for simplicity, this approach was discarded in favor of the serial port.
- The second approach, which was ultimately adopted, utilized the serial port. Using the serial port, the board does not need to be connected to a battery since it can be directly powered by the USB. If the BLE approach would have been chosen, this would have required the board to be connected to a battery in order to be powered.

We chose the serial port; in the following we introduce the program developed to allow the transmission of data between Adafruit and pc. Numerous examples of communication between an Arduino-programmed board and a PC were examined, primarily from forums. These examples utilized the serial port since the serial monitor is the only tool in the Arduino IDE that can be used for a two-way communication. Initially, the `sendData()` function employed the `Serial.print()` method to transmit both strings and measurement data. However, due to the prolonged transmission times associated with `Serial.print()`, the method was later replaced with `Serial.write()`.

The `Serial.print()` and `Serial.println()` methods transmit data as strings over the Serial bus. For a 32-bit integer measurement, these methods first convert each digit of the

measurement into a string. Assuming an average of five digits per measurement, this results in $8 \text{ bits} \times 5 \text{ digits} = 40 \text{ bits}$ being sent for each integer instead of the 32 bits that make up the original data. Taking in consideration the real serial communication speed estimated in the sample reliability tests, 8 bits extra per sample implies $4,16\mu s$ additional time for each sample. Although the additional time may seem negligible, considering that each `sendData()` call sends fifty thousand samples, means an additional total time of $208ms$. Supposing a sample rate of $400Hz$ and that sending 350000 bytes (7 bytes per sample plus identifiers times 50000) requires a little less than $2s$, means losing $\sim 10\%$ extra samples produced by the sensors during the execution of the `sendData()` function ($2s \times 400Hz = 800 \text{ samples}$ and $208ms \times 400Hz = 83 \text{ samples}$). Additionally, it must be considered that extra bits can speed up filling of the transmit buffer of the board.

Conversely, `Serial.write()` transmits data as a bit stream, thereby reducing the transmission time because it does not try to convert the `uint32_t` in to strings while the identifiers remain the same size respect to the other method. However, the method requires additional steps to convert strings into a bit stream for transmission and to later reconvert them on the PC to store the data. Furthermore, `Serial.write()` operates asynchronously, in other words, it does not interrupt the execution of the code, when the number of bits to transmit is less than or equal to the free space in the transmit buffer. By carefully optimizing code execution time, `Serial.write()` could ensure uninterrupted data transmission, enabling continuous sampling of sensor measurements. Although not specified in the datasheet, the transmit buffer was estimated to be of 80 bytes, considering the plausible presence of USB overhead and other factors we hypothesize the real dimension of the buffer to be 128 bytes which is a commercially standard value.

3.6.7 Implementation of the `sendData()` Function

Taking into consideration all the results reported in the above section regarding data transfer capabilities, the `sendData()` was developed with two main objectives:

- Being able to send all the accumulated data in a timely manner
- Ensuring that the data sent by the function would be correctly stored by the client pc

The function is called only when the data array is full (or, when two LED are active, at most one space is empty), and it will interrupt the acquisition of new data from the sensor. Considering the fact that the python client, when it stores the data sent from the serial port, does not provide a way to differentiate different blocks of acquired data. We decided to send a single sample of 0 value, one for each LED active, before sending the real sample. This simplifies the future analysis of the data acquired from the system, because the zero sample clearly marks the end of

an acquisition block and the beginning of the next one. Otherwise, artifacts in the signal could be created when the user does not correctly identify the end of an acquisition block and, for example, mistakenly identifies the presence of two consecutive peaks in the signal as a problem related to the heartbeat instead of simply missing data between the two events.

Another decision we made was to use identifiers before each data. As it is detailed in the next chapter regarding the python client, this decision was taken to allow the client to correctly differentiate to which active LED the samples correspond to. Thus, we developed a function that follows these following steps:

1. **Start:** A zero value sample is sent to differentiate between acquisition blocks.
2. **Iteration:** A for loop iterates over each entry in the data array, processing and transmitting the data to the PC.
3. **Serial Transmission:** Each sample in the data array is sent after his identifier to ensure correct storage without loss of information.
4. **Handling of Additional Data:** If the LED mode is set to 2, the function processes and transmits the IR LED data using similar logic.
5. **Termination Indicator:** Each data entry concludes with the transmission of a line termination indicator ";". This line terminator has been added to ensure that, in the csv file where the data will be stored by the client, the data will be saved in columns.

Below, the implementation is reported:

```
void sendData() {
    //Initially, send a zero value for each led to differentiate between
    //acquisition chunks
    Serial.write((const uint8_t*)"R:", 2);
    Serial.write((uint8_t*)&red_value_splitter, sizeof(red_value_splitter));
    if (ledMode == 2) {
        Serial.write((const uint8_t*)"I:", 3);
        Serial.write((uint8_t*)&ir_value_splitter, sizeof(ir_value_splitter));
    }
    Serial.write((const uint8_t*");", 1); // Invia il terminatore ";"

    for (int j = 0; j < tail; j++) {
        rValue = ledVals[j];
        Serial.write((const uint8_t*)"R:", 2);           // Invia "R:"
```

```

Serial.write((uint8_t*)&rValue, sizeof(rValue)); // Invia valore binario Red

if (ledMode == 2 && j + 1 < tail) {
    j++;
    irValue = ledVals[j];
    Serial.write((const uint8_t*)"I:", 3);           // Invia "I:"
    Serial.write((uint8_t*)&irValue, sizeof(irValue)); // Invia valore binario IR
}

Serial.write((const uint8_t*");", 1); // Invia il terminatore ";"
}

tail = 0;

```


Chapter 4

Python client

To collect and store the acquired data transmitted via the serial port, a Python program was developed. The client is designed to be straightforward, since its use will be restricted to experienced users during the research on sensorized insoles. To correctly interpret the data sent from the board, it was necessary to establish a communication protocol. Without such a protocol, the data collection code would be unable to distinguish the boundaries of individual samples, particularly when the sensor operates in single- or dual-LED mode. For instance, the program must identify whether it should expect one sample (red LED) or two samples (red and IR LEDs). To address this, a format was implemented where the board sends an identifier before each measurement: “R:” for red LED measurements and “IR:” for infrared LED measurements. This enables the client to correctly parse the data received from the server.

4.1 Main Program Execution

The main of the Python client acts as an entry point, executing the `read_and_store_data` function in a controlled environment. The program accepts a `filename` argument to specify where the collected data should be stored. Without this argument, the program halts execution and prompts the user to provide a valid file name. The data collection begins only after this setup is complete.

Additionally, the main program is designed to terminate upon receiving a keyboard interrupt (Ctrl+C). Using a `try/except/finally` structure to call the `read_and_store_data` we can be sure that the program can be closed correctly when the keyboard interrupt is issued ensuring that the serial port will be closed before closing the program. Also, using a `with` to open the file given as an argument: `with open(filename, "w", newline='') as csvfile:`, encases all the function in a `with` environment. This ensures that all resources, such as file handlers, are properly closed when the program exits.

It is important to note that this implementation is unidirectional. The Python script only listens to the data transmitted by the sensor and does not send commands back to it. To configure the sensor parameters, users must rely on the Arduino IDE's serial monitor prior to running the Python program. Moreover, care must be taken to close the Arduino IDE's serial monitor before starting the Python script, as the serial port can only be accessed by one application at a time.

4.2 Acquisition from serial port

The logic implemented in the Python script for data collection is as follows:

```
def read_and_store_data(ser, filename):
    with open(filename, "w", newline='') as csvfile: # Open the file in text write mode
        csvwriter = csv.writer(csvfile)
        csvwriter.writerow(["R:", "Value", "IR:", "Value"]) # Write header row

    buffer = b'' # Use bytes buffer to handle binary data

    while True:
        if ser.in_waiting > 0:
            data = ser.read(ser.in_waiting)
            buffer += data

        # Process complete entries
        while b";" in buffer:
            entry, buffer = buffer.split(b";", 1)

            # Match and process R:uint32_t, IR:uint32_t
            match_full = re.match(rb"R:(.{4}),IR:(.{4})", entry)
            if match_full:
                r_value_bytes = match_full.group(1)
                ir_value_bytes = match_full.group(2)
                # Decode uint32_t
                r_value = struct.unpack('I', r_value_bytes)[0]
                # Decode uint32_t
                ir_value = struct.unpack('I', ir_value_bytes)[0]
```

```

        csvwriter.writerow(["R:", r_value, "IR:", ir_value])
        print(f"Received and stored: R:{r_value}, IR:{ir_value}")
        continue

    # Match and process R:uint32_t
    match_single = re.match(rb"R:({4})", entry)
    if match_single:
        r_value_bytes = match_single.group(1)
        # Decode uint32_t
        r_value = struct.unpack('I', r_value_bytes)[0]
        csvwriter.writerow(["R:", r_value, "IR:", ""])
        print(f"Received and stored: R:{r_value}")
        continue

```

The `read_and_store_data` function handles real-time binary data streaming through a serial port and writes it to a CSV file in a structured format. Initially, it opens the specified file in write mode and writes a header row containing column labels for the red and IR LED values. A binary buffer (`buffer`) is initialized to temporarily store incoming data, ensuring that only complete entries are processed.

Within an infinite loop, the function checks whether data is available in the serial input buffer (`ser.in_waiting`). When data is detected, it is read and appended to the buffer. The buffer is then scanned for completed entries, which are delimited by a semicolon (;). Each complete entry is separated and processed individually.

Two patterns are checked in the data. The first identifies entries containing both red and IR LED measurements, where 4-byte unsigned integers represent the values. The data is extracted, decoded using the `struct.unpack` method, and written to the CSV file. The second pattern matches entries containing only red LED measurements, which are processed similarly.

For every successfully processed entry, the decoded data is printed to the console for real-time feedback, allowing the system operator to monitor the collected data.

4.3 Future Enhancements

Future developments of the client could include bidirectional communication, allowing users to configure sensor parameters directly through the Python script. Alternatively, a different communication interface could be adopted, especially if the system is to be deployed in wearable devices. For the scope of this thesis, however, the current implementation suffices as the primary objective is to evaluate the system's feasibility.

Chapter 5

System testing

Following the development of the measurement system, the subsequent phase involved acquiring waveforms from the foot. Multiple waveforms were collected from several subjects across two rounds of testing. After the initial round of evaluations, three subjects underwent a second round of testing to validate the findings obtained during the first phase.

5.1 Measurement setup

Sensor setup for testing was the following:

- `powerLevel = 31` meaning $6.4mA$ per LED
- `sampleAverage = 1`
- `ledMode = 1`
- `sampleRate = 400Hz`
- `pulseWidth = 411\mu s`
- `adcRange = 16384` meaning $62.5pA$ per LSB

The pulse width was set to its maximum value, enabling 18 bits per sample and thus providing higher resolution for the data collected. The sampling rate was determined to be 400 Hz, deemed sufficient to accurately capture the heartbeat. As previously discussed, the average human heart rate ranges from 60 to 90 beats per minute, corresponding to a frequency of approximately 1 to 1.5 beats per second. Consequently, a sampling rate of 400 samples per second is more than adequate to measure resting heart rates without any loss of information. Each waveform was recorded for a duration of four minutes, yielding approximately 100,000 samples. This extended acquisition time facilitated monitoring of the sensor's behavior over time.

Furthermore, it allowed each subject to find a comfortable resting position, ensuring minimal movement during measurements.

As previously mentioned, the `powerLevel` parameter was adjusted for the final measurement of each subject, increasing from 31 to 255. The objective was to evaluate whether higher power levels would improve light penetration through the hose. However, as will be elaborated in subsequent sections, this adjustment introduced additional complications, necessitating further investigation.

Multiple measurements were collected for each subject, specifically:

1. From the left hand index finger
2. From the foot without the hose
3. From the foot with the hose
4. From the foot with the hose and maximum `powerLevel`

The left index finger was chosen for the initial measurement due to its proximity to the heart, where a stronger signal was anticipated. Conversely, for measurements taken from the foot, it was considered irrelevant whether the left or right foot was used; this decision was therefore left to the participant's discretion. Measurements from the finger were conducted with the subject's arm resting on the workbench and the left index finger positioned on the sensor. Care was taken to avoid applying excessive pressure to the sensor, as this could compress the capillaries in the finger, resulting in poor or invalid readings. The presence of a bypass capacitor beneath the sensor posed a challenge in maintaining the finger's stability, as it created a pivot point, causing the sensor to tilt under pressure. This issue was resolved by placing an insole beneath the sensor. The soft surface of the insole accommodated the capacitor, providing stable support for the sensor.

For foot measurements, subjects were seated in a chair with their legs resting at a 90° angle. To minimize movement-related artifacts in the waveform, participants were instructed to remain as still as possible during data acquisition. The insole was placed on the floor, with the sensor positioned under the foot on the inner side (e.g., the leftmost side for the right foot and the rightmost side for the left foot). According to the literature [7], this region contains the highest concentration of veins near the skin's surface, making it the optimal site for data collection. The exact placement of the sensor varied between subjects, depending on the visibility of veins beneath the skin. The sensor was placed at a location where a vein was discernible. An example of the foot acquisition setup is provided in Figure 5.1.

The subjects remained anonymous; however, certain biological data were collected, including:

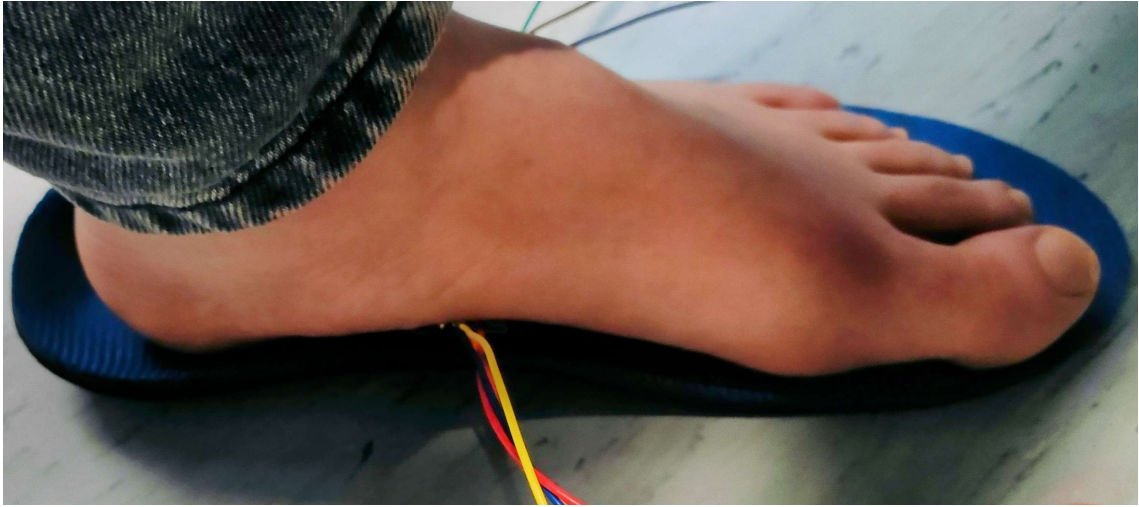


Figure 5.1: Acquisition setup of a subject

- Age
- Gender
- Physical activity level (active or sedentary)
- Weight

The ages of the subjects ranged from 21 to 26 years, while their weights varied between 48 kg and 80 kg. Of the eight subjects tested, five were men and three were women. Among the men, four were physically active and one was sedentary, whereas among the women, two were physically active and one was sedentary. It is worth noting that all subjects had a history of engaging in sports. This criterion was intentionally chosen, as literature suggests that physical activity during youth promotes better vascular development in the foot.

5.2 Waveforms analysis

To analyze the waveforms, a MATLAB script was developed by Professor G. Giorgi to process and interpret the physiological signal dataset. The primary workflow of the script includes data import, downsampling, and noise filtering. Baseline variations are estimated and subtracted to detrend the signal, while key peaks are identified to compute heart rate metrics. The script generates visualizations at each stage, providing valuable insights into the signal's characteristics and periodicity.

Of particular importance for this study were the processes of baseline removal and filtering. The signals consistently exhibited noise above 20 Hz, which significantly hindered waveform

visualization. Filtering was employed to eliminate high-frequency noise, whereas baseline removal facilitated better utilization of the MATLAB plotting scale. Signal values ranged between 28,000 and 35,000, influenced by factors such as respiration and ambient light. Additionally, when acquisition begins, signal values initially start at zero and rapidly increase by several orders of magnitude. This behavior necessitated caution during filtering, as the Butterworth filter used could introduce overshooting artifacts into the processed waveform. Baseline removal mitigated these effects, enhancing the clarity of the acquired signal and reducing the plotting scale in MATLAB, resulting in more interpretable visualizations.

An example of this filtering process is illustrated in Figures 5.2 and 5.3, which display the same signal before and after filtering and baseline removal.

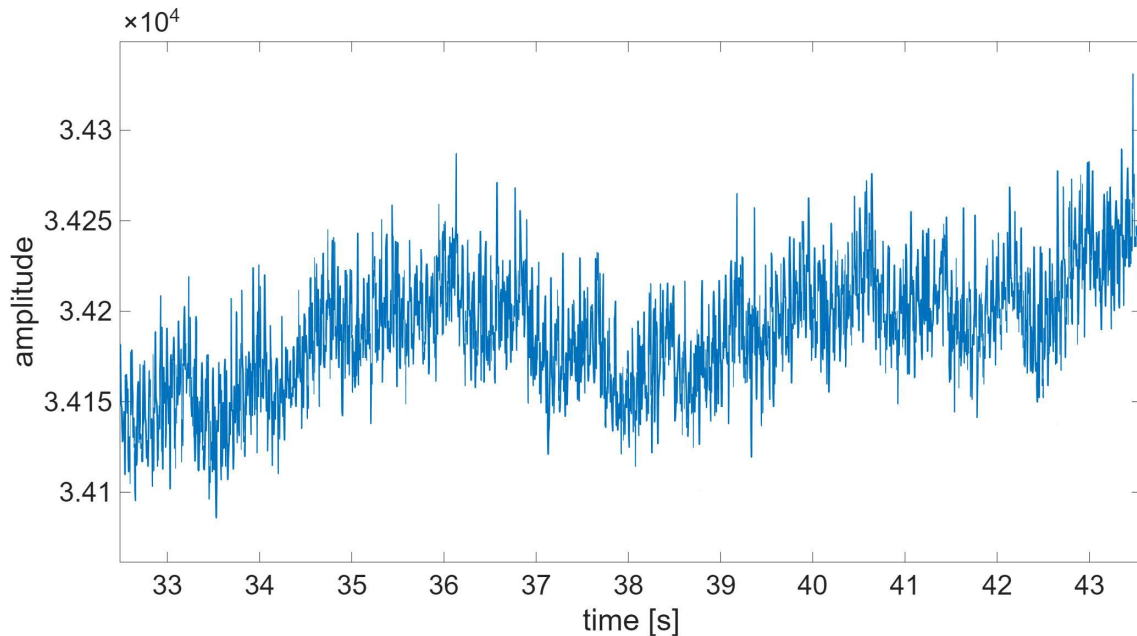


Figure 5.2: Signal from the finger before filtering

Waveforms acquired from the finger consistently displayed clear heartbeat signals across all subjects. This outcome was anticipated, as the sensor is specifically designed for use on the finger. An example of such waveforms is presented in Figure 5.3.

In contrast, barefoot measurements revealed a detectable heartbeat in only 2 out of the 8 participants. Figures 5.4 and 5.5 provide examples of successful and unsuccessful measurements, respectively. However, foot measurements conducted at maximum power settings resulted in the saturation of the ADC. These observations prompted further investigation into the underlying causes and necessitated a second round of testing.

Possible causes of measurement errors include:

- Incorrect measurement site

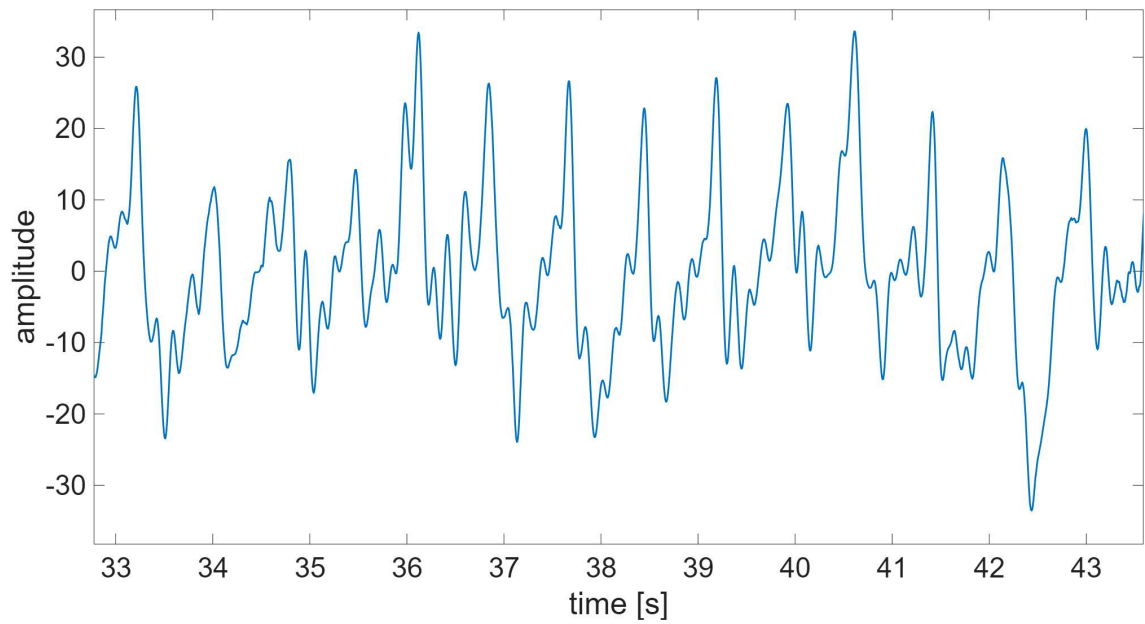


Figure 5.3: Signal from the finger after filtering

- Refraction from adjacent capillary vessels
- Insufficient light penetration into the skin
- Inappropriate pressure applied to the sensor

Among these, an incorrect measurement site was identified as the most probable cause. Refraction from neighboring capillary vessels is likely responsible for a complex morphology in the acquired waveforms compared to the ideal signal, as also suggested in the reviewed literature [17]. A second round of testing was conducted on three subjects who had participated in the first round. Two of these subjects had previously exhibited a detectable signal from the bare foot, while the third had not presented any measurable heartbeat from the foot. New measurements were performed by testing multiple adjacent sites on the foot and also utilizing infrared light. All other sensor parameters remained unchanged. Testing focused solely on barefoot measurements to investigate the reasons for signal detection in the two subjects and to determine if a signal could be obtained from the third subject.

To ensure consistent and adequate pressure application, a piece of cloth was used in some measurements to secure the sensor against the skin, simulating the conditions of an enclosed environment, such as within a shoe. Additionally, smaller, localized measurements were conducted on adjacent sites to examine the impact of accurate sensor placement.

Waveforms obtained during the second round of testing highlighted the critical importance of sensor placement. A visible heart rate signal was successfully obtained from all three subjects, alongside noise-contaminated signals from the two subjects who had previously exhibited

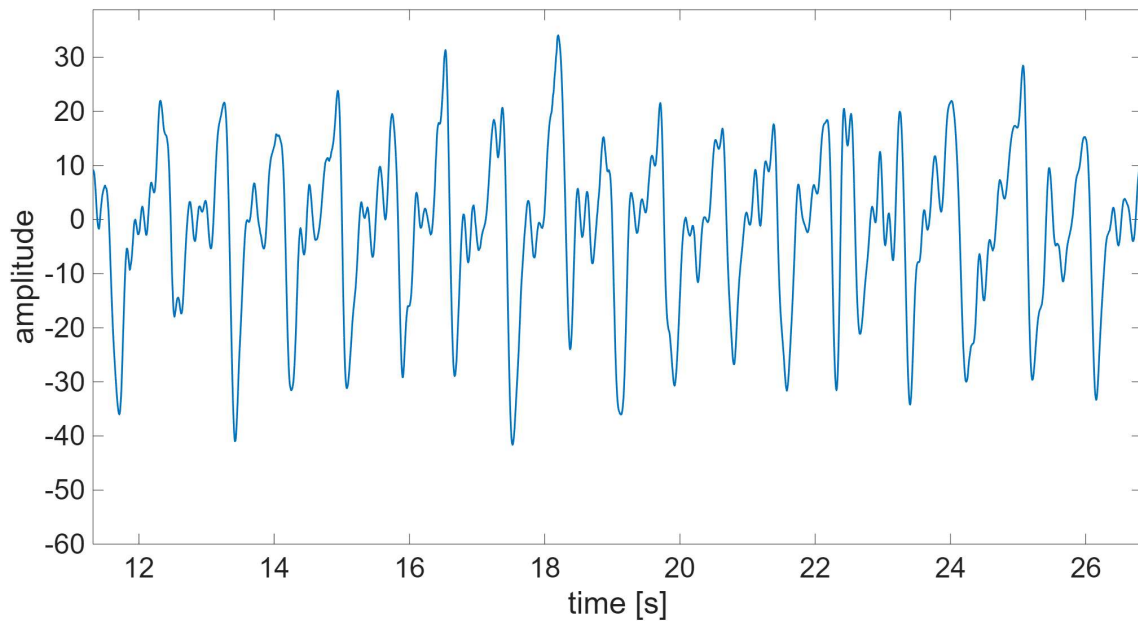


Figure 5.4: Filtered waveform of a heartbeat from the foot

a detectable signal. Specifically, waveforms recorded from the side of the foot demonstrated clearer signals compared to those from the sole. The best measurement, reported in Figure 5.7, was acquired from the side of the foot with the sensor secured by a piece of cloth, as depicted in Figure 5.6. This result can be attributed to the presence of the great saphenous vein, discussed earlier in the introduction. The higher blood volume flowing from the feet to the heart produces greater attenuation in the reflected red light. This increased amplitude excursion in the signal enhances the heart rate’s visibility amidst noise, facilitating easier identification. Furthermore, waveform analysis showed a lower likelihood of incorrect heartbeat estimations when negative peaks were analyzed. These negative peaks, generated by deoxygenated hemoglobin, reinforce the reliability of measurements taken from veins. Unfortunately, the substantial variability in the venous anatomy of the foot among individuals necessitates either customized insoles for each user or the implementation of a sensor matrix to expand the measurement area and identify the optimal site.

With respect to the use of different wavelengths, measurements taken with both LEDs revealed no significant differences. Apart from the smaller baseline observed with infrared light compared to red light, the infrared light did not provide additional information. The findings suggest that while the second wavelength may be valuable for blood oxygenation measurements, the red light alone is sufficient for heart rate estimation, consistent with the literature [18].

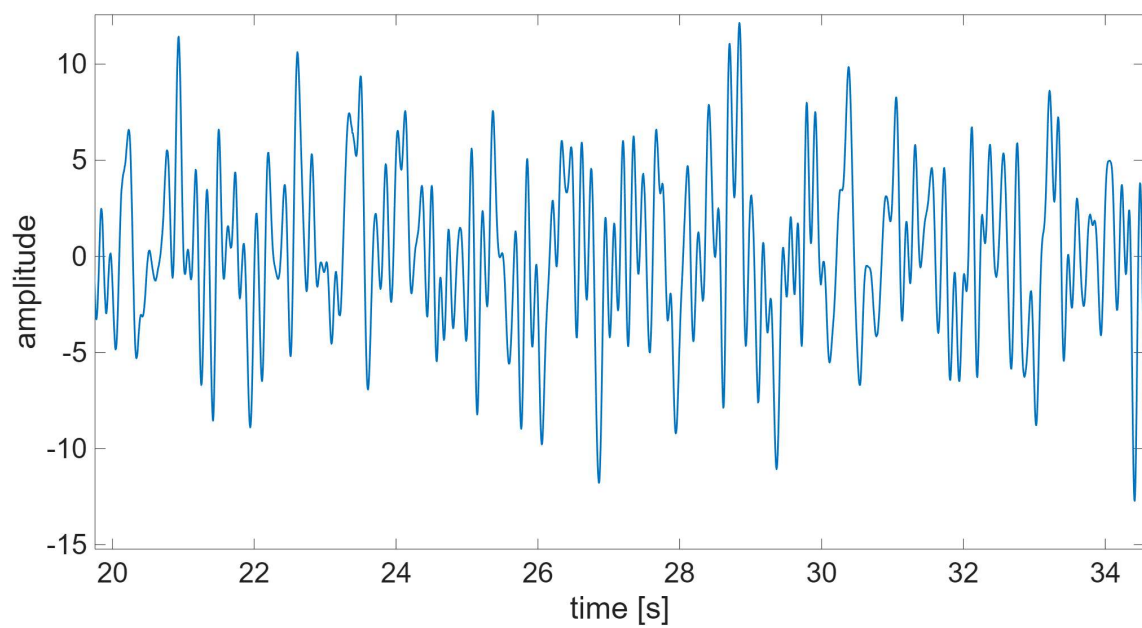


Figure 5.5: Filtered waveform from the foot not exhibiting a heartbeat



Figure 5.6: Lateral acquisition from the foot

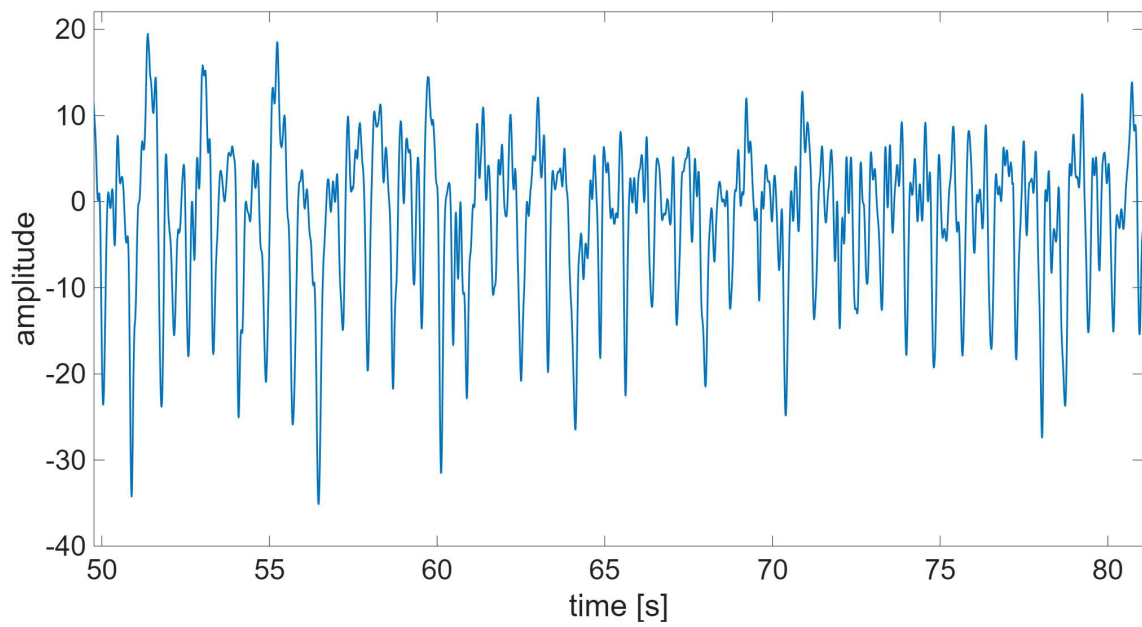


Figure 5.7: Heart rate acquired from the side of the foot

Chapter 6

Conclusions

The scope of this master's thesis is to provide a device that enables an initial study on the feasibility of obtaining a heartbeat signal from a PPG sensor at foot level. The concept originated from the aim of equipping workers with a health-monitoring device that does not interfere with their clothing or require the addition of wearable accessories. Should the ability to reliably measure heart rate from the sole of the foot be confirmed, this would pave the way for the development of sensorized insoles capable of performing such measurements. When integrated into safety footwear, these insoles would offer a means of health monitoring that is unobtrusive and practical in occupational settings.

This work focuses on the development of the firmware for the aforementioned device. Specifically, a program was developed to acquire all samples generated by the PPG sensor and transmit them correctly to a Python-based client. Although the program is functional, further development is required. For the purpose of the initial study, the acquisition of data in discrete chunks does not present a significant limitation; however, should the device be developed into a finalized product, this approach would become inadequate. Continuous data streaming is more desirable for effective health monitoring, as it allows for more accurate and timely detection of potential health issues. Likewise, the current use of the serial port as the communication interface is not ideal for a final product; eventually, it would need to be replaced by a wireless interface, such as Bluetooth Low Energy (BLE) or WiFi, both of which are supported by the board.

Furthermore, firmware testing revealed that program efficiency improves when multiple samples are collected simultaneously, thereby reducing the total acquisition time. To this end, the use of the `FIFO_ALMOST_FULL` interrupt presents an optimal solution, as it could trigger data acquisition only when a specified number of samples are available in the FIFO buffer. This would also eliminate the need to manually compute the number of samples to retrieve, since the board already tracks this internally. Additionally, having estimated the board's transmit

buffer size and depending on the communication speed, continuous data transmission appears achievable. Further investigation is therefore encouraged to understand why the interrupt and its corresponding pin failed to function as intended. By leveraging the transmit buffer and the non-blocking nature of the `serial.write()` function, the board could utilize the time between acquisitions to transmit data, thereby preventing the data array from reaching capacity and requiring manual intervention to clear it.

With regard to the Python client developed for persistent storage of the acquired PPG signal, although it is functional, further development is also needed. In its current state, the program only receives data from the measurement system, which imposes constraints on the firmware by requiring it to label each sample individually. As discussed earlier, enabling two-way communication between the client and the device could reduce the size of transmitted data and, consequently, the overall transmission time via the serial port.

Finally, the device was tested on a small group of subjects. Analysis of the acquired waveforms indicated that clear signals could be obtained from the foot, though proper sensor positioning is critical. In particular, pressure and placement appear to be the most influential factors. To address the positioning issue, the implementation of a small sensor matrix is proposed. By using multiple sensors, the device could identify and prioritize the sensor providing the clearest signal. Additionally, the remaining sensors could be used to isolate and subtract common noise components, likely caused by ambient light or physiological artifacts such as capillary and muscle movement. Further studies are required, particularly regarding the sensor's ability to acquire signals through layers such as socks or work garments typically worn in occupational environments. The effects of varying light intensities have not been fully explored, with the exception of maximum intensity, which consistently caused sensor saturation.

Bibliography

- [1] R. Mukundan, A. Khosla, P. K. Sekhar, P. Hesketh, C. Henry, and L. Magagnin, “Ubiquitous sensors are becoming an integral part of internet of things (iot) applications, and progress in this domain can be seen each month.,” *Journal of The Electrochemical Society*, vol. 165, no. 8, Y9–Y9, 2018. doi: 10.1149/2.0271808jes.
- [2] World Health Organization, *Who surgical safety checklist and implementation manual*, ISBN: 9789241598590, Geneva: World Health Organization, 2008. [Online]. Available: <https://www.who.int/docs/default-source/patient-safety/9789241598590-eng-checklist.pdf>.
- [3] S Ryals, A Chang, S Schutte-Rodin, *et al.*, “Photoplethysmography—new applications for an old technology: A sleep technology review,” *Journal of Clinical Sleep Medicine*, vol. 19, no. 1, pp. 189–195, 2023.
- [4] H. W. Loh *et al.*, “Application of photoplethysmography signals for healthcare systems: An in-depth review,” *Computer Methods and Programs in Biomedicine*, vol. 216, Apr. 2022. doi: 10.1016/j.cmpb.2022.106677.
- [5] H. Lee, H. Chung, H. Ko, and J. Lee, “Wearable multichannel photoplethysmography framework for heart rate monitoring during intensive exercise,” *IEEE Sensors Journal*, vol. 18, no. 7, pp. 2983–2993, Apr. 2018. doi: 10.1109/JSEN.2018.2801385.
- [6] P. G. Rukmini, R. B. Hegde, B. K. Basavarajappa, *et al.*, “Recent innovations in footwear and the role of smart footwear in healthcare—a survey,” *Sensors*, vol. 24, no. 13, p. 4301, 2024. doi: 10.3390/s24134301. [Online]. Available: <https://www.mdpi.com/1424-8220/24/13/4301>.
- [7] G. Anastasi, *Anatomia Umana. Trattato*, Italian, 4th ed. Edi-Ermes, 2006, vol. 2, p. 550.
- [8] Sobotta, *Sobotta Atlas of Human Anatomy, Vol. 2: Trunk, Viscera, Lower Limb*, 14th ed. Churchill Livingstone, 2006, isbn: 0443103496; 9780443103490.
- [9] K. Rembor, *Adafruit feather nrf52840 sense*, Last major update January 24, 2024., 2020. [Online]. Available: <https://learn.adafruit.com/adafruit-feather-sense>.

- [10] B. Wire, *Analog devices completes acquisition of maxim integrated*, Last accessed 26 August 2021, 2021. [Online]. Available: <https://www.businesswire.com/news/home/20210826005486/en/>.
- [11] S. Electronics, *Sparkfun bio sensor hub library*, n.d. [Online]. Available: https://github.com/sparkfun/SparkFun_Bio_Sensor_Hub_Library.
- [12] A. Devices, *Max30101 pulse oximeter and heart-rate sensor for wearable health*, n.d. [Online]. Available: <https://www.analog.com/media/en/technical-documentation/data-sheets/max30101.pdf>.
- [13] A. Devices, *Max3010x evaluation kits recommended configurations and operating profiles*, Page 28, n.d. [Online]. Available: <https://www.analog.com/media/en/technical-documentation/user-guides/max3010x-ev-kits-recommended-configurations-and-operating-profiles.pdf>.
- [14] M. Massaro, *Analysis and programming of the adafruit feather sense multi-sensor system*, University of Padua, Thesis., 2025. [Online]. Available: <https://hdl.handle.net/20.500.12608/67644>.
- [15] Arduino. “Arduino - open-source electronic prototyping platform.” (2025), [Online]. Available: <https://www.arduino.cc/>.
- [16] devxplained, *Max3010x sensor library*, Accessed: 2025-03-19, 2025. [Online]. Available: <https://github.com/devxplained/MAX3010x-Sensor-Library>.
- [17] F. W. Jerve, D. R. Hjelme, H. Kalvøy, J. Allen, and C. Tronstad, “Exploration of the conditions for occurrence of photoplethysmographic signal inversion above the dorsalis pedis artery,” *Sensors*, vol. 24, p. 6505, 2024, Academic Editors: Mimma Nardelli and Raquel Bailón. Received: 15 September 2024; Revised: 1 October 2024; Accepted: 8 October 2024; Published: 10 October 2024. doi: 10.3390/s24206505.
- [18] J. Allen, “Photoplethysmography and its application in clinical physiological measurement,” *Physiological Measurement*, vol. 28, no. 3, R1, 2007. doi: 10.1088/0967-3334/28/3/R01. [Online]. Available: <https://dx.doi.org/10.1088/0967-3334/28/3/R01>.

Addendum

For completion, in this addendum all the code developed in this master thesis is reported.

Firmware for the Adafruit board

```
#include "MAX30105.h"

MAX30105 particleSensor;

#define debug Serial //Uncomment this line if you're using an Uno or ESP
//#define debug SerialUSB //Uncomment this line if you're using a SAMD21
#define INTERRUPT_STATUS_1 0x00 //interrupt register address
//setup constants
uint8_t powerLevel = 0x1F; //how much power the led has
uint8_t sampleAverage = 1; //how many samples are averaged for each measurement
uint8_t ledMode = 1; //how many led are active; 1=red only, 2=red+IR
int sampleRate = 400; //how many samples taken per second
int pulseWidth = 69; //how long the led pulse is
int adcRange = 16384; //definition of the adc

void setup() {
  debug.begin(9600);

  // Initialize sensor
  if (particleSensor.begin() == false) {
    debug.println("MAX30102 was not found. Please check wiring/power. ");
  }
}
```

```

    while (1)
    ;
}
// Wait for serial port to connect. Needed for native USB port only
while (!Serial) {
    ; // wait for serial port to connect. Needed for native USB port only
}

```

```

Serial.println("Default power level is 0x1F=6.2mA, minimum value is 0x00 corresponding
to 0mA, maximum value is 0xFF corresponding to 51mA. Higher values will increase
the detection range. Would you like to modify it? (y or n)");
// Wait for response and process it
while (true) {
    // Check if response is available to read
    if (Serial.available() > 0) {
        String answer = Serial.readStringUntil('\n');
        // Trim any whitespace
        answer.trim();
        if (answer == "n") {
            break;
        } else if (answer == "y") {
            Serial.println("Please provide a value for the led Power level");
            while (true) {
                // Check if data is available to read
                if (Serial.available() > 0) {
                    // Read the incoming data as a string
                    String receivedString = Serial.readStringUntil('\n');
                    // Trim any whitespace
                    receivedString.trim();
                    // Convert the string to an integer
                    int newLevel = receivedString.toInt();
                    // Check if the integer is within the valid range for uint8_t
                    if (newLevel >= 0 && newLevel <= 255) {
                        uint8_t newpowerLevel = (uint8_t)newLevel;
                        powerLevel = newpowerLevel;
                    }
                }
            }
        }
    }
}

```

```

        break;
    } else {
        Serial.println("Invalid power level entered. Please enter a value
        between 0 and 255.");
    }
}
}
break;
}
}
}
Serial.println("Default sample average is 1. Available sample average are 1, 2,
4, 8, 16 and 32. Defines the number of samples averaged for each measurement.
Would you like to modify it? (y or n)");
// Wait for response and process it
while (true) {
    // Check if response is available to read
    if (Serial.available() > 0) {
        // Read the incoming response as a String
        String answer = Serial.readStringUntil('\n');
        // Trim any whitespace
        answer.trim();
        if (answer == "n") {
            break;
        } else if (answer == "y") {
            Serial.println("Please provide a value fot the sample average");
            while (true) {
                // Check if data is available to read
                if (Serial.available() > 0) {
                    // Read the incoming data as a string
                    String receivedString = Serial.readStringUntil('\n');
                    // Trim any whitespace
                    receivedString.trim();
                    // Convert the string to an integer
                    int newAverage = receivedString.toInt();
                    // Check if the integer is within the valid range for uint8_t

```

```

        if (newAverage == 1 || newAverage == 2 || newAverage == 4 ||
newAverage == 8 || newAverage == 16 || newAverage == 32) {
            uint8_t newsampleAverage = (uint8_t)newAverage;
            sampleAverage = newsampleAverage;
            break;
        } else {
            Serial.println("Invalid number of samples to be averaged entered.
Please enter a value between 1, 2, 4, 8, 16 and 32.");
        }
    }
}
break;
}
}

Serial.println("Default led mode is 1, corresponding to red only. Available
led modes are 1 and 2 with the latter meaning red+IR. Would you like to
modify it? (y or n)");
// Wait for response and process it
while (true) {
    // Check if response is available to read
    if (Serial.available() > 0) {
        // Read the incoming response as a String
        String answer = Serial.readStringUntil('\n');
        // Trim any whitespace
        answer.trim();
        if (answer == "n") {
            break;
        } else if (answer == "y") {
            Serial.println("Please provide a value fot the sample rate");
            while (true) {
                // Check if data is available to read
                if (Serial.available() > 0) {
                    /// Read the incoming data as a string
                    String receivedString = Serial.readStringUntil('\n');
                    // Trim any whitespace

```

```

        receivedString.trim();
        // Convert the string to an integer
        int newMode = receivedString.toInt();
        // Check if the integer is within the valid range for uint8_t
        if (newMode == 1 || newMode == 2) {
            ledMode = newMode;
            break;
        } else {
            Serial.println("Invalid led mode entered. Please enter a value
            between 1 and 2.");
        }
    }
}
break;
}
}

Serial.println("Default sample rate is 50. Available rates are 50, 100, 200,
400, 800, 1000, 1600, 3200. Would you like to modify it? (y or n)");
// Wait for response and process it
while (true) {
    // Check if response is available to read
    if (Serial.available() > 0) {
        // Read the incoming response as a String
        String answer = Serial.readStringUntil('\n');
        // Trim any whitespace
        answer.trim();
        if (answer == "n") {
            break;
        } else if (answer == "y") {
            Serial.println("Please provide a value fot the sample rate");
            while (true) {
                // Check if data is available to read
                if (Serial.available() > 0) {
                    // Read the incoming data as a string
                    String receivedString = Serial.readStringUntil('\n');

```

```

// Trim any whitespace
receivedString.trim();
// Convert the string to an integer
int newRate = receivedString.toInt();
// Check if the integer is within the valid range for uint8_t
Serial.print(newRate);
if (ledMode == 1) {
    if (newRate == 50 || newRate == 100 || newRate == 200 ||
newRate == 400 || newRate == 800 || newRate == 1000 || newRate == 1600
|| newRate == 3200) {
        sampleRate = newRate;
        break;
    } else {
        Serial.println("Invalid sample rate entered. Please enter a value
between 50, 100, 200, 400, 800, 1000, 1600 and 3200.");
    }
} else {
    if (newRate == 50 || newRate == 100 || newRate == 200 || newRate == 400
|| newRate == 800 || newRate == 1000 || newRate == 1600) {
        sampleRate = newRate;
        break;
    } else {
        Serial.println("Invalid sample rate entered. Please enter a value
between 50, 100, 200, 400, 800, 1000, and 1600.");
    }
}
}
break;
}
}
}
Serial.println("Default pulse width is 69 us. Available width are 69, 118, 215,
411. Please note that not all pulse width are available for all sample rates
and led modes. Would you like to modify it? (y or n)");
// Wait for response and process it

```

```

while (true) {
    // Check if response is available to read
    if (Serial.available() > 0) {
        // Read the incoming response as a String
        String answer = Serial.readStringUntil('\n');
        // Trim any whitespace
        answer.trim();
        if (answer == "n") {
            break;
        } else if (answer == "y") {
            Serial.println("Please provide a value fot the pulse width");
            while (true) {
                // Check if data is available to read
                if (Serial.available() > 0) {
                    // Read the incoming data as a string
                    String receivedString = Serial.readStringUntil('\n');
                    // Trim any whitespace
                    receivedString.trim();
                    // Convert the string to an integer
                    int newWidth = receivedString.toInt();
                    // Check if the integer is within the valid range for uint8_t
                    if (ledMode = 1) {
                        if (newWidth < 118) {
                            pulseWidth = newWidth;
                            break;
                        } else if (newWidth > 69 && newWidth < 411) {
                            //check before if the sample rate is too high otherwise do nothing
                            if (sampleRate > 1600) {
                                sampleRate = 1600;
                                Serial.println("Sample rate set at 1600");
                            }
                            pulseWidth = newWidth;
                            break;
                        } else if (newWidth == 411) {
                            //check before if the sample rate is too high otherwise do nothing
                            if (sampleRate > 1000) {

```

```

        sampleRate = 1000;
        Serial.println("Sample rate set at 1000");
    }
    pulseWidth = newWidth;
    break;
} else {
    Serial.println("Invalid pulse width entered. Please enter a value
    between 69, 118, 215 or 411.");
}
}
if (ledMode = 2) {
    if (newWidth < 118) {
        //check before if the sample rate is too high otherwise do nothing
        if (sampleRate > 1600) {
            sampleRate = 1600;
            Serial.println("Sample rate set at 1600");
        }
        pulseWidth = newWidth;
        break;
    } else if (newWidth > 69 && newWidth < 215) {
        //check before if the sample rate is too high otherwise do nothing
        if (sampleRate > 1000) {
            sampleRate = 1000;
            Serial.println("Sample rate set at 1000");
        }
        pulseWidth = newWidth;
        break;
    } else if (newWidth > 118 && newWidth < 411) {
        //check before if the sample rate is too high otherwise do nothing
        if (sampleRate > 800) {
            sampleRate = 800;
            Serial.println("Sample rate set at 800");
        }
        pulseWidth = newWidth;
        break;
    } else if (newWidth == 411) {

```

```

        //check before if the sample rate is too high otherwise do nothing
        if (sampleRate > 400) {
            sampleRate = 400;
            Serial.println("Sample rate set at 400");
        }
        pulseWidth = newWidth;
        break;
    } else {
        Serial.println("Invalid pulse width entered. Please enter a value
        between 69, 118, 215 or 411.");
    }
}
}
}
break;
}
}
}

Serial.println("Default adc range is 16384. Available ranges are 2048, 4096,
8192, 16384. The lower the adc range the higher the resolution of the signal.
Would you like to modify it? (y or n)");
// Wait for response and process it
while (true) {
    // Check if response is available to read
    if (Serial.available() > 0) {
        // Read the incoming response as a String
        String answer = Serial.readStringUntil('\n');
        // Trim any whitespace
        answer.trim();
        if (answer == "n") {
            break;
        } else if (answer == "y") {
            Serial.println("Please provide a value fot the adc range");
            while (true) {
                // Check if data is available to read
                if (Serial.available() > 0) {

```

```

        // Read the incoming data as a string
        String receivedString = Serial.readStringUntil('\n');
        // Trim any whitespace
        receivedString.trim();
        // Convert the string to an integer
        int newRange = receivedString.toInt();
        // Check if the integer is within the valid range for uint8_t
        if (newRange < 16384 || newRange == 16384) {
            adcRange = newRange;
            break;
        } else {
            Serial.println("Invalid adc range entered. Please enter a value
            between 2048, 4096, 8192, 16384.");
        }
    }
}
break;
}
}

particleSensor.setup(powerLevel, sampleAverage, ledMode, sampleRate, pulseWidth,
adcRange); //Configure sensor.
particleSensor.clearFIFO();
particleSensor.enableDATARDY();
Serial.println("Acquisition started");
}

const int array_size = 50000;
uint32_t ledVals[array_size];
int tail;
uint16_t newSamples;

void loop() {
if (particleSensor.readRegister8(0x57, INTERRUPT_STATUS_1) & 0x40) {
    fetchData();
}
}

```

```

}
//Serial.write(1);
}

void fetchData() {
newSamples = particleSensor.check();
if (newSamples > 0) {
    for (int i = 0; i < newSamples; i++) {
        if (tail + (ledMode == 2 ? 2 : 1) > array_size) {
            sendData();
            break;
        }

        // Salva i dati dal LED rosso
        ledVals[tail] = particleSensor.getFIFORed();
        tail++;

        // Se in modalità doppio LED, salva anche IR
        if (ledMode == 2) {
            ledVals[tail] = particleSensor.getFIFOIR();
            tail++;
        }

        particleSensor.nextSample();
    }
}

uint32_t rValue;
uint32_t irValue;
uint32_t red_value_splitter = 0;
uint32_t ir_value_splitter = 0;

void sendData() {
    //Initially, send a zero value for each led to differentiate between acquisition c
    Serial.write((const uint8_t*)"R:", 2);

```

```

Serial.write((uint8_t*)&red_value_splitter, sizeof(red_value_splitter));
if (ledMode == 2) {
    Serial.write((const uint8_t*)"I:", 3);
    Serial.write((uint8_t*)&ir_value_splitter, sizeof(ir_value_splitter));
}
Serial.write((const uint8_t*)";", 1); // Invia il terminatore ";"

for (int j = 0; j < tail; j++) {
    rValue = ledVals[j];
    Serial.write((const uint8_t*)"R:", 2); // Invia "R:"
    Serial.write((uint8_t*)&rValue, sizeof(rValue)); // Invia valore binario Red

    if (ledMode == 2 && j + 1 < tail) {
        j++;
        irValue = ledVals[j];
        Serial.write((const uint8_t*)"I:", 3); // Invia "I:"
        Serial.write((uint8_t*)&irValue, sizeof(irValue)); // Invia valore binario IR
    }

    Serial.write((const uint8_t*)";", 1); // Invia il terminatore ";"
}

tail = 0;
}

```

Loop cycle for the check() logic

```

void loop() {

newSamples = particleSensor.check();
if (newSamples > 0) {
    for (int i = 0; i < newSamples; i++) {
        if (tail + (ledMode == 2 ? 2 : 1) > array_size) {

            sendData();

        }
    }
}

```

```

    // Salva i dati dal LED rosso
    ledVals[tail] = particleSensor.getFIFORed();
    tail++;

    // Se in modalità doppio LED, salva anche IR
    if (ledMode == 2) {
        ledVals[tail] = particleSensor.getFIFOIR();
        tail++;
    }

    particleSensor.nextSample();
}
}
}

```

Python client

```

import serial
import struct
import csv
import re
import argparse

def read_and_store_data(ser, filename):
    # Open the file in text write mode
    with open(filename, "w", newline='') as csvfile:
        csvwriter = csv.writer(csvfile)
        csvwriter.writerow(["R:", "Value", "I:", "Value"]) # Write header row

    buffer = b'' # Use bytes buffer to handle binary data

    while True:
        if ser.in_waiting > 0:
            data = ser.read(ser.in_waiting)
            buffer += data

```

```

# Process complete entries
while b";" in buffer:
    entry, buffer = buffer.split(b";", 1)

    # Match and process R:uint32_t,IR:uint32_t
    match_full = re.match(rb"R:(.{4}),I:(.{4})", entry)
    if match_full:
        r_value_bytes = match_full.group(1)
        ir_value_bytes = match_full.group(2)
        # Decode uint32_t
        r_value = struct.unpack('I', r_value_bytes)[0]
        # Decode uint32_t
        ir_value = struct.unpack('I', ir_value_bytes)[0]
        csvwriter.writerow(["R:", r_value, "I:", ir_value])
        print(f"Received and stored: R:{r_value}, I:{ir_value}")
        continue

    # Match and process R:uint32_t
    match_single = re.match(rb"R:(.{4})", entry)
    if match_single:
        r_value_bytes = match_single.group(1)
        # Decode uint32_t
        r_value = struct.unpack('I', r_value_bytes)[0]
        csvwriter.writerow(["R:", r_value, "I:", ""])
        print(f"Received and stored: R:{r_value}")
        continue

if __name__ == "__main__":
    # Set up command line argument parsing
    parser = argparse.ArgumentParser(description="Read and store data from Arduino via
    serial port.")
    parser.add_argument("filename", help="The name of the file to store the data.")

```

```
# Parse command line arguments
args = parser.parse_args()

# Change 'COM3' to your serial port (e.g., '/dev/ttyACMO' on Linux)
ser = serial.Serial(port='COM4', baudrate=9600, timeout=1)

try:
    read_and_store_data(ser, args.filename)
except KeyboardInterrupt:
    print("Program interrupted. Closing serial port.")
finally:
    ser.close()
```

