HAMM-LIPPSTADT
UNIVERSITY OF APPLIED SCIENCES

Multi Parameter Health Monitoring System on FPGA

PROJECT WORK

Submitted by
SAIKOT DAS JOY

Bachelor of Engineering
Mat.Nr.: 2190340

15th July 2023

Supervisor: Prof. Dr. Hayek Ali

0.1 Abstract

This project documentation presents an advanced real-time health monitoring system that employs state-of-the-art technology to accurately measure heart rate(pulse), oxygen saturation, and body temperature. The core of the system revolves around the Nexys A7-100T FPGA development board, integrating a SpO2 & heart rate sensor, infrared temperature sensor, wifi module, and OLED display. The data processing is handled by the microblaze processor, enabling the information to be displayed on the OLED screen and transmitted to a cloud server for remote monitoring. The development of this system utilized Xilinx Vivado and Vitis tools, ensuring the incorporation of cutting-edge design and programming methodologies.

Contents

1 Introduction

Health monitoring systems have become an integral aspect in the healthcare industry, providing individuals with the ability to closely track and monitor their physiological parameters in real time. With the advancement of technology, these systems have become increasingly accessible and provide a powerful tool for individuals to take control of their health and well-being. This project aims to contribute to this growing field by designing and implementing a health monitoring system that accurately measures pulse, oxygen saturation, and body temperature. The system utilizes advanced sensors and processing technology, ensuring highly accurate and reliable data. The integration of a user-friendly OLED display and the ability to send data to a cloud server for storage and analysis make this system a complete solution for health monitoring. This project will provide valuable insights and a comprehensive guide to anyone interested in the field of health monitoring.
The heart of this real-time health monitoring system lies in the MicroBlaze processor, seamlessly integrated within the Nexys A7-100T development board. This processor plays a vital role in efficiently processing the data from the sensors, making it possible to monitor physiological parameters in real-time. The use of Xilinx Vivado for hardware design and Xilinx Vitis for software design provides a highly customizable and adaptable solution, allowing for further advancements and modifications. The Nexys A7-100T board serves as the ideal platform to emulate the system, ensuring its robust and reliable performance.
In the quest to provide a comprehensive solution, PmodESP32 WiFi module is also integrated into the system. This allows for seamless data transfer to a cloud server for storage and further analysis, so that users can access their health information from anywhere at any time.
To enhance the user experience,an OLED display is also incorporated into the system that serves as a user-friendly interface for real-time monitoring. The display provides clear and concise information about the user's physiological parameters, making it easy to understand and use.
The design and implementation of this health monitoring system were carried out using the latest technology, Xilinx Vivado and Vitis, ensuring a highly flexible and customizable solution. This documentation provides a comprehensive overview of the design process, from the initial hardware and software setup to the final implementation of the system. It includes a step-by-step guide to the implementation process, making it an invaluable resource for anyone looking to develop their own health monitoring system. Additionally, the document provides an in-depth analysis of the hardware and software design, including a description of the MicroBlaze processor, AXI IIC IP, Pmod bridge IP, AXI UARTlite IP, PmodOLEDrgb IP, and other critical components. This document is aimed at providing a comprehensive guide to the design and implementation of a health monitoring system, covering all aspects of the system, from the hardware design to the software implementation. Whether you are a student, researcher, or engineer in the field of health monitoring, this document is a valuable resource that will help you understand the underlying principles

and design considerations involved in creating a health monitoring system. Fig 1.1 presents an overall concept of the project in a single block diagram, encapsulating the essence of this health monitoring system and its seamless integration with technology.
In conclusion, this innovative and comprehensive health monitoring system, designed to empower individuals to take charge of their health with its seamless integration of technology and real-time monitoring capabilities, it represents a new era in healthcare and well-being. Get ready to experience the future of health monitoring with this state-of-the-art solution.

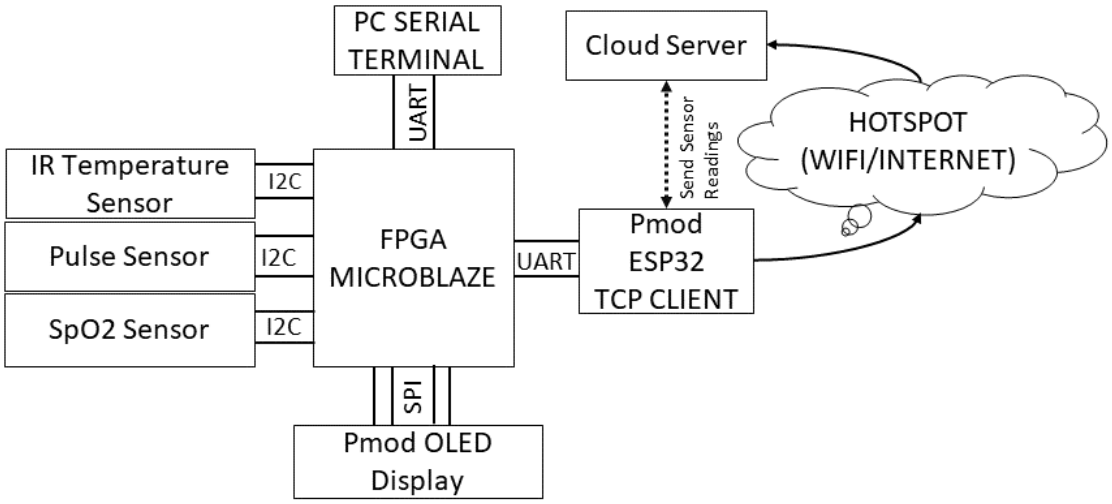

Figure 1.1: Concept Description Block Diagram

2 Fundamentals

2.1 Communication Protocols

An information flow between devices or systems through a communication channel is governed by a set of rules and conventions called a communication protocol. Communication protocols can be used to enable communication between objects or systems that are physically close to one another or to enable communication between objects or systems over large distances, such as the internet. Communication protocols come in a wide variety and are each designed to address a particular set of requirements. The types of data that can be transferred, how it is formatted and encoded, the strategies employed to guarantee data transmission reliability and accuracy, and the systems for error detection and correction are some common characteristics of communication protocols. In embedded systems, communication protocols are frequently employed to enable interaction between various components and with outside devices. Using a communication protocol, for instance, a microcontroller may send data to a sensor or receive data from a display. Following is a list and description of the communication protocols we used in this project:

2.1.1 Universal Asynchronous Receiver/Transmitter (UART)

A form of communication protocol called UART (Universal Asynchronous Receiver/Transmitter) enables devices to send and receive data asynchronously, i.e. without the use of a fixed clock signal. UART is frequently used in embedded systems to enable communication between components and with outside devices like computers.
In a UART communication, data is delivered and received as serial bits, with each bit denoting a single data element (such as a letter or a number). The timing and format of the data bits, as well as the methods for sending and receiving data, are all specified by the UART protocol.[NP16]
Two wires are normally needed for UART communication: one for sending data and one for receiving it. To regulate the data flow, some UART devices could also have extra control signals including flow control and interrupt signals.
A popular and widely used protocol, UART is frequently combined with others to speed up device connection, including TCP/IP and USB.
In our project, All measured data is sent to the PmodESP32 module with UART protocol.UART protocol is also used to establish a connection between the computer and the Nexys A7-100T board.In Fig 2.1 , commmunication between two devices through UART protocol is shown in block diagram.

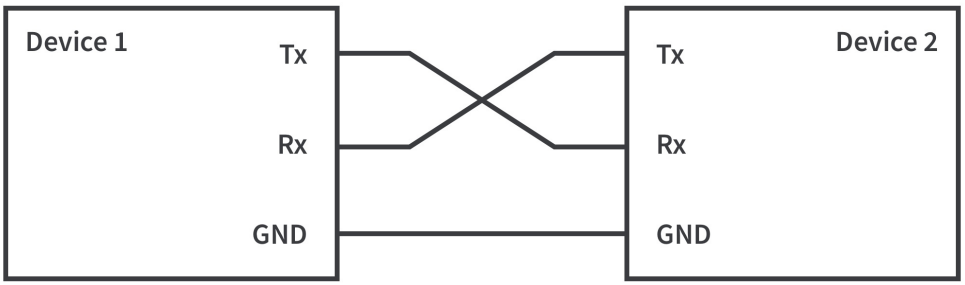

Figure 2.1: UART Communication

2.1.2 Inter-Integrated Circuit(I2C)

I2C (Inter-Integrated Circuit) is a communication protocol that allows devices to exchange data over a two-wire bus.It is frequently used in embedded systems to let devices communicate with one another and with external devices like computers. Each bit in an I2C transmission represents a single data element, and data is transmitted and received in the form of serial bits (such as a letter or a number). The I2C protocol defines the methods for transmitting and receiving data, as well as the time and format of the data bits. Two wires are needed for I2C communication: one for the clock signal (SCL) and one for the data signal (SDA). Each device on the I2C bus has a specific address, and when communicating, the devices send and receive data to and from these addresses.[LCK18] I2C is a popular and easy-to-use protocol that is frequently combined with others, like TCP/IP and USB, to help devices communicate with one another. The microblaze processor in our project uses the I2C communication protocol to receive data from the Pulse & Oxygen saturation sensor and same communication protocol is also used to receive data from IR temperature sensor. Block diagram representation of I2C communication is shown in in Fig 2.2

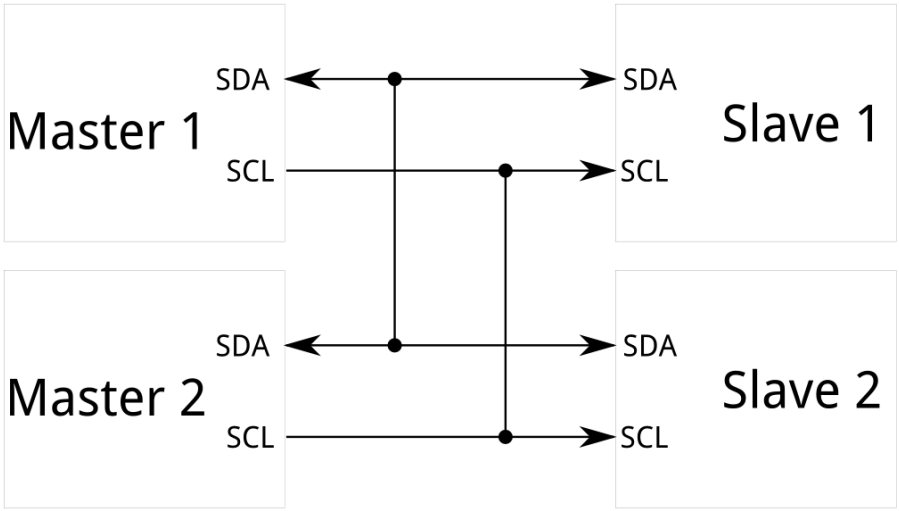

Figure 2.2: I2C Communication

2.1.3 Serial Peripheral Interface(SPI)

SPI (Serial Peripheral Interface) is a full-duplex synchronous serial communication protocol that is widely used for communication between microcontrollers, microprocessors, and other digital devices. It provides a simple and efficient way to transfer data between devices, allowing them to exchange information in real-time.
In SPI communication, there is one master device and one or more slave devices. The master device controls the communication and initiates data transfers, while the slave devices respond to the master's requests. The master device generates a clock signal, which is used to synchronize the data transfer between the devices.
The main components of the SPI communication protocol are:

- Master Out Slave In (MOSI) : This line is used by the master device to send data to the slave devices.
- Master In Slave Out (MISO) : This line is used by the slave devices to send data to the master device.
- Serial Clock (SCLK) : This is a clock signal generated by the master device and used to synchronize the data transfer. The clock rate determines the speed of the data transfer.
- Slave Select (SS) : This line is used by the master device to select the specific slave device that it wants to communicate with.

The SPI communication protocol uses a communication pattern known as "shifting out and shifting in". In this pattern, the master device sends a byte of data on the MOSI line and simultaneously receives a byte of data on the MISO line. This process is repeated for each byte of data that needs to be transferred.[Lee09]
One of the advantages of the SPI communication protocol is its simplicity. The protocol requires only a few lines to transfer data, which makes it easy to implement and use. Additionally, the full-duplex nature of the protocol allows data to be sent and received at the same time, which results in fast data transfer speeds.
Another advantage of SPI is its versatility. The protocol can be used with a variety of devices, including microcontrollers, sensors, and displays, and it can be used in a variety of applications, such as data acquisition, control systems, and communication networks.
Fig 2.3 shows the SPI communication between a master device and three slave device in a block diagram format.

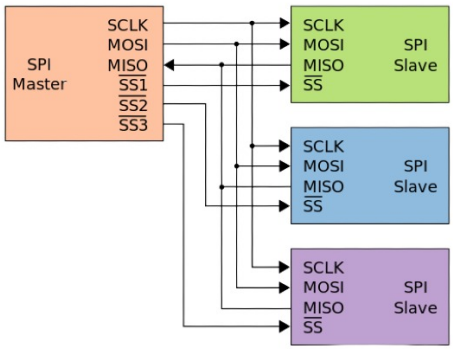

Figure 2.3: SPI Communication

2.1.4 Transmission Control Protocol (TCP)

TCP (Transmission Control Protocol) is a communication protocol that is used to transmit data over the internet and other networks. It is a transport layer protocol, which means that it resides in the transport layer of the Internet Protocol Suite (often referred to as the TCP/IP architecture). With the help of TCP, two systems or devices can connect in a dependable, stream-oriented manner. This is accomplished by establishing a connection between the two devices, segmenting the data that is being transmitted, and transferring the segmented data to the target device. The final step is for the destination device to recombine the segments back into the initial data stream after acknowledging receipt of them. TCP can request that any missing or damaged segments be sent again, ensuring that the data is carried accurately and entirely. TCP is frequently used in systems that demand a dependable connection, including file transfers, email, and web browsing. It frequently works in tandem with Internet Protocol (IP), which is in charge of directing the data packets over the network. In this project, the data is transferred from Microblaze to the ESP32 to Thingspeak cloud server using this protocol. [LLY07]

2.2 Vivado Design Suite

Vivado Design Suite is a comprehensive and powerful software tool suite for designing and implementing digital logic circuits on FPGA (Field-Programmable Gate Array) devices. It was created by Xilinx, a top supplier of FPGA technology, and is widely utilized by experts in the digital design and embedded systems industries.To generate, implement, and debug digital designs for FPGAs, designers can use a variety of tools and functionalities in the Vivado Design Suite. These tools consist of a graphical design environment for constructing and configuring digital circuits, a synthesis tool for transforming the design into an FPGA-implementable format, and a place-and-route tool for placing the design on the unique resources of an FPGA device. The suite also includes a number of IP (Intellectual Property) building blocks that are simple to incorporate into a design, along with simulation and verification tools.The Vivado Design Suite also provides a variety of other features and tools to support the design process in addition to these basic tools, such as support for high-level design languages, reuse and version control for designs, and integration with external tools and flows. We utilized Vivado Design Suite to create the hardware part of our project.[Viva]

2.3 Vitis Unified Software Platform

Vitis is a platform developed by Xilinx for developing and deploying high-performance, software-defined systems on Xilinx devices.It is a comprehensive and efficient set of tools and libraries that can be used to develop, implement, and optimize systems for a variety of applications, including computer vision, machine learning (ML), artificial intelligence (AI), and more. Developers can build and deploy software-defined systems on Xilinx hardware with the aid of a variety of tools and resources available on the Vitis platform. These technologies include the Vitis Integrated Environment, a software package for creating digital logic circuits on FPGA (Field-Programmable Gate Array) devices, and the Vitis AI libraries, a collection of streamlined libraries and tools for creating AI

and ML applications. Additionally, the platform offers a variety of tools for high-level synthesis and hardware-software co-design to improve system performance. [Vivb] The Vitis platform also includes a number of extra features and resources to support the creation and deployment of software-defined systems, such as support for a variety of programming languages and frameworks, integration with third-party tools and libraries, and a variety of design examples and tutorials. These extra features and resources are in addition to the platform's core tools and resources.In our project, we used this platform to integrate software into the hardware that was created and to validate the design uploading to an FPGA board (Nexys A7-100T)

2.4 FPGA (Field-Programmable Gate Array)

FPGA (Field-Programmable Gate Array) is a type of digital integrated circuit that can be programmed after manufacturing to implement custom logic functions. Unlike microcontrollers and microprocessors, which have a fixed set of instructions hardcoded in the chip, an FPGA can be reconfigured to perform various tasks, making it highly flexible and adaptable for different applications.
FPGAs consist of an array of configurable logic blocks (CLBs) and programmable interconnects. The CLBs can be programmed to perform various Boolean operations, and the interconnects allow the CLBs to be connected in various configurations to implement custom logic functions.

2.4.1 Nexys A7-100T FPGA Board

The Nexys A7-100T is a FPGA (Field-Programmable Gate Array) development board produced by Digilent. It is based on the Xilinx Artix-7 FPGA, which is a mid-range FPGA device that is suitable for a wide range of applications.The Nexys A7-100T development board is designed to be a flexible and effective platform for exploring and analyzing embedded systems and digital design.[Nex] It has a variety of features and add-ons to assist customers in getting started with FPGA development, including a microSD card slot for data storage, an HDMI interface for video output, and a number of expansion ports.The Vivado Design Suite from Xilinx, which may be used to design and construct digital circuits on the FPGA, is one of many software tools that are supported by the Nexys A7-100T development board. Additionally, a number of libraries and programming languages, including as C/C++, Python, and VHDL (VHSIC Hardware Description Language), support it, enabling users to develop original ideas and applications for the board.To upload the design and review it, we used this board for our project. Additionally, we used this board's on-board temperature sensor. A general image of this board is shown in Fig 2.4

2.5 Pmod ESP32

The Digilent Pmod ESP32 is a Peripheral Module (Pmod) that enables quickly integrate Wi-Fi connection into the designs. It is based on the ESP32 microcontroller, a high-performance, low-power microcontroller with built-in Bluetooth and Wi-Fi.The 6-pin

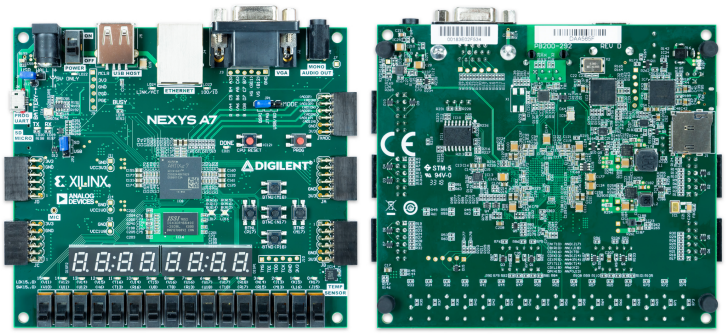

Figure 2.4: NEXYS A7-100T FPGA Board

Pmod connector on the Pmod ESP32 can be used to connect it to a host board, and it is designed to be readily integrated into a variety of designs. In order to support Wi-Fi networking, it has a variety of features and add-ons, including an antenna, a power amplifier, and a low-noise amplifier. The Pmod ESP32 also works with a variety of libraries and programming languages, including as Arduino, Python, and C/C++, enabling users to quickly develop unique applications for their creations.[Pmo]The Pmod ESP32 can be used with a variety of host boards, including the Nexys A7-100T FPGA development board.The Pmod ESP32 should be connected to the relevant Pmod connectors on the Nexys A7-100T board before being used with the board. The ESP32 microcontroller will then be used to implement Wi-Fi connection in a digital design. The design can then be implemented on the FPGA and tested using the Vivado Design Suite's tools, and the Pmod ESP32's Wi-Fi capabilities can then be utilised.In our project, we attached this module to the Nexys A7-100T board's peripheral. For our design, we also used Pmod ESP32 IP. In Fig 2.5, a general picture of Pmod ESP32 is shown.

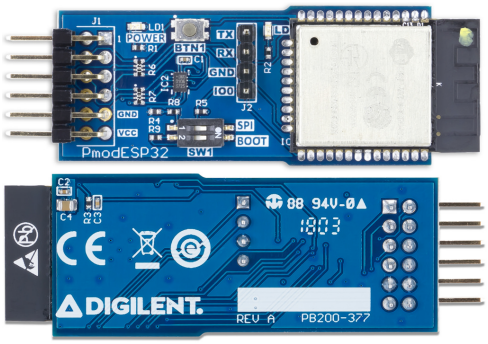

Figure 2.5: Pmod ESP32

- AT commands are used to control a modem, and are generally used to set up and configure a wireless modem. Here are a few common AT commands used with PMOD ESP32:
  - AT+CWJAP=<ssid>,<pwd>: Connect to a WiFi network where, ssid = wifi network name , pwd = wifi network password
  - AT+CIPSTART=<type>, <addr>,<port>: Start a connection over TCP or UDP

– AT+CIPSEND=<length>: Send data over the connection

2.6 Pmod OLEDrgb

The Pmod OLEDrgb is a peripheral module designed to provide a low-power, high-contrast OLED display. The module is based on the OLED (Organic Light Emitting Diode) technology, which offers a number of advantages over traditional LED displays, including high contrast, wide viewing angles, and fast refresh rates. The OLEDrgb module provides a high-resolution 96x64 pixel display that is capable of displaying text and graphics in full color.
The Pmod OLEDrgb interface to a microcontroller or microprocessor using a standard 6-pin Pmod interface, which allows for fast and easy integration into a variety of systems. The module uses a serial peripheral interface (SPI) to communicate with the microcontroller or microprocessor, and supports a variety of color depths and refresh rates.[OLE]
The Pmod OLEDrgb can be used in a wide range of applications, including portable devices, embedded systems, and scientific instruments. The module is small and lightweight, making it ideal for space-constrained systems, and it consumes very little power, making it suitable for battery-powered applications. Fig 2.6 shows a pictuere of Pmod OLEDrgb from DIGILENT.
Here are the pin descriptions for the Pmod OLEDrgb:

VCC: This the power supply pin, typically connected to a voltage source between 3.3V to 5V.
GND: This is the ground pin.
SCK: This is the serial clock pin, used to synchronize the data transfer between the OLEDrgb module and the microcontroller or microprocessor.
MOSI: This is the master out, slave in pin, used for transmitting data from the microcontroller or microprocessor to the OLEDrgb module.
CS: This is the chip select pin, used to select the OLEDrgb module for communication.
RES: This is the reset pin, used to reset the OLEDrgb module.

The Pmod OLEDrgb is a versatile and high-performance display solution that is easy to use and integrate into a variety of systems. With its high-resolution OLED display, fast refresh rate, and low power consumption, the Pmod OLEDrgb is an ideal choice for a wide range of applications.

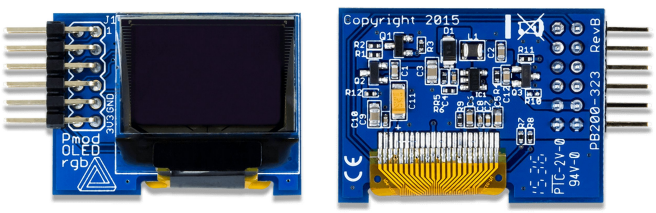

Figure 2.6: Pmod OLEDrgb

2.7 Sensors

2.7.1 GY-906 Infrared-Temperature sensor MLX90614

The GY-906 Infrared-Temperature Sensor MLX90614 is a non-contact temperature sensor that is designed to measure the temperature of objects and surfaces. The device uses infrared technology to detect the temperature of an object by measuring the emitted infrared radiation.[tem]
The MLX90614 sensor consists of a thermopile that detects the infrared radiation and a signal processing unit that converts the signal into a temperature reading. The sensor operates by emitting a beam of infrared radiation towards the object, and then measuring the amount of radiation that is reflected back. The signal processing unit then uses this information to calculate the temperature of the object.
The MLX90614 is particularly well-suited for measuring the temperature of human bodies, as it can accurately measure the temperature of skin and other surfaces even when they are not in direct contact with the sensor. This makes it a useful tool for medical and health applications, such as monitoring the temperature of a person for signs of fever or illness.
To interface the MLX90614 with a microcontroller or microprocessor, the device uses a two-wire I2C communication interface. The microcontroller or microprocessor sends a request for a temperature reading to the MLX90614, and the sensor sends back the temperature measurement.
Overall, the GY-906 Infrared-Temperature Sensor MLX90614 is a compact, low-power, and highly accurate solution for temperature sensing applications. Its ability to measure temperature without making physical contact with the object makes it a versatile tool for a wide range of applications, including monitoring human body temperature.GY-906 Infrared-Temperature sensor MLX90614 is shown in Fig. 2.7
The pin descriptions for the GY-906 Infrared-Temperature Sensor MLX90614:

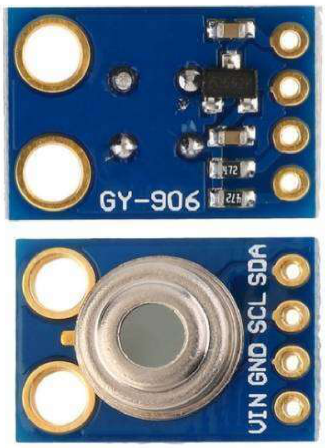

Figure 2.7: GY-906 Infrared-Temperature sensor (MLX90614)

VCC: This is the power supply pin, typically connected to a voltage source between 3.3V to 5V.
GND: This is the ground pin.

SDA: This is the serial data pin, used for communication between the sensor and the microcontroller or microprocessor.
SCL: This is the serial clock pin, used for synchronizing the data transfer between the sensor and the microcontroller or microprocessor.
ADDR: This is an address pin, used to set the I2C address of the sensor.
SMBUS ALERT: This is an optional pin, which can be used to signal the microcontroller or microprocessor when a new temperature measurement is available.

It's important to note that the MLX90614 is designed to be used with microcontrollers and microprocessors that support I2C communication protocols. By connecting the sensor to a microcontroller or microprocessor and writing appropriate software, it is possible to interface with the MLX90614 and use it to measure the temperature of objects and surfaces.

2.7.2 Oxygen saturation and Pulse sensor(MAX30102)

The MAX30102 from Hailege is a highly integrated pulse oximetry and heart-rate sensor module that provides a convenient solution for monitoring both pulse rate and oxygen saturation. This device operates by utilizing light-emitting diodes (LEDs) and photodiodes to detect changes in blood volume in the microvascular bed of tissue.
The MAX30102 has two LEDs, one emitting red light and the other emitting infrared light. These LEDs emit light through the skin and into the blood vessels, where the light is absorbed by the blood. The photodiodes in the device detect this light and convert it into electrical signals, which are then processed by the device to determine both pulse rate and oxygen saturation.[oxy]
To measure oxygen saturation, the MAX30102 calculates the ratio of red light absorbed to infrared light absorbed, with the results being used to calculate the oxygen saturation (SpO2) of the blood. The device uses a sophisticated algorithm to perform this calculation, ensuring highly accurate results.
For heart-rate measurement, the photodiodes detect changes in blood volume with each heartbeat, which are then used to calculate heart rate. The MAX30102 has a digital interface for communication with a microcontroller or microprocessor, allowing for seamless integration into a wider system. The device sends its readings to the microcontroller or microprocessor, which can then process and display the information.
In conclusion, the MAX30102 is a compact, low-power, and highly accurate solution for pulse oximetry and heart-rate monitoring applications. Its widespread use in wearable devices, fitness trackers, and medical equipment attests to its versatility and effectiveness.
Fig 2.8 shows the picture of a MAX30102 pulse oximetry and heart-rate sensor from the provider "the Hailege". MAX30102 Heart Rate and oxymetry Sensor Module from Hailege

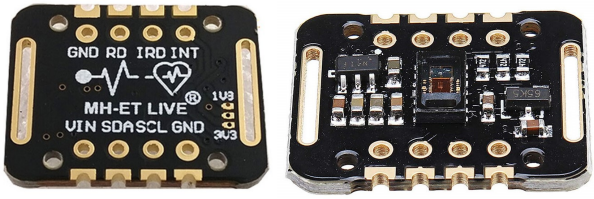

Figure 2.8: MAX30102 pulse oximetry and heart-rate sensor

can be easily integrated into a system that uses a microprocessor(i.e. Microblaze) or microcontroller.
Here are the pin descriptions for the device:

VCC: This is the power supply pin, typically connected to a voltage source between 2.7V to 3.6V.
GND: This is the ground pin.
SDA: This is the serial data pin, used for communication between the sensor and the Microblaze processor.
SCL: This is the serial clock pin, used for synchronizing the data transfer between the sensor and the Microblaze processor.
INT: This is the interrupt pin, which can be used to signal the Microblaze processor when a new data sample is available.
LED+: This is the positive terminal of the red and infrared LEDs.
LED-: This is the negative terminal of the red and infrared LEDs.
PD+: This is the positive terminal of the photodiodes, used to detect the light emitted by the LEDs.
PD-: This is the negative terminal of the photodiodes.

In order to interface the MAX30102 with the Microblaze processor, the necessary I2C communication protocols need to be implemented in software. The Interrupt line can be used to trigger the Microblaze processor when new data is available, allowing the processor to retrieve the latest readings from the sensor.
By utilizing the MAX30102 with a Microblaze processor, a high-performance system for monitoring heart rate and oxygen saturation can be easily developed while the MAX30102 provides highly accurate readings of heart rate and oxygen saturation.

2.8 Microblaze Processor

Microblaze is a soft processor core that can be implemented in an FPGA (Field-Programmable Gate Array) and configured to meet the specific needs of a particular design.It was created by Xilinx and is frequently used in embedded systems to build specialized microcontroller-based systems and manage the different peripherals and devices attached to the FPGA.Microblaze is a RISC (Reduced Instruction Set Computing) processor designed specifically for FPGAs. Its small size, light weight, and low resource utilization make it an excellent choice for designs with limited resources. To satisfy the unique requirements of a particular design, Microblaze can be built to support a variety of instruction set architectures (ISAs) and be customized with a variety of peripherals and memory interfaces.[Mic]The Microblaze processor and any other required peripherals and devices must be included in the digital design in order to utilize Microblaze in a design. The design can then be implemented on an FPGA using the Vivado Design Suite tools or other digital design tools, and the performance of the Microblaze processor and the customized embedded system can then be tested. Internet of things (IoT) systems, industrial automation, embedded control, and other applications all frequently use Microblaze. In figure 2.9, a core block diagram of Microblaze Processor is given.

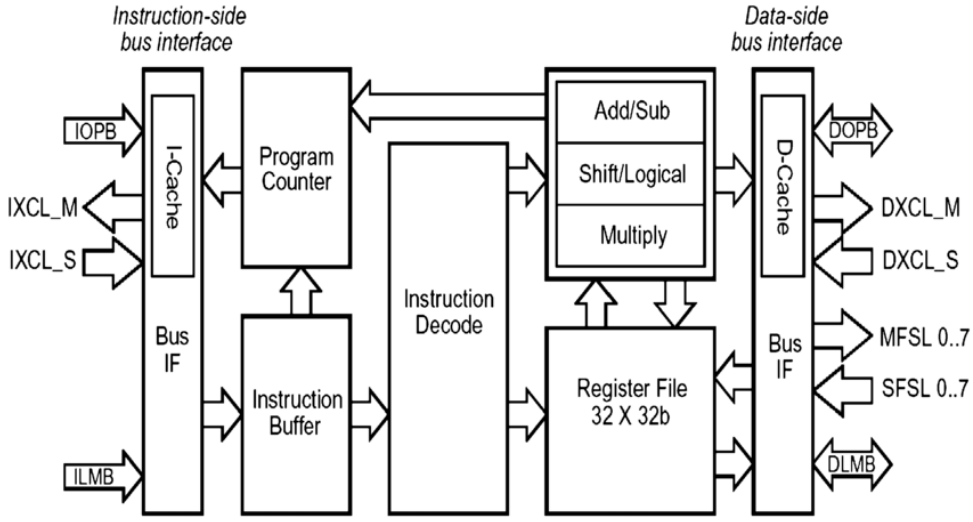

Figure 2.9: Microblaze Core Block Diagram

2.9 Intellectual Property (IP)

In Vivado, IP (Intellectual Property) refers to a pre-designed, pre-verified functional block that is simple to incorporate into an FPGA design. When implementing common functionalities like memory interfaces, processors, and communication protocols, IP blocks are frequently utilized. These blocks can be altered to match the unique requirements of a particular design.By allowing designers to reuse functional blocks that have already been tried and tested rather than having to create and validate these blocks from scratch, using IP blocks can accelerate the design process. A large range of pre-made IP blocks are available in Vivado, and designers can also buy or request the creation of unique IP blocks from outside vendors.[IP]
Before employing IP blocks in Vivado, designers must first add them to their designs using the "Create and Package IP" flow or by dragging and dropping IP blocks from the IP catalog. After then, the designer can modify the IP block to suit the particular requirements of the design, such as choosing the proper interface standards or adjusting the memory amount.The IP block can then be integrated into the broader FPGA implementation by connecting to other blocks in the design. In our project, we modified and properly connected already-existing IPs to design the hardware component.

3 Related Work

The related works section of this project includes various studies on IoT-based patient monitoring systems. In April 2020, A. Athira, T.D. Devika, K.R. Varsha, and Sree Sanjanaa Bose S. designed and developed an IoT-based multi-parameter patient monitoring system. Their system included sensors for heart rate, respiration rate, oxygen saturation, and temperature, and sent an email to the patient's guardian in case of an emergency. The project was based on Arduino Uno microcontroller.[ADVBS20]
Similarly, in April 2022, Mohammad Monirujjaman Khan, Turki M. Alanazi, Amani Abdulrahman Albraikan, and Faris A. Almalki developed a health monitoring system using Arduino and IoT. Their system monitored body temperature, oxygen saturation, and heartbeat, and sent the data to a mobile application over Bluetooth. [KAAA22]
In 2015, R. Kumar and M. Pallikonda Rajasekaran published their work on an IoT-based patient monitoring system using Raspberry Pi. Their system monitored body temperature, respiration rate, heart rate, and body movement using the Raspberry Pi board.[KR16]
In 2018, Mohammad Salah Uddin, Jannat Binta Alam, and Suraiya Banu proposed a real-time patient monitoring system based on IoT. Their system used sensors to automatically monitor patients' health conditions and forwarded the information to the IoT cloud.[UAB17]
In 2019, Sambit Satpathy, Prakash Mohan, Sanchali Das, and Swapan Debbarma presented a new healthcare diagnosis system that used an IoT-based fuzzy classifier with FPGA. Their IoT-based analysis system could alert the user when any of their health parameters were above or below the normal range. The data collected from the system were uploaded to the cloud via a mobile application and then transferred to the field-programmable gate.[SMDD20]
However, it is worth noting that not much work has been done on designing an IoT-based health monitoring system using Microblaze processor. This presents an opportunity to develop a more flexible, scalable, and reliable real-time health monitoring system that can be easily upgraded to different versions.
By leveraging the capabilities of Microblaze processor, it is possible to develop an IoT-based patient monitoring system on FPGA that is more efficient in terms of resource utilization and processing speed. This can ultimately lead to a more accurate and responsive system that can provide better health outcomes for patients. Therefore, there is a need for more research in this area to explore the potential of Microblaze processor in developing IoT-based patient monitoring systems.

4 Analysis

4.1 System requirements

A Requirement Diagram is a delightful illustration of the must-haves and conditions that a system must adhere to, which encompasses both its functional and non-functional requirements. These requirements are presented as charming rectangles with the keyword "requirement" and can be further segregated into fascinating categories such as performance, interface, design constraint, and physical requirements. This diagram serves the purpose of outlining the relationships between these requirements and other elements within the system in a clear and concise manner, including model components that bring the requirements to life and test cases that validate their effectiveness. By utilizing this diagram, one can guarantee that all crucial requirements are properly specified and tracked throughout the development journey. Requirements for the multi-parameter health monitoring system is noted in Fig 4.1. The following requirements were included:

Requirement 1.1: Measure body temperature - the system should be capable of accurately measuring human body temperature.
Requirement 1.2: Measure heart rate - the system should be capable of accurately measuring human heart rate (Pulse).
Requirement 1.3: Measure oxygen saturation data - the system should be capable of accurately measuring human blood oxygen saturation.
Requirement 1.4: Display data - the system should be capable of displaying all the measured data in an OLED display in a user-friendly manner.
Requirement 1.5: Data transmission - all the data should be transmitted to a cloud server for further visualization and analysis.
Each requirement is assigned a unique requirement ID for easy reference and tracking.

4.2 Main system use cases

A use case diagram is a visual representation of the functional requirements of a system. It is used to capture the interactions between a system and its actors in terms of a set of use cases. Use case diagrams are commonly used in modeling to communicate the high-level requirements of a system to stakeholders.
In a UML use case diagram, the main elements include:
Actors: Actors represent the entities that interact with the system, such as people, organizations, or other components. Actors are represented as stick figures or simple icons.
Use cases: Use cases are the individual actions or functions that a system can perform, such as processing a transaction, updating a record, or generating a report. Use cases are represented as ellipses.

Associations: Associations are the relationships between actors and use cases. They represent the communication between the entities and the system. Associations are represented as arrows.
System boundary: The system boundary represents the scope of the system being modeled and defines the limit of the use case diagram.
use case for this multiparameter health monitoring system in well captured in the use case diagram in Fig. 4.4 Fig 4.4 represents the interactions between the actors and the system in a healthcare monitoring system. The actors in the system are the patient/nurse and doctor.
The use cases include:
Attach Sensor: Both the patient and nurse can attach the oxygen saturation and heart rate sensors on the patient's finger and the IR temperature sensor on the ear.
View Patient's Vital Signs: The doctor can view the patient's oxygen saturation level, pulse, and heart rate on a cloud server.
Display Vital Signs: The doctor, nurse, or patient can view the patient's vital signs on an OLED display.
Send Data to Cloud Server: The system can send the patient's vital signs data to the cloud server.
Process Sensor Data: The system can process the sensor data to provide accurate vital signs measurements.

4.3 Main system states

UML (Unified Modeling Language) State Machine diagrams, also known as Statechart diagrams, are used to model the behavior of an object, system, or process as it changes over time in response to events.
A UML state machine diagram consists of: States: Represent the conditions or phases that the object, system, or process can be in.
Transitions: Represent the events or triggers that cause the object, system, or process to move from one state to another. Transitions are indicated by arrows pointing from the source state to the target state.
Actions: Represent the behavior that occurs when the object, system, or process enters or exits a state, or when a transition occurs. Actions are usually written within the state or transition.
Initial state: Represent the state where the object, system, or process starts.
Final state: Represent the state where the object, system, or process ends.
For this multiparameter health monitoring system, corresponding uml state machine diagram is shown on Fig. 4.2.
The state machine diagram represents the different states and transitions of a healthcare monitoring system. The first state is the Initialization state, where all the sensors are initialized for data reading. Once the initialization is completed, the system moves to the Oxygen Saturation Level Data Collection state, Temperature Data Collection state, and Heart Rate Data Collection state based on the corresponding triggers.
After collecting data from all the sensors, the system moves to the Data Processing state to process the collected data. From there, the system moves to the Display state to display the processed data on an OLED display. Once the data is displayed, the system returns

to the Processing state to continue processing.
Next, the system moves to the Cloud Transmission state to transmit the processed data to a cloud server. If the system is stopped by an actor or any other way, the state machine stops at the Cloud Transmission state. Otherwise, the system returns to the Initialization state to start a new cycle of data collection.

4.4 Main system sequences

UML (Unified Modeling Language) sequence diagrams are a type of UML diagram used to represent and model interactions and the flow of messages, events, and operations among objects or components in a system. They are commonly used in software engineering and systems design to visualize the interactions between objects over time.
A UML sequence diagram is composed of several elements: Lifelines: Represent the objects or components involved in the interaction and their lifecycle. They are represented as vertical lines and are usually labeled with the name of the object or component.
Messages: Represent the interactions between objects or components. Messages can be represented as arrows connecting two lifelines and labeled with the name of the operation or message.
Activation Bars: Represent the duration of an object's processing of a message. They are shown as horizontal bars above a lifeline and indicate the time during which the object is executing an operation.
Combined Fragments: Represent alternative and optional interactions. They are denoted by a rectangle surrounding a group of messages and are often labeled with a keyword such as "alt" (alternative), "opt" (optional), or "loop" (iteration).
Notes: Provide additional information about the diagram or specific elements in the diagram. Notes are represented as rectangular boxes with a pointer arrow and are often used to explain specific interactions or conditions.
the messege passing between different objects in the multiparameter health monitoring system is visualized in the sequence diagram in Fig 4.3

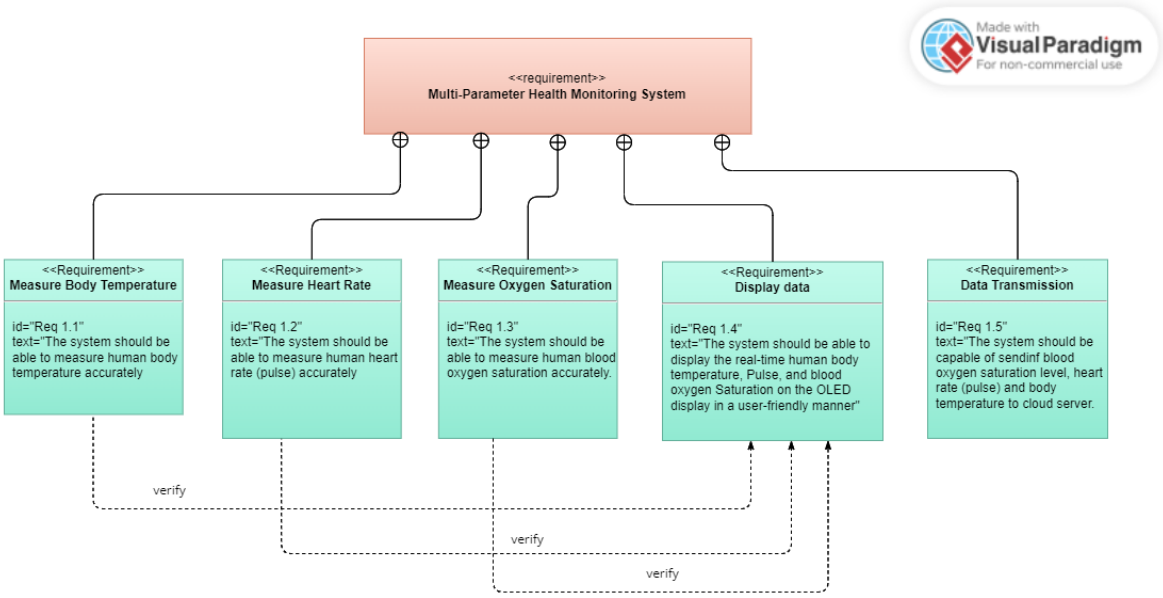

Figure 4.1: Requirement Diagram

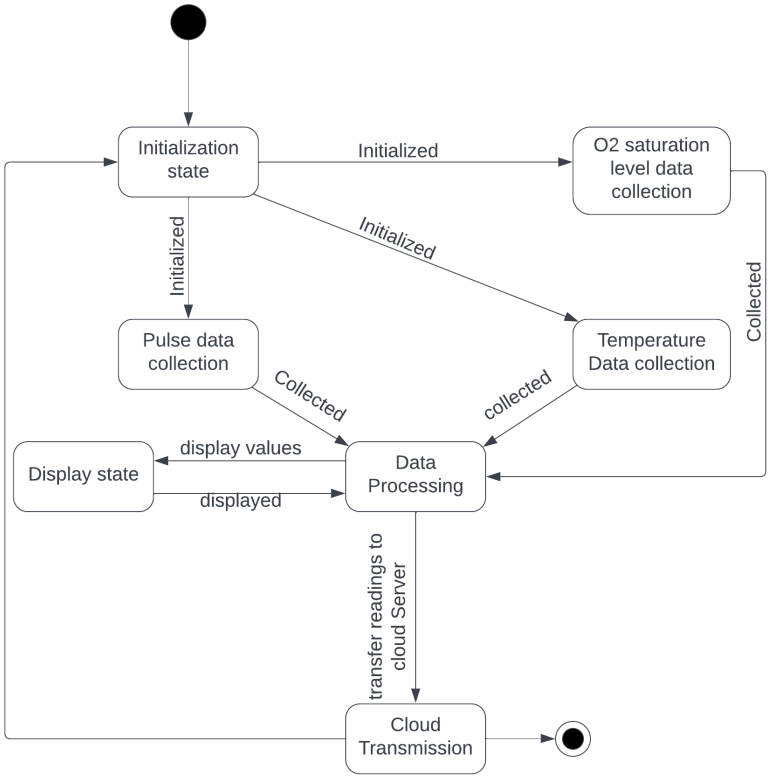

Figure 4.2: State Machine Diagram

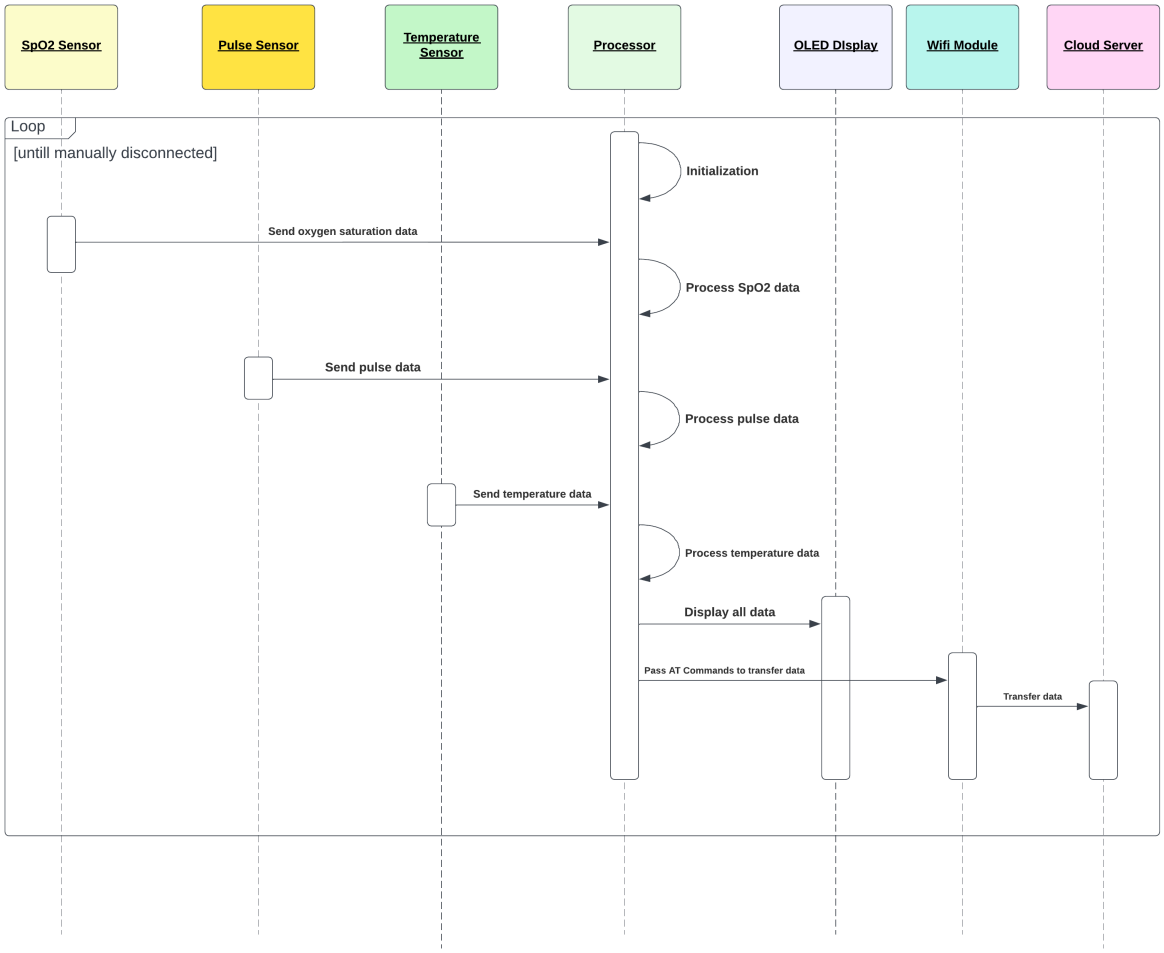

Figure 4.3: Sequence Diagram

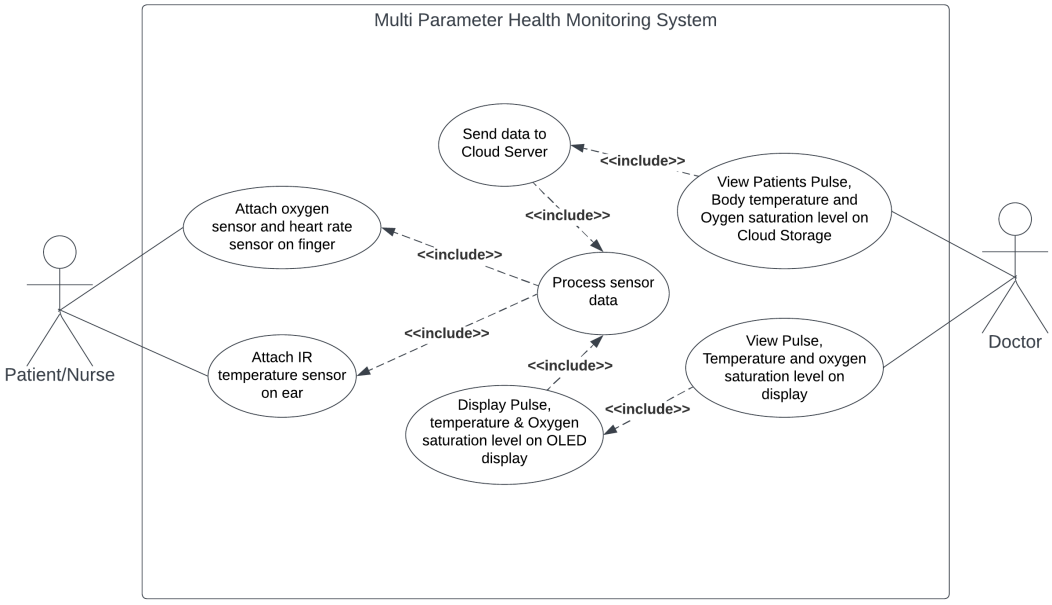

Figure 4.4: Use case diagram

5 Design and Implementation

5.1 Hardware Design

5.1.1 Design Steps

The following steps outline the process for designing the hardware for the Multi-Parameter Health Monitoring System using Xilinx Vivado software:

5.1.1.1 Setting up the Environment:

• Xilinx Vivado 2022.2 is installed and the necessary board files for the Nexys A7-100T board are downloaded.
• The Nexys A7-100T board file is added to Xilinx Vivado and the Vivado library master is added to the Xilinx repository.

5.1.1.2 Creating the Project:

• A new RTL project is opened and the Nexys A7-100T board is selected as the target device.

5.1.1.3 Designing the Block Diagram:

• The IP Integrator is utilized to create a block design for the project.
• A Microblaze IP is added, configured with a local memory of 128 KB and an interrupt controller.
• Block automation is run to interconnect the IP blocks.

5.1.1.4 Configuring the Clocking Wizard:

• Clocking Wizard IP is configured to use the system clock board interface (CLK_IN1) and the reset of the board interface (EXT_RESET_IN).
• In the output clock option, the reset type is selected as active low.

5.1.1.5 Adding Interrupts and Peripheral Devices:

• Concat IP is configured to have 4 ports.
• An AXI Uartlite IP is added and its interrupt port is connected to the Concat IP.
• A PmodOLEDrgb IP is added and connected to the board interface jb.
• A PmodESP32 IP is added, its gpio_interrupt port is connected to the Concat IP, and

the board interface ja is selected.
• Two AXI IIC IPs are added, configured with a clock frequency of 100 KHz and a 7-bit address mode.
• One AXI IIC IP external is named as IR_sensor_jc and another AXI IIC IP external as SpO2_Pulse_Sensor_jd.
• A constraint file is added as shown in Figure 5.4.

5.1.1.6 Running Connection Automation and Re-arranging the Design:

• Connection automation is run to interconnect the IP blocks.
• The automatic re-arrange option is run to make the block design look neat and organized. The final IP block design is displayed in Fig 5.2.

5.1.1.7 Validating and Wrapping the Design:

• The design is validated and it is successful.Fig 5.3 is the validation successful window.
• An HDL wrapper for the design is created, the option to let Vivado manage and auto-update is selected, and the wrapper is named.

5.1.1.8 Synthesis and Implementation:

• Synthesis and implementation are run and both are successful.
• A bitstream is generated and the bitstream generation is also successful.Fig 5.1 depicts the message window which indicates the successful completion of bitstream generation.

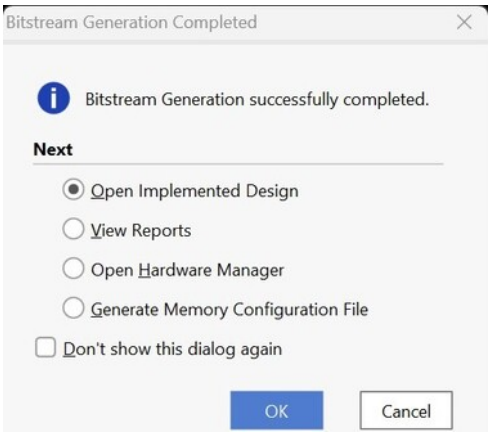

Figure 5.1: Bitstream generation

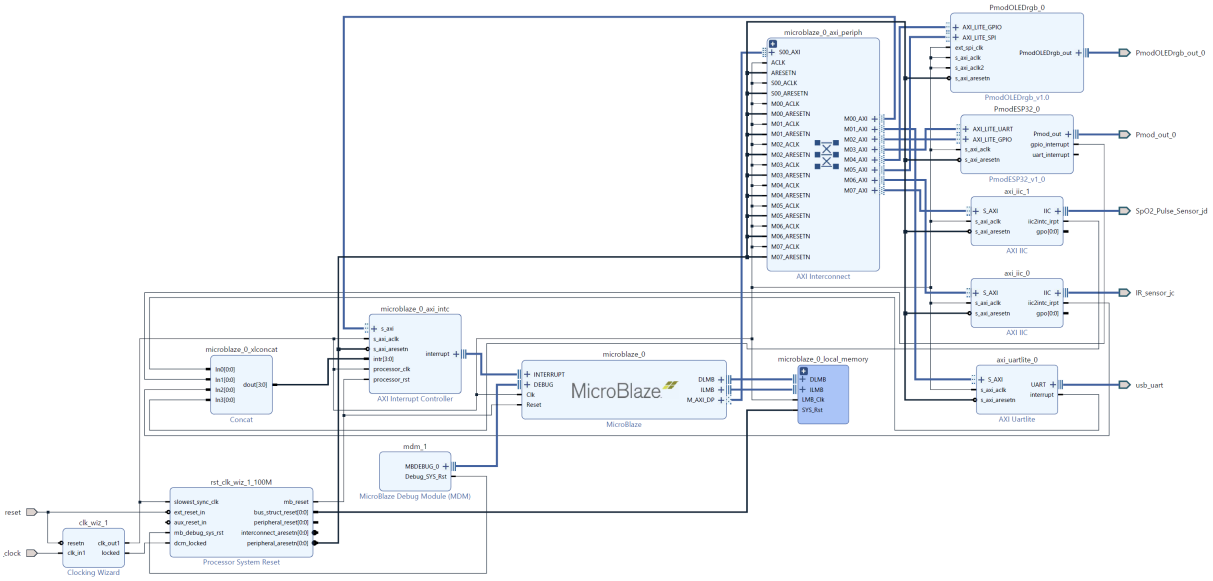

Figure 5.2: IP block design

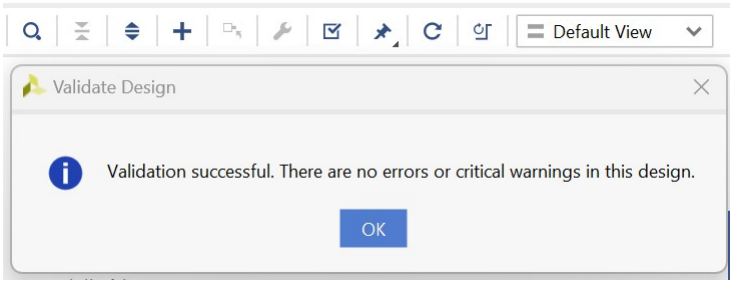

Figure 5.3: design validation

5.1.2 Unveiling Challenges in Hardware Design Steps

• There were limited learning video resources available. It was necessary to read IP documentation and understand it.

• The project encountered challenges in determining suitable interfaces for connecting the sensor with the microblaze processor. To overcome this obstacle, multiple interfaces, namely the Pmod IP bridge IP and the AXI IIC IP, were utilized. However, attempts to automate the connection process for the Pmod IP Bridge IP were unsuccessful, necessitating manual intervention, which was not feasible within the project's constraints. In contrast, exploring the AXI IIC IP option yielded more positive outcomes, although uncertainties persisted due to unresolved pin assignment issues that led to repeated failures during the Bitstream generation stage. Additionally, configuring the external parameters of the AXI IIC IP and determining the appropriate settings remained unclear. Nonetheless, by investing considerable time and thoroughly studying the sensor documentation, the correct IP configuration was eventually achieved.

• Initially, the generation of the constraint file posed considerable confusion, leaving me uncertain about the required port names and whether it was even necessary for the project. Nevertheless, through persistent experimentation and exploration, the constraint file was successfully configured, ensuring accurate port definitions.

```
##Pmod Header JC

#set_property -dict { PACKAGE_PIN K1    IOSTANDARD LVCMOS33 } [get_ports { JC[1] }]; #IO_L23N_T3_35 Sch=jc[1]
#set_property -dict { PACKAGE_PIN F6    IOSTANDARD LVCMOS33 } [get_ports { JC[2] }]; #IO_L19N_T3_VREF_35 Sch=jc[2]
set_property -dict { PACKAGE_PIN J2    IOSTANDARD LVCMOS33 } [get_ports { IR_sensor_jc_sda_io }]; #IO_L22N_T3_35 Sch=jc[3]
set_property -dict { PACKAGE_PIN G6    IOSTANDARD LVCMOS33 } [get_ports { IR_sensor_jc_scl_io }]; #IO_L19P_T3_35 Sch=jc[4]
#set_property -dict { PACKAGE_PIN E7    IOSTANDARD LVCMOS33 } [get_ports { JC[7] }]; #IO_L6P_T0_35 Sch=jc[7]
#set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports { JC[8] }]; #IO_L22P_T3_35 Sch=jc[8]
#set_property -dict { PACKAGE_PIN J4    IOSTANDARD LVCMOS33 } [get_ports { JC[9] }]; #IO_L21P_T3_DQS_35 Sch=jc[9]
#set_property -dict { PACKAGE_PIN E6    IOSTANDARD LVCMOS33 } [get_ports { JC[10] }]; #IO_L5P_T0_AD13P_35 Sch=jc[10]


##Pmod Header JD

#set_property -dict { PACKAGE_PIN H4    IOSTANDARD LVCMOS33 } [get_ports { JD[1] }]; #IO_L21N_T3_DQS_35 Sch=jd[1]
#set_property -dict { PACKAGE_PIN H1    IOSTANDARD LVCMOS33 } [get_ports { JD[2] }]; #IO_L17P_T2_35 Sch=jd[2]
set_property -dict { PACKAGE_PIN G1    IOSTANDARD LVCMOS33 } [get_ports { SpO2_Pulse_Sensor_jd_scl_io }]; #IO_L17N_T2_35 Sch=jd[3]
set_property -dict { PACKAGE_PIN G3    IOSTANDARD LVCMOS33 } [get_ports { SpO2_Pulse_Sensor_jd_sda_io }]; #IO_L20N_T3_35 Sch=jd[4]
#set_property -dict { PACKAGE_PIN H2    IOSTANDARD LVCMOS33 } [get_ports { JD[7] }]; #IO_L15P_T2_DQS_35 Sch=jd[7]
#set_property -dict { PACKAGE_PIN G4    IOSTANDARD LVCMOS33 } [get_ports { JD[8] }]; #IO_L20P_T3_35 Sch=jd[8]
#set_property -dict { PACKAGE_PIN G2    IOSTANDARD LVCMOS33 } [get_ports { JD[9] }]; #IO_L15N_T2_DQS_35 Sch=jd[9]
#set_property -dict { PACKAGE_PIN F3    IOSTANDARD LVCMOS33 } [get_ports { JD[10] }]; #IO_L13N_T2_MRCC_35 Sch=jd[10]
```

Figure 5.4: Constraint File

5.1.3 Hardware Design Result

To enhance understanding of the hardware design, a dashboard image can be accompanied by a description of the information it presents. This includes resource utilization, which displays the FPGA resources used such as logic elements, flip-flops, and block RAMs. Timing analysis provides details on timing constraints like maximum frequency and setup and hold times. Power analysis shows the estimated power consumption, including a breakdown by IP. Implementation summary offers a design status overview, including errors and warnings, design utilization, and status. This accompanying information helps assess system performance and constraints. Fig 5.5 shows the hardware design result in a dashboard.

Furthermore, the project summary depicts the resource utilization, implementation

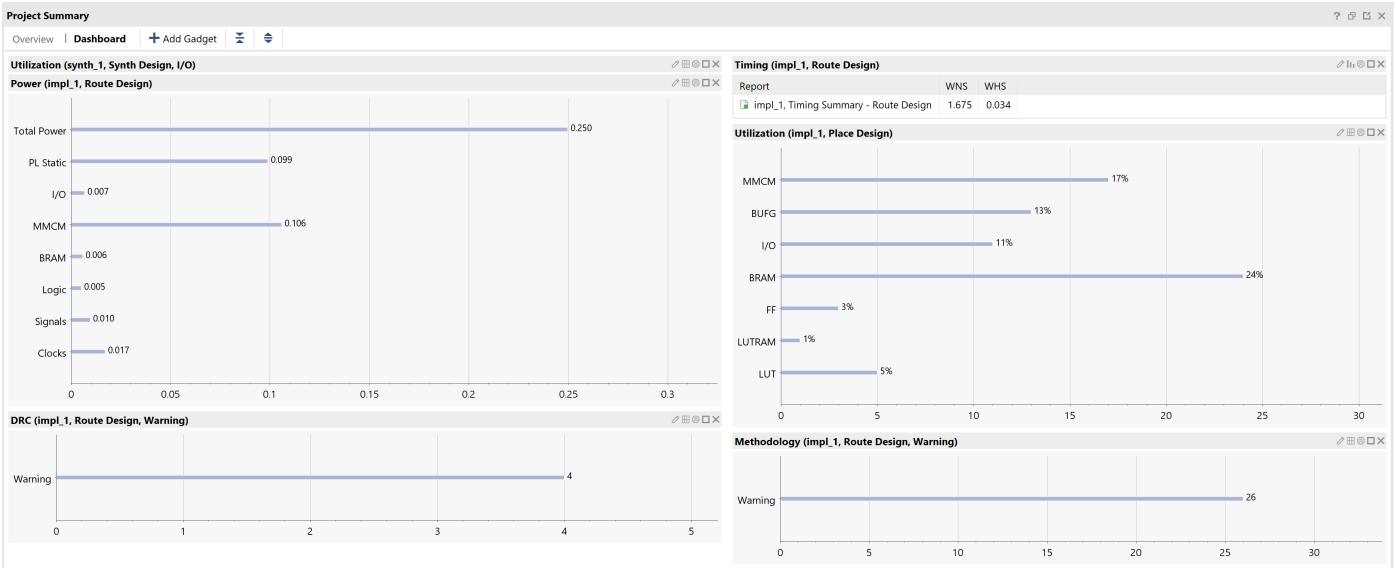

Figure 5.5: Dashboard

results, power consumption, and timing behavior in Figures 5.6, 5.7,5.8 and 5.9.

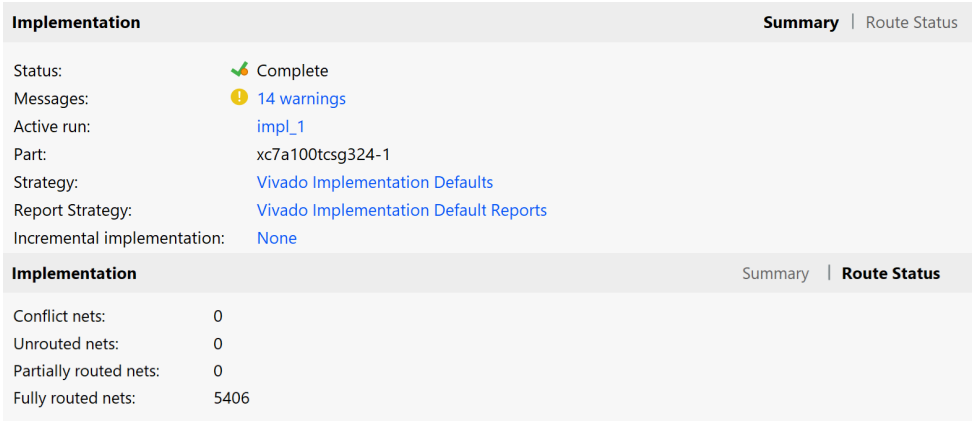

Figure 5.6: Implementation Summery

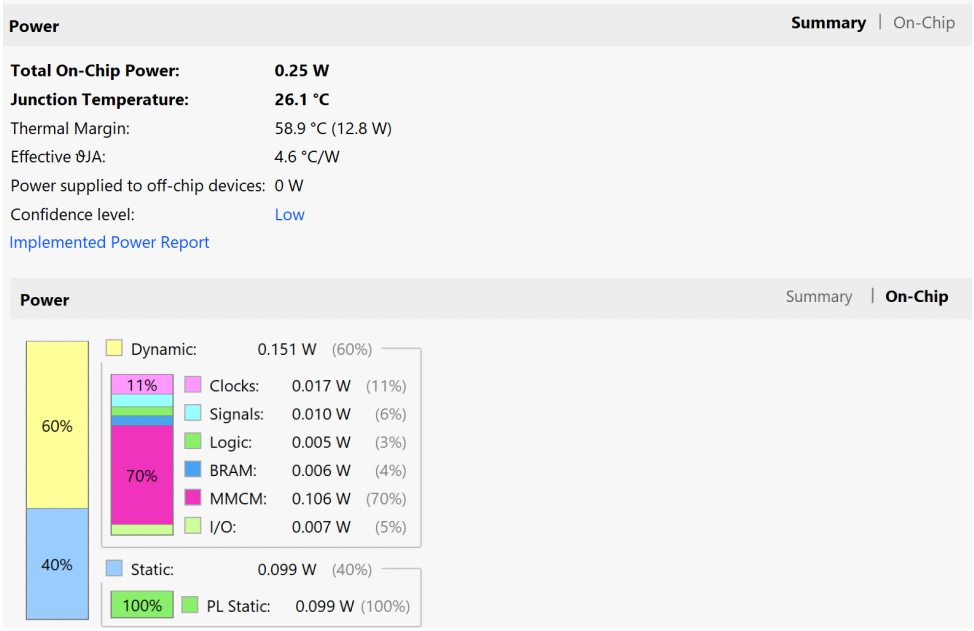

Figure 5.7: Power Consumption

5.2 Cloud Server Setup

The cloud server was set up using the Thingspeak server, an IoT server offered by Mathworks. Registration was done on their website, Thingspeak Website Link, and a new channel titled "Multiparameter Health Monitoring" was created, with three fields: Heart Rate (Pulse), Oxygen Saturation Level (SpO2), and On Ear Temperature. Three widgets were added to obtain precise numerical values for these parameters. To write data to the channel fields, an API request with the write API key was used, which is shown in Fig 5.10 . Fig 5.10 also shows the graphical and numerical interpretation of all measured health parameter data.

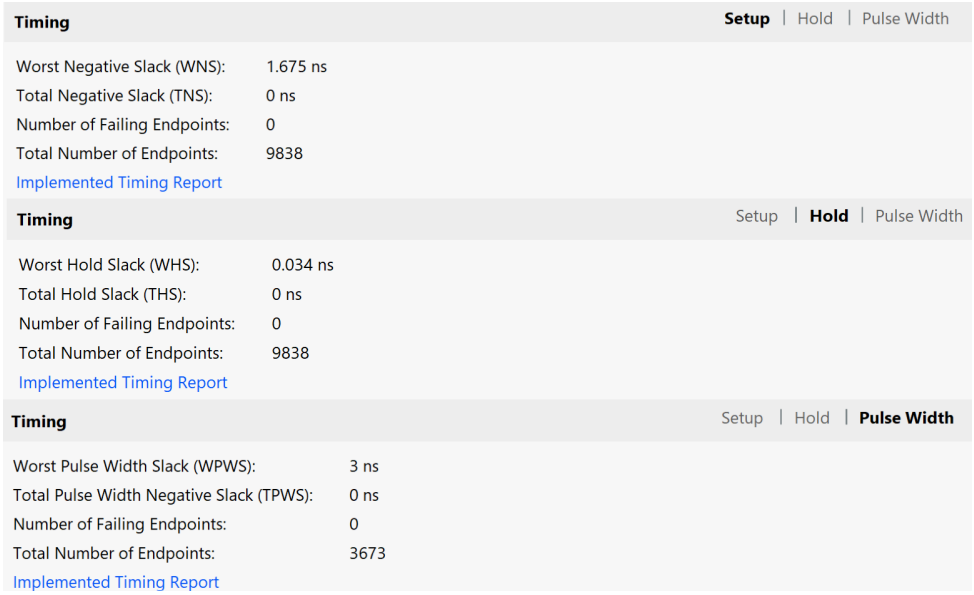

Figure 5.8: Timing Behaviour

5.3 Software Design

5.3.1 Software Architecture

The software architecture of the system is organized into the following components, each serving a specific purpose:

1. Initialization
    - Initializes peripherals and configurations
2. Sensor Configuration
    - Configures the MAX30102 and MLX90614 sensor for measurements
    - measure Infrared body temperature
    - Sets SpO2 and Heart rate mode for MAX30102 sensor
    - Configures other SpO2 and heart rate related settings
    - Sets FIFO configuration
    - measure Oxygen saturation (SpO2) and heartrate (Pulse)
3. WiFi Setup
    - Sends AT commands to reset ESP32 module
    - Sets WiFi mode to Station
    - Connects to the specified WiFi network
4. WiFi Transmission
    - Sets single connection mode

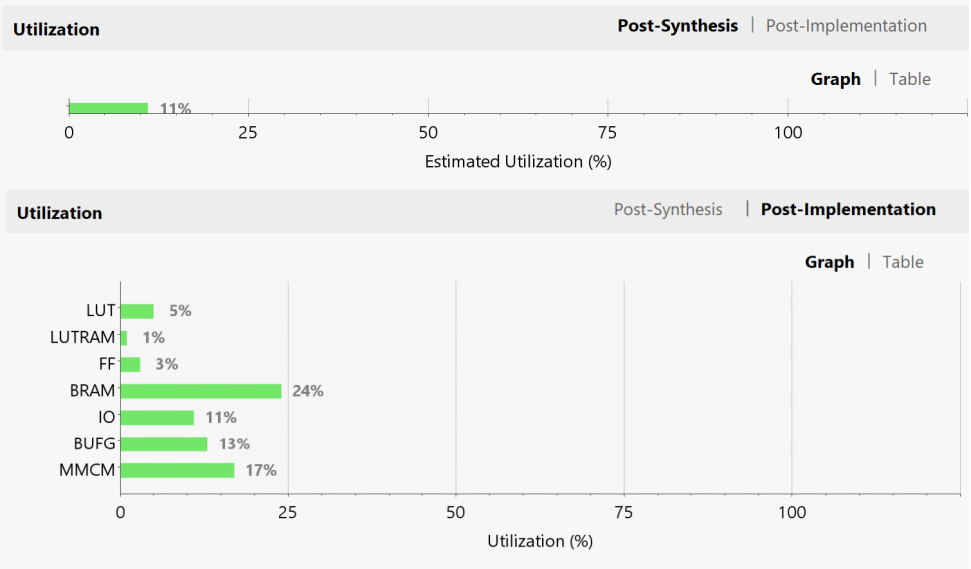

Figure 5.9: Resource Utilization

- Establishes TCP connection with Thingspeak server
- Formats data string with temperature, SpO2, and pulse values
- Transmits data to the Thingspeak server over WiFi
- Closes TCP connection

5. Data Display
   - Displays temperature, SpO2, and pulse information on Pmod OLEDrgb display
   - Updates relevant data on the display
6. FIFO Data Reading
   - Retrieves FIFO write and read pointers
   - Calculates the number of available samples
   - Reads data from MAX30102 FIFO buffer
7. Reading Temperature
   - This component is responsible for reading the temperature from the MLX90614 sensor. It communicates with the sensor and retrieves the temperature data and return temperature in degree celcious.

5.3.2 Dependencies and Requirements

The software has dependencies on the following libraries and communication protocols:

- Xilinx IIC Library (I2C): This library is used for communication with the MAX30102 sensor and MLX90614 sensor. It provides functions for sending and receiving data over the I2C bus.

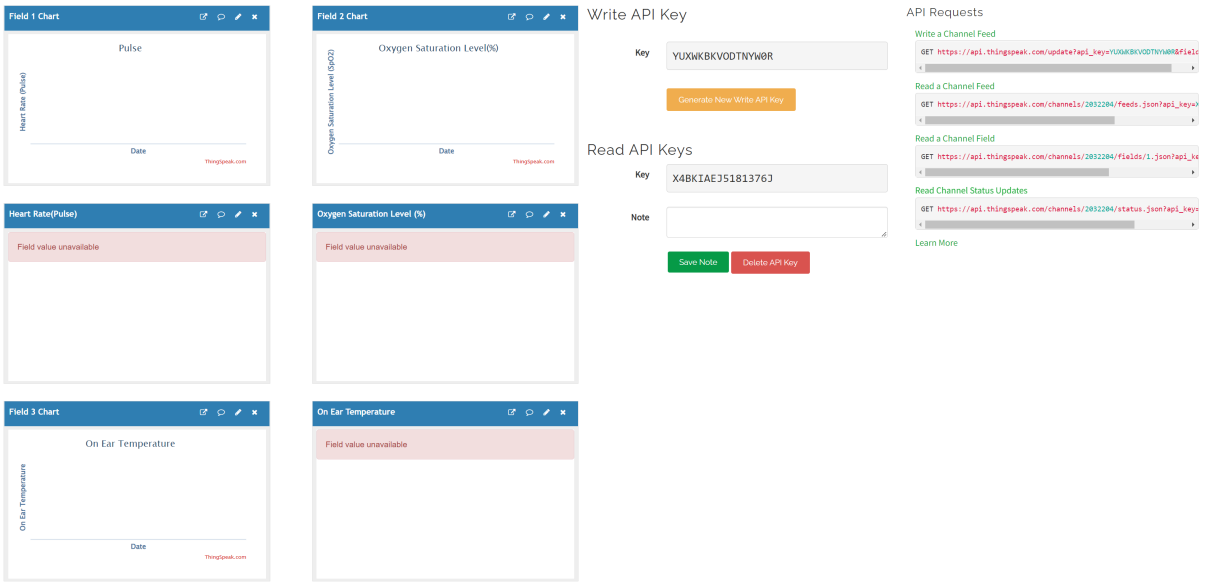

Figure 5.10: Thingspeak Cloud Server

- Pmod ESP32 Library (UART): This library enables WiFi communication. It includes functions for establishing WiFi connections, sending AT commands to the ESP32 module (using UART communication), and receiving responses.
- Pmod OLEDrgb Library (SPI): This library is responsible for controlling the OLED display. It provides functions for clearing the display, setting the font color, positioning the cursor, and printing text. It communicates with the OLED display using the SPI protocol.
- Algorithm Library (algorithm.h): This library is a dependency for the software and is used to calculate oxygen saturation (SpO2) and heart rate (Pulse) from the raw data obtained from the MAX30102 sensor. It includes functions that process the raw data and provide measurements of SpO2 and pulse values.

These libraries serve as the foundation for the software, providing the necessary functionality to interact with external devices and perform communication tasks.

5.3.3 Key Functions and Algorithms

The software implementation includes several key functions and algorithms that are essential for the system's functionality. These functions and algorithms perform crucial tasks and calculations, enabling the accurate measurement of vital parameters. Some of the key functions and algorithms are:

- `MLX90614_initialize()`:This function initializes the MLX90614 sensor.
- `MAX30102_Init()`: This function initializes MAX30102 sensor.
- `Sensor_Config()`: This function is responsible for configuring the MAX30102 sensor for measurements. It sets the LED mode, SpO2-related settings, and FIFO configuration.

- `ReadFifoData()`: This function retrieves 100 raw data samples from the FIFO buffer and performs calculations for pulse and oxygen saturation. Additionally, it verifies finger detection and ensures the acquisition of an adequate number of valid samples. It is set to the capacity of processing a maximum of 300 samples, including both valid and invalid readings. If fewer than 100 valid samples are collected within this 300, the function exits without calculating SpO2 and pulse. However, if the required number of valid samples is obtained, it proceeds with the computation of SpO2 and pulse.
- `UpdateData()`: The `UpdateData()` function is responsible for handling insufficient or failed initial data readings. It follows a specific process to ensure the availability of a satisfactory number of data points for accurate calculations. Initially, the function discards the first 25 samples from a total of 100 samples by shifting the subsequent 75 samples to occupy the first 75 positions in the array. Consequently, the last 25 positions in the array remain empty.To supplement the data, the function retrieves an additional 25 samples from the FIFO buffer and incorporates them into the array. This retrieval process is implemented through a loop that acquires the necessary number of samples. Furthermore, the function incorporates error handling mechanisms to ensure its continued operation even in the absence of a finger on the sensor. It detects the absence of a finger and only adds samples that are deemed valid. However, it does not wait indefinitely for a valid reading. Instead, it can acquire a maximum of 75 samples, including both valid and invalid readings, before exiting the function.When a finger is detected, the function takes the reading and updates the FIFO sample array. It repeats this process up to a maximum of 5 times, until valid SpO2 and heart rate readings are obtained. Once these readings are acquired, the function exits.However, in the final implementation, this function is not included in the main code and remains unused. This decision is made to conserve the memory of the system.
- `ReadFifoPointers()`: This function retrieves the FIFO write pointer from the MAX30102 sensor. It communicates with the sensor using the I2C protocol and obtains the current positions of the write pointer in the FIFO buffer and return the number of available samples to read
- `maxim_heart_rate_and_oxygen_saturation()`: This algorithm, implemented in the `algorithm.h` library, calculates the heart rate and oxygen saturation (SpO2) values based on the raw data obtained from the MAX30102 sensor. It utilizes signal processing techniques and algorithms to extract relevant information from the sensor data and provide accurate measurements.
  `Steps of the Algorithm are described below:`
  - `Data Preprocessing`
    - The algorithm first preprocesses the raw data by subtracting the mean value from each sample, normalizing the data for further analysis.
  - `Finding IR Valley Locations`
    - The algorithm searches for valleys in the preprocessed IR data. These valleys correspond to points where the intensity of infrared light passing through the finger is at its lowest.

    - The algorithm identifies these valley locations using a peak detection algorithm, considering a minimum threshold value.

  - `Heart Rate Calculation`
    - If the algorithm finds at least two IR valley locations, it proceeds to calculate the heart rate.
    - It calculates the average distance between consecutive valleys and uses it to estimate the heart rate in beats per minute (BPM).
  - `Calculating Oxygen Saturation (SpO2)`
    - The algorithm extracts oxygen saturation (SpO2) information based on the ratio between the AC (alternating current) and DC (direct current) components of the IR and red signals.
    - It analyzes the peak-to-peak amplitudes of the AC and DC components to derive the SpO2 value.
    - The algorithm computes the ratio between the AC and DC components and uses a lookup table to determine the corresponding SpO2 value.
  - `Validating Results`
    - The algorithm performs checks to validate the calculated heart rate and SpO2 values.
    - If the results pass the validation criteria, they are considered accurate and valid.
    - If the results do not meet the validation criteria, they are marked as invalid, and alternative measures may be required.

- `wifi_Setup() & WiFi_Transmission()`: This two function establishes a WiFi connection and transmits the collected data to the Thingspeak server. It sets the ESP32 module to the appropriate mode, connects to the specified WiFi network, and establishes a TCP connection with the server. The AT commands used in this function are as follows:
  - `AT+RST`: This command sends a reset signal to the ESP32 module, ensuring a clean start.
  - `AT+CWMODE=1`: This command sets the WiFi mode of the ESP32 module to Station mode, allowing it to connect to a WiFi network.
  - `AT+CWJAP="SSID","password"`: This command connects the ESP32 module to the specified WiFi network using the provided SSID (network name) and password.
  - `AT+CIPMUX=0`: This command sets the ESP32 module to single connection mode, allowing it to establish a single TCP connection with the Thingspeak server.

  - `AT+CIPSTART="TCP","api.thingspeak.com",80`: This command establishes a TCP connection with the Thingspeak server using the specified IP address and port number.
  - `AT+CIPSEND=<data_length>`: This command sends the formatted data string to the server, where <data_length> is the length of the data string.
  - `AT+CIPCLOSE`: This command closes the TCP connection with the server after the data transmission is complete.

- `readTemperature()` The `readTemperature()` function retrieves the temperature reading from the MLX90614 sensor.
- `Data_Display()`: This function displays the measured temperature, SpO2, and pulse information on the OLED display. It utilizes the OLEDrgb library to clear the display, set the font color, position the cursor, and print the relevant data in a user-friendly format.

These key functions and algorithms work together to enable the measurement and transmission of vital parameters, providing valuable insights for monitoring health and wellness.

5.3.4 Error Handling and Exception Cases

The software implementation includes robust error handling mechanisms to handle communication errors and finger detection issues. It incorporates the following measures to ensure reliable and accurate operation:

- Communication Error Handling: The software checks for errors during data transmission and reception. If any communication errors occur, appropriate feedback or error messages are provided to indicate the failure and assist in troubleshooting.
- Finger Detection Failure: In cases where finger detection fails or no finger is detected, the software handles these exceptions gracefully. It shows the Error message on display , and doesn't transmit the faulty data to Thingspeak channel . Also , there is a timeout , so that if no finger is detected , after a specified time passed , it continues to take measurement of other sensor and so on .
- Sufficient FIFO Data: To ensure accurate results, the software verifies that a sufficient number of samples are read from the FIFO buffer. This helps mitigate the impact of incomplete or unreliable data, enhancing the overall accuracy of measurements.
- Timing Delays: The software incorporates appropriate timing delays to allow for proper sensor stabilization and reliable data acquisition. These delays ensure that the sensor and other components have sufficient time for initialization and operation, reducing the likelihood of erroneous readings.

5.4 Testing and Validation

To ensure the reliability and accuracy of the software implementation, comprehensive testing and validation were performed using the following approaches:

- **Unit Testing**: Each individual function and algorithm was subjected to unit testing. This involved testing each function in isolation, providing known input values, and comparing the output against expected results. This approach allowed for the verification of the correctness and functionality of individual software components.
- **Integration Testing**: Integration testing was conducted to assess the seamless communication and compatibility between different software components. By simulating real-world scenarios and interactions between modules, the integration testing phase verified the overall system behavior. This testing approach ensured that the software components properly interacted and produced the desired outcomes when combined.
- **Functional Testing**: Extensive functional testing was performed to evaluate the software's overall functionality and performance. This testing phase involved validating the sensor configuration process, data reading from the FIFO buffer, WiFi transmission of data, and the proper functioning of the display module. Through rigorous functional testing, the software's adherence to the specified requirements and its ability to perform the intended tasks were confirmed.
- **Boundary Testing**: The software was subjected to boundary testing to assess its behavior in extreme scenarios. This involved testing with maximum and minimum values, empty FIFO buffers, and unexpected sensor readings. By evaluating the software's response in these boundary conditions, its robustness and ability to handle exceptional cases were determined.

Through a combination of unit testing, integration testing, functional testing, and boundary testing, the software implementation was thoroughly examined and validated. This systematic approach ensured that the software met the desired standards of reliability, accuracy, and functionality. Any identified issues or discrepancies were addressed and resolved to ensure a robust and dependable software solution.

5.4.1 Final Outcome

The final outcome of the project is as follows:

1. The system exhibits intermittent functionality, sometimes working properly and sometimes failing to execute.
2. The system should operate continuously.However, after several iterations of sensor data readings, calculations, and data transfer to Thingspeak, it stops executing. This issue is particularly evident when errors occur during the calculation process, as the loop seems to cease functioning inexplicably.
3. The Wi-Fi connection frequently disconnects after a certain period, requiring reconnection each time it is lost.
4. The temperature measurement functionality is accurate; however, the measurements of oxygen saturation and pulse show inconsistencies without any discernible cause.

One plausible explanation for these issues could be attributed to memory overlap, as evidenced by recurrent errors indicating excessive memory consumption. A potential

resolution to address this problem involves augmenting the available memory capacity.

A visual representation of the Thingspeak server and the Pmod display is provided in Figure 5.11.

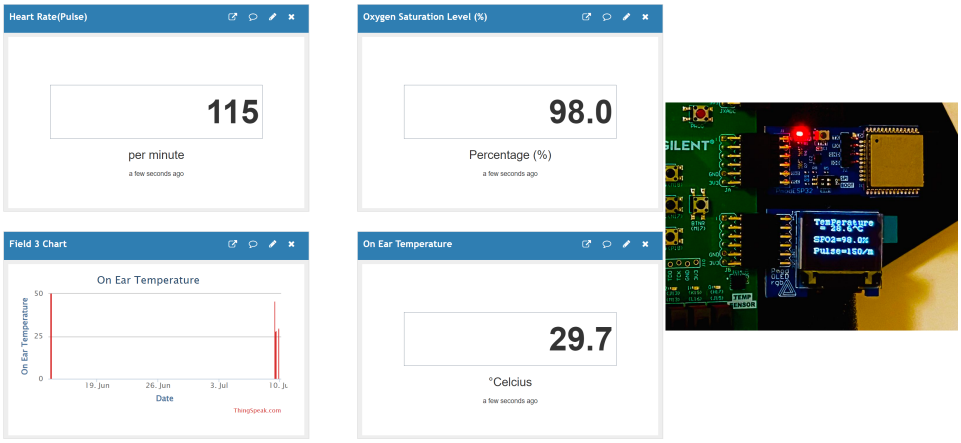

Figure 5.11: Final Outcome Visualization

5.5 Evaluation

The evaluation of the project aimed to assess the performance and functionality of the implemented system. Several aspects were considered, including meeting requirements, accuracy of temperature measurement, calculation of oxygen saturation and pulse, and successful data transmission to the Thingspeak server etc.
The project successfully fulfilled almost all the specified requirements, indicating that the system meets the intended objectives effectively.

Infrared Body Temperature Measurement
Requirement 1.1: Successful Temperature Measurement
The infrared body temperature measurement component successfully provided accurate temperature readings during the evaluation. The sensor consistently measured the expected temperature of in degrees Celsius, demonstrating its reliability and accuracy. Therefore, there were no inaccuracies observed in the temperature measurement process.

Oxygen Saturation and Pulse Calculation
Requirement 1.2 and 1.3:: Accurate Red LED and Infrared Data Acquisition, but inconsistent.
infrared (IR) and red LED (redled) samples from pulse oximetry sensor were in an acceptable format for calculating oxygen saturation and pulse. However, when applying existing online algorithms designed for Arduino environments, the calculated results were found to be incorrect. Therefore, further modifications to the algorithm or the development of a new algorithm specifically tailored to the system's architecture are necessary to ensure accurate calculation of oxygen saturation and pulse values. Also, the reason for this problem can the presence of signal noise. It is recommended to use the sensor in a dark environment.

Sensor can be attached to a device that blocks out surrounding light when it's placed on finger. To improve accuracy,the photo receptor can be positioned on the opposite side of the finger from where the infrared (IR) and red LED light are emitted. This setup can help reduce inaccuracies and minimize signal noise.

Display Data
Requirement 1.4: Successful Execution
The data is successfully displayed on a Pmod OLED without any errors.

Data Transmission and Visualization
Requirement 1.5: Effective Implementation
The project successfully accomplished the transmission of data to the Thingspeak server. The transmitted data was received and displayed in a readable format on the Thingspeak server interface. This successful data transmission demonstrates the effectiveness of the implemented communication protocols and validates the integrity of the collected data.

While the project has successfully met most of the requirements, certain areas require further attention and improvement. Addressing the issues related to refining the oxygen saturation and pulse calculation algorithm, and implementing signal processing techniques are necessary for enhancing the accuracy and reliability of the system.

6 Outlook and Summary

The project documentation presented the development and evaluation of a real-time health monitoring system using cutting-edge technology and the Nexys A7-100T FPGA development board. The system incorporated sensors for pulse, oxygen saturation, and body temperature, along with a microblaze processor for data processing.
The project successfully met the majority of the specified requirements, demonstrating the effectiveness of the implemented solutions. The system exhibited succesful sensor readiings and reliable data transmission to the Thingspeak server, allowing for remote monitoring of patient's health conditions.
While the system fulfilled the intended objectives, some challenges were encountered during the evaluation phase.The algorithm for calculating oxygen saturation and pulse showed inconsistencies when applied in the project environment. Further modifications or the development of a new algorithm specific to the system's architecture are recommended to ensure accurate and reliable measurements.Also, it is recommended to increase the total memory to achieve a consistent system.
The project has showcased the potential of utilizing the Microblaze processor in developing IoT-based patient monitoring systems. By leveraging its capabilities, the system achieved better resource utilization, processing speed, and potential for future upgrades.
Overall, the project has contributed to the advancement of real-time health monitoring systems and highlights the need for ongoing research and improvement in this domain. The findings underscore the importance of addressing challenges related to component reliability, algorithm accuracy, and signal noise mitigation to ensure the system's effectiveness in providing accurate and timely health information for better patient outcomes.

Bibliography

[ADVBS20] A. Athira, T.D. Devika, K.R. Varsha, and Sree Sanjanaa Bose S. Design and development of iot based multi-parameter patient monitoring system. In *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 862–866, 2020.

[IP] https://www.xilinx.com/products/intellectual-property.html.

[KAAA22] Mohammad Monirujjaman Khan, Turki M. Alanazi, Amani Abdulrahman Albraikan, and Faris A. Almalki. Iot-based health monitoring system development and analysis. *Security and Communication Networks*, 2022:9639195, Apr 2022.

[KR16] R. Kumar and M. Pallikonda Rajasekaran. An iot based patient monitoring system using raspberry pi. In *2016 International Conference on Computing Technologies and Intelligent Data Engineering (ICCTIDE'16)*, pages 1–4, 2016.

[LCK18] Dmitry Levshun, Andrey Chechulin, and Igor Kotenko. A technique for design of secure data transfer environment: Application for i2c protocol. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pages 789–794, 2018.

[Lee09] Frederic Leens. An introduction to i2c and spi protocols. *IEEE Instrumentation Measurement Magazine*, 12(1):8–13, 2009.

[LLY07] Ka-cheong Leung, Victor O.k. Li, and Daiqin Yang. An overview of packet reordering in transmission control protocol (tcp): Problems, solutions, and challenges. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):522–535, 2007.

[Mic] https://www.xilinx.com/products/design-tools/microblaze.html.

[Nex] https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual.

[NP16] Umakanta Nanda and Sushant Kumar Pattnaik. Universal asynchronous receiver and transmitter (uart). In *2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS)*, volume 01, pages 1–5, 2016.

[OLE] https://digilent.com/reference/$_m$*edia/pmod* : *pmod* : *pmodoledrgb*$_r$*m.pdf*.

[oxy] https://www.analog.com/media/en/technical-documentation/data-sheets/max30102.pdf.

[Pmo] https://digilent.com/reference/pmod/pmodesp32/start.

[SMDD20] Sambit Satpathy, Prakash Mohan, Sanchali Das, and Swapan Debbarma. A new healthcare diagnosis system using an iot-based fuzzy classifier with fpga. *The Journal of Supercomputing*, 76(8):5849–5861, Aug 2020.

[tem] https://www.melexis.com/en/documents/documentation/datasheets/datasheet-mlx90614.

[UAB17] Mohammad Salah Uddin, Jannat Binta Alam, and Suraiya Banu. Real time patient monitoring system based on internet of things. In *2017 4th International Conference on Advances in Electrical Engineering (ICAEE)*, pages 516–521, 2017.

[Viva] https://ebics.net/xilinx-vivado/.

[Vivb] https://www.xilinx.com/products/design-tools/vivado.html.

List of Figures

List of Tables

Affidavit

I Saikot Das Joy herewith declare that I have composed the present paper and work by myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance.

Saikot Das Joy

Lippstadt, 15th July 2023

A Appendix

A.0.1 Complete Xilinx C code

```
#include "xparameters.h"
#include "xil_printf.h"
#include "sleep.h"
#include "stdio.h"
#include "PmodOLEDrgb.h"
#include "PmodESP32.h"
#include "xiic.h"
#include "xparameters.h"
#include "xstatus.h"
#include <xiic_i.h>
#include "xintc.h"
#include "xil_exception.h"
#include <stdbool.h>
#include <stdlib.h>
#include "algorithm.h"
#include <stdint.h>


#define MAX30102_IIC_DEVICE_ID XPAR_IIC_1_DEVICE_ID // for
   MAX30102 sensor
#define MLX90614_IIC_DEVICE_ID XPAR_IIC_0_DEVICE_ID // for
   MLX90614 sensor

// Register addresses of MLX90614
#define GY906_ADDRESS 0x5A   // I2C device address of the sensor
   MLX90614
#define TEMP_REG_ADDRESS 0x07 // obj temperature register of
   MLX90614 sensor

// Register adresses of MAX30102
#define MAX30102_I2C_ADDRESS 0x57 // 7-bit slave address of
   MAX30102 sensor
#define MAX30102_INT_ENABLE_1 0x02
#define MAX30102_FIFO_DATA 0x07
#define REG_FIFO_CONFIG 0x08
#define MAX30102_REG_MODE_CONFIG   0x09
#define MAX30102_REG_LED1_PA       0x0C
#define MAX30102_REG_LED2_PA       0x0D
#define MAX30102_REG_FIFO_WR_PTR   0x04
```

```
#define MAX30102_REG_FIFO_RD_PTR   0x06
#define MAX30102_REG_FIFO_OVERFLOW 0x05
#define MAX30102_REG_SPO2_CONFIG   0x0A

#ifdef __MICROBLAZE__
#define HOST_UART_DEVICE_ID XPAR_AXI_UARTLITE_0_BASEADDR
#define HostUart XUartLite
#define HostUart_Config XUartLite_Config
#define HostUart_CfgInitialize XUartLite_CfgInitialize
#define HostUart_LookupConfig XUartLite_LookupConfig
#define HostUart_Recv XUartLite_Recv
#define HostUartConfig_GetBaseAddr(CfgPtr) (CfgPtr->RegBaseAddr)
#include "xuartlite.h"
#include "xil_cache.h"
#else
#define HOST_UART_DEVICE_ID XPAR_PS7_UART_1_DEVICE_ID
#define HostUart XUartPs
#define HostUart_Config XUartPs_Config
#define HostUart_CfgInitialize XUartPs_CfgInitialize
#define HostUart_LookupConfig XUartPs_LookupConfig
#define HostUart_Recv XUartPs_Recv
#define HostUartConfig_GetBaseAddr(CfgPtr) (CfgPtr->BaseAddress)
#include "xuartps.h"
#endif

HostUart myHostUart;

#define PMODESP32_UART_BASEADDR
   XPAR_PMODESP32_0_AXI_LITE_UART_BASEADDR
#define PMODESP32_GPIO_BASEADDR
   XPAR_PMODESP32_0_AXI_LITE_GPIO_BASEADDR

void EnableCaches();
void DisableCaches();
void Initialize();
void Sensor_Config();
void MAX30102_WriteReg(unsigned char reg, unsigned char value);
int MAX30102_Init();
void MLX90614_initialize();
void read_temp();
void ReadFifoData();
void UpdateData();
int ReadFifoPointers();
void displayDATA();
void wifi_Setup();
void wifi_transmission();
void DemoCleanup();


PmodOLEDrgb oledrgb;
```

```
84 PmodESP32 myESP32;

86 XIic Iic; // instance for MLX9061
   XIic IicInstance; // instance for MAX30102
88
   int Pulse = 0;
90 int SpO2 = 0;
   float temp = 0.00;
92
   int32_t spo2;
94 int8_t spo2Valid; // value 1 if valid , value 0 if not valid
   int32_t heartRate;
96 int8_t heartRateValid;  // value 1 if valid , value 0 if not
      valid
   int sample = 0;
98 u32 redData,irData; // redled data, and infrared data read from
      MAX30102 sensor
   u32 redDataArr[100]; // raw redled data
100 u32 irDataArr[100]; // raw infrared data

102 int main() {
      Initialize(); // initialize everything except  MAX30102 and
      MLX90614  sensor
104    MLX90614_initialize();
      MAX30102_Init(); // initialize MAX30102 sensor
106    usleep(50000);
      displayDATA(); // display setting , initially all data set to
      NULL
108    usleep(50000);
      // Filling redDataArr with 0's
110    for (int i = 0; i < 100; i++) {
          redDataArr[i] = 0;
112    }

114    // Filling irDataArr with 0's
      for (int i = 0; i < 100; i++) {
116        irDataArr[i] = 0;
      }
118
      wifi_Setup(); // setup wifi connection
120

122   while(1){

124      // read the  temperature
       read_temp();
126
       // update temperature data on Oled display
128    if(temp!= 0.00){
           // update temperature data on display
```

```
130          OLEDrgb_SetCursor(&oledrgb, 1, 0);
             OLEDrgb_PutString(&oledrgb, "Temperature");
132          OLEDrgb_SetCursor(&oledrgb, 2, 1);
             char tempStr[20];
134          snprintf(tempStr, 20, "= %.1f", temp);
             OLEDrgb_PutString(&oledrgb, tempStr);
136          OLEDrgb_PutString(&oledrgb, "^C");
      }
138
      ReadFifoData();
140
     // update SpO2 and heart rate data on display
142  if (spo2Valid == 1 && heartRateValid == 1)
     {
144      char spo2str[4];
         char pulsestr[4];
146      SpO2 = (int)spo2;
         Pulse = (int)heartRate;
148
         OLEDrgb_SetCursor(&oledrgb, 1, 3);
150      OLEDrgb_PutString(&oledrgb, "SpO2=");
         snprintf(spo2str, 4, "%d", SpO2);
152      OLEDrgb_PutString(&oledrgb, spo2str);
         OLEDrgb_PutString(&oledrgb, ".0%");
154
         OLEDrgb_SetCursor(&oledrgb, 1, 5);
156      OLEDrgb_PutString(&oledrgb, "Pulse=");
         snprintf(pulsestr, 4, "%d", Pulse);
158      OLEDrgb_PutString(&oledrgb, pulsestr);
         OLEDrgb_PutString(&oledrgb, "/m");
160
         u8 check_resp[128];
162      unsigned char wifi_check[] = "AT+CWJAP?\r\n";
         ESP32_SendBuffer(&myESP32, wifi_check, strlen((char*)
    wifi_check));
164      xil_printf("Sent command: %s\r\n", wifi_check);
         usleep(1000000);
166      // Receive response
         ESP32_Recv(&myESP32, check_resp, sizeof(check_resp));
168      xil_printf("Received response:%s\r\n", check_resp);
         usleep(1000000);
170      // Check the response for Wi-Fi connection status
         if (strstr((char*)check_resp, "WIFI CONNECTED") != NULL)
    {
172        wifi_transmission();
         }
174      else {
           // Wi-Fi not connected or unexpected response
176        wifi_Setup();
           wifi_transmission();
```

```
178             }
        }
180     else
        {
182             OLEDrgb_SetCursor(&oledrgb, 1, 3);
                OLEDrgb_PutString(&oledrgb, "SpO2=");
184             OLEDrgb_PutString(&oledrgb, "Error ");

186             OLEDrgb_SetCursor(&oledrgb, 1, 5);
                OLEDrgb_PutString(&oledrgb, "Pulse=");
188             OLEDrgb_PutString(&oledrgb, "Error");
        }
190
        spo2Valid = 0;
192     heartRateValid = 0;
}
194    DemoCleanup();
        return 0;
196 }
// initialize everything except  MAX30102 and MLX90614  sensor
198 void Initialize () {
        HostUart_Config *CfgPtr2;
200     EnableCaches();
        ESP32_Initialize(&myESP32, PMODESP32_UART_BASEADDR,
    PMODESP32_GPIO_BASEADDR);
202     CfgPtr2 = HostUart_LookupConfig(HOST_UART_DEVICE_ID);
        HostUart_CfgInitialize(&myHostUart, CfgPtr2,
    HostUartConfig_GetBaseAddr(CfgPtr2));
204     OLEDrgb_begin(&oledrgb,
    XPAR_PMODOLEDRGB_0_AXI_LITE_GPIO_BASEADDR,
                XPAR_PMODOLEDRGB_0_AXI_LITE_SPI_BASEADDR);
206
}
208 void MLX90614_initialize(){
        XIic_Config *ConfigPtr1;    /* Pointer to configuration data
    */
210
        /* Initialize the IIC driver so that it is ready to use. */
212     ConfigPtr1 = XIic_LookupConfig(MLX90614_IIC_DEVICE_ID);

214     XIic_CfgInitialize(&Iic, ConfigPtr1, ConfigPtr1->BaseAddress)
    ;

216     XIic_Start(&Iic);
        // Set the slave address of MLX90614
218     XIic_SetAddress(&Iic, XII_ADDR_TO_SEND_TYPE, GY906_ADDRESS);
}
220 void read_temp() {
        for(int i = 0 ; i<10; i++) // read temperature 10 times , we
    take only the last reading
```

```
222     {
            // Send the register address of temperature data to
    MLX90614
224         u8 reg_address[1];
            reg_address[0] = TEMP_REG_ADDRESS;
226         XIic_Send(Iic.BaseAddress, GY906_ADDRESS, reg_address, 1,
     XIIC_REPEATED_START);
            // Read temperature data from MLX90614
228         u8 temp_data[3];
            XIic_Recv(Iic.BaseAddress, GY906_ADDRESS, temp_data, 3,
    XIIC_STOP);
230         // Convert temperature data to Celsius
            temp = (((temp_data[1])<<8|temp_data[0]) * 0.02) -
    273.15;
232     }
            }
234
// display All default data initially
236 void displayDATA() {
    OLEDrgb_Clear(&oledrgb);
238 OLEDrgb_SetFontColor(&oledrgb, OLEDrgb_BuildRGB(255, 255, 255)
    );
    OLEDrgb_SetCursor(&oledrgb, 1, 0);
240 OLEDrgb_PutString(&oledrgb, "Temperature");
    OLEDrgb_SetCursor(&oledrgb, 2, 1);
242 OLEDrgb_PutString(&oledrgb, "= Null");

244 OLEDrgb_SetCursor(&oledrgb, 1, 3);
    OLEDrgb_PutString(&oledrgb, "SpO2=");
246 OLEDrgb_PutString(&oledrgb, "Null");

248 OLEDrgb_SetCursor(&oledrgb, 1, 5);
    OLEDrgb_PutString(&oledrgb, "Pulse=");
250 OLEDrgb_PutString(&oledrgb, "Null");
}
252 //establish wifi connection
void wifi_Setup() {
254     u8 conn_resp[128];
    // AT command for resetting ESP32
256     unsigned char wifi_reset[] = "AT+RST\r\n";
    ESP32_SendBuffer(&myESP32, wifi_reset, strlen((char*)
    wifi_reset));
258     xil_printf("Sent command: %s\r\n", wifi_reset);
    usleep(100000);
260
    // AT command for setting Wi-Fi mode to Station
262     unsigned char mode_cmd[] = "AT+CWMODE?\r\n";
    ESP32_SendBuffer(&myESP32, mode_cmd, strlen((char*)mode_cmd))
    ;
264     xil_printf("Sent command: %s\r\n", mode_cmd);
```

```
    usleep(100000);

    // AT command for connecting to Wi-Fi
    unsigned char conn_cmd[] = "AT+CWJAP=\"saikot-das.joy@stud.hshl.de\",\"28452467MaBaba\"\r\n";
    ESP32_SendBuffer(&myESP32, conn_cmd, strlen((char*)conn_cmd));
    xil_printf("Sent command: %s\r\n", conn_cmd);
    usleep(1000000);
    // Receive response
    ESP32_Recv(&myESP32, conn_resp, sizeof(conn_resp));
    xil_printf("Received response:%s\r\n", conn_resp);
    usleep(5000000);

            u8 check_resp[64];
            // AT command for connecting to Wi-Fi
            unsigned char wifi_check[] = "AT+CWJAP?\r\n";
            ESP32_SendBuffer(&myESP32, wifi_check, strlen((char*)wifi_check));
            xil_printf("Sent command: %s\r\n", wifi_check);
            usleep(1000000);
            // Receive response
            ESP32_Recv(&myESP32, check_resp, sizeof(check_resp));
            xil_printf("Received response:%s\r\n", check_resp);
            usleep(1000000);

}
// transmit data to thingspeak
void wifi_transmission(){

    // AT commands to determine single connection mode
      unsigned char SMconMode[] = "AT+CIPMUX=0\r\n";
      ESP32_SendBuffer(&myESP32, SMconMode, strlen((char*)SMconMode));
      usleep(1000000);
      // AT command for TCP connection with Thingspeak server
      unsigned char TCP[] = "AT+CIPSTART=\"TCP\",\"api.thingspeak.com\",80\r\n";
      ESP32_SendBuffer(&myESP32, TCP, strlen((char*)TCP));
      usleep(1000000);

      // Format the string with variables using sprintf
      unsigned char ch_upd[100];
      sprintf((char*)ch_upd, "GET /update?api_key=YUXWKBKVODTNYW0R&field1=%d&field2=%d&field3=%.1f\r\n", Pulse, SpO2, temp);

      int url_length = strlen((char*)ch_upd);
      unsigned char send_thing[50];
```

```
        sprintf((char*)send_thing, "AT+CIPSEND=%d\r\n", url_length)
    ;
308     ESP32_SendBuffer(&myESP32, send_thing, strlen((char*)
    send_thing));
        usleep(1000000);
310     ESP32_SendBuffer(&myESP32, ch_upd, strlen((char*)ch_upd));
        usleep(1000000);
312     unsigned char Close_TCP[] = "AT+CIPCLOSE\r\n";
        ESP32_SendBuffer(&myESP32, Close_TCP, strlen((char*)
    Close_TCP));
314
}
316 // this function initializes MAX30102 sensor
int MAX30102_Init() {
318     // Declare a pointer to configuration data
     XIic_Config *ConfigPtr3;
320     // Obtain the IIC configuration data
     ConfigPtr3 = XIic_LookupConfig(MAX30102_IIC_DEVICE_ID);
322     // Configure and initialize the IIC driver
      XIic_CfgInitialize(&IicInstance, ConfigPtr3, ConfigPtr3->
    BaseAddress);
324     // Add a delay to ensure that the sensor has enough time to
    stabilize
     usleep(10000);
326     // Start the I2C driver
     XIic_Start(&IicInstance);
328     // Set the device address in write mode
     XIic_SetAddress(&IicInstance, XII_ADDR_TO_SEND_TYPE,
    MAX30102_I2C_ADDRESS);
330     usleep(10000);
     Sensor_Config();
332     return XST_SUCCESS;
}
334
// set up MAX30102 sensor configuration
336 void Sensor_Config(){
     // Clear the fifo Data Buffer if there is something
338     MAX30102_WriteReg(MAX30102_REG_FIFO_WR_PTR, 0x00);
     MAX30102_WriteReg(MAX30102_REG_FIFO_RD_PTR, 0x00);
340     MAX30102_WriteReg(MAX30102_REG_FIFO_OVERFLOW, 0x00);


342
     // Now, we will configure the sensor
344     /*
      Set LedMode (Options, Red Only, IR ONly , Red+IR both)
346      0x03= 00000011, B7= 0 (0 for normal mode,1 for power saving
    mode),B6 (1 for power rest , 0 if reset is done)
      B5,B4,B3 = No significance , B2,B1,B0 = 011 = to set in both
     led+IR mode
348     */
```

```
    MAX30102_WriteReg(MAX30102_REG_MODE_CONFIG, 0x03); // SET led
    mode to Red+IR
350
    //Configure the SpO2-related settings
352 /*
    B7= No significance, B6,B5 = ADC Range , B4,B3,B2 = Sample
   Rate , B1,B0 = PulseWidth
354   SpO2 ADCRange =4096 (Binary = 01), sample rate = 100 Hz(
   Binary = 001), PulseWidth 411(Binary = 11)
     So, final Binary Value to be written = 0 01 001 11 = 0x27 (
   in hexadecimal)
356  */
    MAX30102_WriteReg(MAX30102_REG_SPO2_CONFIG, 0x27);
358
    /*
360  * Sample Average = 4 , B7,B6,B5 = Sample Average[2,0]
     * B4 = FIFO_Rollover_enable
362  * B3,B2,B1,B0 = FIFO_A_FULL[3,0]
     */
364 MAX30102_WriteReg(REG_FIFO_CONFIG, 0x50); // in arduino code
   its 0x40 setting fifo rollover enable to 0 (disabling it)

366
    // Enable interrupt 1
368 MAX30102_WriteReg(MAX30102_INT_ENABLE_1, 0x40);

370
    /*
372  * Set led current for Red led and IR led
     */
374 MAX30102_WriteReg(MAX30102_REG_LED1_PA, 0x1F);     // 6.2 mA
    MAX30102_WriteReg(MAX30102_REG_LED2_PA, 0x1F);     // 6.2 mA
376 }

378 // this function is for writing data to register .
void MAX30102_WriteReg(u8 reg, u8 value) {
380 u8 buf[2];
    buf[0] = reg;
382 buf[1] = value;
    XIic_Send(IicInstance.BaseAddress, MAX30102_I2C_ADDRESS, buf,
    2, XIIC_STOP);
384 }
// read data from fifo data register of max30102 sensor
386 void ReadFifoData() {
    int num_sample_to_read;
388 int lim = 0;

390 u8 fifo_data[6];
    u8 fifo_data_reg_addr[1];
392 fifo_data_reg_addr[0] = MAX30102_FIFO_DATA;
```

```
    usleep(500000);
394 // Call the function to calculate heart rate and oxygen
    saturation level
    num_sample_to_read =  ReadFifoPointers();
396 while(sample<100 && lim <= 300) {
        if (num_sample_to_read ==0)
398     {
            usleep(150000);
400     }
        else if((100-sample) >= num_sample_to_read){
402         for (int i = 0; i<num_sample_to_read; i++) {
                                // Read the FIFO data registers
404                             XIic_Send(IicInstance.BaseAddress
    , MAX30102_I2C_ADDRESS, fifo_data_reg_addr, 1,
    XIIC_REPEATED_START);
                                XIic_Recv(IicInstance.BaseAddress
    , MAX30102_I2C_ADDRESS, fifo_data, 6, XIIC_STOP);
406                             // Parse the raw data into red
    and IR values
                                redData = (fifo_data[0] << 16) |
    (fifo_data[1] << 8) | fifo_data[2];
408                             irData = (fifo_data[3] << 16) | (
    fifo_data[4] << 8) | fifo_data[5];

410                             // Store the data in arrays
                                redDataArr[sample] = redData;
412                             irDataArr[sample] = irData;

414                             if (redData > 9000 || irData >
    9000) {
                                    redDataArr[sample] = redData;
416                                 irDataArr[sample] = irData;
                                    sample+=1 ;
418                             }
                            lim+=1;
420                        usleep(100000);
            }
422         usleep(150000);
            num_sample_to_read =  ReadFifoPointers();
424     }
        else {
426         for (int i = 0; i<(100-sample); i++) {
                                // Read the FIFO data registers
428                             XIic_Send(IicInstance.BaseAddress
    , MAX30102_I2C_ADDRESS, fifo_data_reg_addr, 1,
    XIIC_REPEATED_START);
                                XIic_Recv(IicInstance.BaseAddress
    , MAX30102_I2C_ADDRESS, fifo_data, 6, XIIC_STOP);
430
```

```
                                    // Parse the raw data into red
    and IR values
432                                 redData = (fifo_data[0] << 16) |
    (fifo_data[1] << 8) | fifo_data[2];
                                    irData = (fifo_data[3] << 16) | (
    fifo_data[4] << 8) | fifo_data[5];
434
                                    // Store the data in arrays
436                                 if (redData > 9000 || irData >
    9000) {
                                        redDataArr[sample] = redData;
438                                     irDataArr[sample] = irData;
                                        sample+=1 ;
440                                 }
                                    lim+=1;
442                               usleep(100000);
            }
444             usleep(150000);
                num_sample_to_read =  ReadFifoPointers();
446         }
     }
448 /*
     if (lim >=290) {
450         lim = 0;
          sample = 0;
452     }
     else
454     {
          maximum_heart_rate_and_oxygen_saturation(irDataArr, 100,
    redDataArr, &spo2, &spo2Valid, &heartRate, &heartRateValid);
456          if (spo2Valid != 1 || heartRateValid != 1)
           {
458             UpdateData();
           }
460         lim = 0;
          sample = 0;
462     }
    */
464     maximum_heart_rate_and_oxygen_saturation(irDataArr, 100,
    redDataArr, &spo2, &spo2Valid, &heartRate, &heartRateValid);
     lim = 0;
466     sample = 0;
}
468
// Update Fifo data of MAX30102 sensor , there is 100 sample ,
    and this function drops first 25 samples
470 // and add 25 more sample at the end . so there is now 100 sample
     , and it's updated
void UpdateData(){
472     int num_sample_to_read;
```

```
    u8 fifo_data[6];
474 u8 fifo_data_reg_addr[1];
    fifo_data_reg_addr[0] = MAX30102_FIFO_DATA;
476 usleep(500000);
    num_sample_to_read =  ReadFifoPointers();
478 int cnt = 0;
    int lim_ex =0;
480 while ((spo2Valid != 1 || heartRateValid != 1) && cnt < 5) {

482  while(sample<25 && lim_ex <= 75 ){

484      // shift last 75 samples to first 75 samples
         for (int i = 0; i < 75; i++) {
486         redDataArr[i] = redDataArr[i+25];
            irDataArr[i] = irDataArr[i+25];
488      }

490         if (num_sample_to_read == 0){usleep(150000);}
            else if((25-sample) >= num_sample_to_read){
492             for (int i = 0; i<num_sample_to_read; i++) {
                                    // Read the FIFO data
registers
494                                 XIic_Send(IicInstance.
BaseAddress, MAX30102_I2C_ADDRESS, fifo_data_reg_addr, 1,
XIIC_REPEATED_START);
                                    XIic_Recv(IicInstance.
BaseAddress, MAX30102_I2C_ADDRESS, fifo_data, 6, XIIC_STOP);
496                                 // Parse the raw data into
red and IR values
                                    redData = (fifo_data[0] <<
16) | (fifo_data[1] << 8) | fifo_data[2];
498                                 irData = (fifo_data[3] << 16)
 | (fifo_data[4] << 8) | fifo_data[5];
                                    // Store the data in arrays
500                                 if (redData > 9000 || irData
> 9000) {
                                        redDataArr[sample+75] =
redData;
502                                     irDataArr[sample+75] =
irData;
                                        sample+=1 ;
504                                 }
                                    lim_ex+=1;
506                               usleep(100000);
                }
508             usleep(150000);
                num_sample_to_read =  ReadFifoPointers();
510         }
            else {
512             for (int i = 0; i<(25-sample); i++) {
```

```
                                    // Read the FIFO data
    registers
514                                 XIic_Send(IicInstance.
    BaseAddress, MAX30102_I2C_ADDRESS, fifo_data_reg_addr, 1,
    XIIC_REPEATED_START);
                                    XIic_Recv(IicInstance.
    BaseAddress, MAX30102_I2C_ADDRESS, fifo_data, 6, XIIC_STOP);
516                                 // Parse the raw data into
    red and IR values
                                    redData = (fifo_data[0] <<
    16) | (fifo_data[1] << 8) | fifo_data[2];
518                                 irData = (fifo_data[3] << 16)
     | (fifo_data[4] << 8) | fifo_data[5];
                                    // Store the data in arrays
520                                 if (redData > 9000 || irData
    > 9000) {
                                        redDataArr[sample+75] =
    redData;
522                                     irDataArr[sample+75] =
    irData;
                                        sample+=1 ;
524                                 }
                                    lim_ex+=1;
526                               usleep(100000);
                }
528             usleep(150000);
                num_sample_to_read =  ReadFifoPointers();
530         }
        }
532     maxim_heart_rate_and_oxygen_saturation(irDataArr, 100,
    redDataArr, &spo2, &spo2Valid, &heartRate, &heartRateValid);
        sample = 0;
534     cnt += 1 ;
        lim_ex =0 ;
536 }
    cnt = 0;
538 }
// read fifo pointer register of max30102 sensor to determine how
    many sample available to read
540 int ReadFifoPointers() {
    u8 fifo_write_ptr;
542 u8 write_ptr_reg_addr = MAX30102_REG_FIFO_WR_PTR;
    // Read the FIFO write pointer
544 XIic_Send(IicInstance.BaseAddress, MAX30102_I2C_ADDRESS, &
    write_ptr_reg_addr, 1, XIIC_REPEATED_START);
    XIic_Recv(IicInstance.BaseAddress, MAX30102_I2C_ADDRESS, &
    fifo_write_ptr, 1, XIIC_STOP);
546 int diff = fifo_write_ptr;
    return diff;
548 }
```

```
void DemoCleanup() {
    DisableCaches();
}
void EnableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_DCACHE
   Xil_DCacheEnable();
#endif
#ifdef XPAR_MICROBLAZE_USE_ICACHE
   Xil_ICacheEnable();
#endif
#endif
}
void DisableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_DCACHE
   Xil_DCacheDisable();
#endif
#ifdef XPAR_MICROBLAZE_USE_ICACHE
   Xil_ICacheDisable();
#endif
#endif
}
```

A.0.2 Algorithm.h

```
#ifndef ALGORITHM_H
#define ALGORITHM_H
#include <stdint.h>
#include <stdbool.h>

// Includes necessary header files
#include <stdint.h>
#include <stdbool.h>

// Defines a boolean data type
#define true 1
#define false 0
// Sets the sampling frequency to 100 Hz
#define FS 25
// Sets the buffer size to 100
#define BUFFER_SIZE (4*FS)
// Sets the Moving Average Filter size to 4, and it is constant,
   so should not be changed
#define MA4_SIZE 4 // DONOT CHANGE
// Defines a macro that returns the minimum of two values
#define min(x,y) ((x) < (y) ? (x) : (y))
// SpO2 lookup table with 184 values
```

```
22 const uint8_t uch_spo2_table[184]={ 95, 95, 95, 96, 96, 96, 97,
     97, 97, 97, 97, 98, 98, 98, 98, 98, 99, 99, 99, 99,
                99, 99, 99, 99, 100, 100, 100, 100, 100, 100, 100,
     100, 100, 100, 100, 100, 100, 100, 100, 100,
24              100, 100, 100, 100, 99, 99, 99, 99, 99, 99, 99, 99,
      98, 98, 98, 98, 98, 98, 97, 97,
                97, 97, 96, 96, 96, 96, 95, 95, 95, 94, 94, 94, 93,
      93, 93, 92, 92, 92, 91, 91,
26              90, 90, 89, 89, 89, 88, 88, 87, 87, 86, 86, 85, 85,
      84, 84, 83, 82, 82, 81, 81,
                80, 80, 79, 78, 78, 77, 76, 76, 75, 74, 74, 73, 72,
      72, 71, 70, 69, 69, 68, 67,
28              66, 66, 65, 64, 63, 62, 62, 61, 60, 59, 58, 57, 56,
      56, 55, 54, 53, 52, 51, 50,
                49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37,
      36, 35, 34, 33, 31, 30, 29,
30              28, 27, 26, 25, 23, 22, 21, 20, 19, 17, 16, 15, 14,
      12, 11, 10, 9, 7, 6, 5,
                3, 2, 1 } ;
32 // Define two static arrays of 32-bit integers of size
     BUFFER_SIZE, used for storing the IR and
   static  int32_t an_x[ BUFFER_SIZE]; //ir
34 static  int32_t an_y[ BUFFER_SIZE]; //red
   // Define variables used in the algorithm for calculating the
     SpO2 value
36 uint32_t un_ir_mean,un_only_once ;
   int32_t k, n_i_ratio_count;
38 int32_t i, s, m, n_exact_ir_valley_locs_count, n_middle_idx;
   int32_t n_th1, n_npks, n_c_min;
40
   void maxim_peaks_above_min_height(int32_t *pn_locs, int32_t *
     n_npks, int32_t *pn_x, int32_t n_size, int32_t n_min_height)
42 {
     int32_t i = 1, n_width;
44   *n_npks = 0; // Initialize number of peaks found to zero

46   while (i < n_size - 1) // Loop through all elements in the
      signal array
     {
48     if (pn_x[i] > n_min_height && pn_x[i] > pn_x[i - 1]) // Check
      if the current value is greater than minimum height and
     greater than the previous value
       {
50       n_width = 1; // Initialize width of peak to one

52       // Check if there are any more values with same amplitude
     as current value
         while (i + n_width < n_size && pn_x[i] == pn_x[i + n_width
     ])
54         n_width++;
```

```
56        // If current value is a peak and number of peaks found is
      less than 15, record its location and increment number of peaks
          if (pn_x[i] > pn_x[i + n_width] && (*n_npks) < 15)
58        {
            pn_locs[(*n_npks)++] = i;
60          i += n_width + 1; // Move to the next position after the
      peak
          }
62        else
            i += n_width; // If current value is not a peak, move to
      the next position
64      }
        else
66        i++; // If current value is less than or equal to minimum
      height or less than or equal to previous value, move to the
      next position
    }
68  }

70
    void maxim_sort_ascend(int32_t *pn_x, int32_t n_size) {
72    int32_t i, j, n_temp; // declare variables

74    // iterate through the array
      for (i = 1; i < n_size; i++) {
76      n_temp = pn_x[i]; // store current element in temp variable

78      // iterate backwards from current element to the start of the
        array
        for (j = i; j > 0 && n_temp < pn_x[j-1]; j--) {
80        pn_x[j] = pn_x[j-1]; // shift elements to the right
        }
82      pn_x[j] = n_temp; // insert the current element in the
      correct position
    }
84  }

86  /*
     * Sorts the indices of an array of values in descending order.
88   *
     * Arguments:
90   *  - pn_x: pointer to the array of values to sort indices for.
     *  - pn_indx: pointer to an array of indices to be sorted.
92   *  - n_size: size of the array of values and indices.
     *
94   * This function uses an insertion sort algorithm to sort the
       indices in
     * descending order based on the values in the array. The sorted
       indices
```

```
 * are stored in the pn_indx array. The values in the pn_x array are
 * not modified by this function.
 */
void maxim_sort_indices_descend(int32_t *pn_x, int32_t *pn_indx, int32_t n_size)
{
  int32_t i, j, n_temp;

  for (i = 1; i < n_size; i++) {
    n_temp = pn_indx[i];

    // Insertion sort to sort indices in descending order based on values
    for (j = i; j > 0 && pn_x[n_temp] > pn_x[pn_indx[j-1]]; j--)
      pn_indx[j] = pn_indx[j-1];

    pn_indx[j] = n_temp;
  }
}

void maxim_remove_close_peaks(int32_t *pn_locs, int32_t *pn_npks, int32_t *pn_x, int32_t n_min_distance)
{
  int32_t i, j, n_old_npks, n_dist;

  // Sort the peak locations in descending order based on their corresponding values in the input data
  maxim_sort_indices_descend( pn_x, pn_locs, *pn_npks );

  // Loop through each peak location and remove any peak that is too close to a previously detected peak
  for ( i = -1; i < *pn_npks; i++ ){
    n_old_npks = *pn_npks;
    *pn_npks = i+1;
    for ( j = i+1; j < n_old_npks; j++ ){
      // Calculate the distance between two adjacent peaks
      n_dist =  pn_locs[j] - ( i == -1 ? -1 : pn_locs[i] );
      // If the distance is greater than the minimum distance, keep the peak
      if ( n_dist > n_min_distance || n_dist < -n_min_distance )
        pn_locs[(*pn_npks)++] = pn_locs[j];
    }
  }

  // Sort the peak locations in ascending order
  maxim_sort_ascend( pn_locs, *pn_npks );
}

/**
```

```
 * This function finds the indices of the peaks in an input array
    'pn_x' that are greater than or
 * equal to the input 'n_min_height' and have a minimum distance
   between them of 'n_min_distance'.
 * The number of peaks found is limited to 'n_max_num' peaks.
 *
 * @param pn_locs Pointer to an array to store the peak indices
   found in 'pn_x'
 * @param n_npks Pointer to the number of peaks found
 * @param pn_x Pointer to the input array
 * @param n_size Size of the input array
 * @param n_min_height Minimum height of the peak to be
   considered
 * @param n_min_distance Minimum distance between two peaks to be
    considered
 * @param n_max_num Maximum number of peaks to be considered
 */
void maxim_find_peaks( int32_t *pn_locs, int32_t *n_npks,
   int32_t  *pn_x, int32_t n_size, int32_t n_min_height, int32_t
   n_min_distance, int32_t n_max_num )
{
  // Find the indices of peaks that are greater than or equal to
   the minimum height
  maxim_peaks_above_min_height( pn_locs, n_npks, pn_x, n_size,
   n_min_height );

  // Remove the indices of peaks that are too close to each other
  maxim_remove_close_peaks( pn_locs, n_npks, pn_x, n_min_distance
   );

  // Limit the number of peaks found to the maximum number of
   peaks
  *n_npks = min( *n_npks, n_max_num );
}


void maxim_heart_rate_and_oxygen_saturation(uint32_t *
   pun_ir_buffer, int32_t n_ir_buffer_length, uint32_t *
   pun_red_buffer, int32_t *pn_spo2, int8_t *pch_spo2_valid,
                int32_t *pn_heart_rate, int8_t *pch_hr_valid) {

  int32_t an_ir_valley_locs[15] ;
  int32_t n_peak_interval_sum;
  int32_t n_y_ac, n_x_ac;
  int32_t n_spo2_calc;
  int32_t n_y_dc_max, n_x_dc_max;
  int32_t n_y_dc_max_idx, n_x_dc_max_idx;
  int32_t an_ratio[5], n_ratio_average;
  int32_t n_nume, n_denom ;
  un_ir_mean =0;
```

```
for (k=0 ; k<n_ir_buffer_length ; k++ ) un_ir_mean +=
 pun_ir_buffer[k] ;
un_ir_mean =un_ir_mean/n_ir_buffer_length ;
for (k=0 ; k<n_ir_buffer_length ; k++ )
  an_x[k] = -1*(pun_ir_buffer[k] - un_ir_mean) ;
for(k=0; k< BUFFER_SIZE-MA4_SIZE; k++){
  an_x[k]=( an_x[k]+an_x[k+1]+ an_x[k+2]+ an_x[k+3])/(int)4;}
n_th1=0;
for ( k=0 ; k<BUFFER_SIZE ;k++){
  n_th1 +=  an_x[k];}
n_th1=  n_th1/ ( BUFFER_SIZE);
if( n_th1<30) n_th1=30; // min allowed
if( n_th1>60) n_th1=60; // max allowed
for ( k=0 ; k<15;k++) an_ir_valley_locs[k]=0;
maxim_find_peaks( an_ir_valley_locs, &n_npks, an_x, BUFFER_SIZE
 , n_th1, 4, 15 );//peak_height, peak_distance, max_num_peaks
n_peak_interval_sum =0;
if (n_npks>=2){
  for (k=1; k<n_npks; k++) n_peak_interval_sum += (
 an_ir_valley_locs[k] -an_ir_valley_locs[k -1] ) ;
  n_peak_interval_sum =n_peak_interval_sum/(n_npks-1);
  *pn_heart_rate =(int32_t)( (FS*60)/ n_peak_interval_sum );
  *pch_hr_valid  = 1;}
else  {
  *pn_heart_rate = -999; // unable to calculate because # of
 peaks are too small
  *pch_hr_valid  = 0;}
for (k=0 ; k<n_ir_buffer_length ; k++ )  {
    an_x[k] =  pun_ir_buffer[k] ;
    an_y[k] =  pun_red_buffer[k] ;}
n_exact_ir_valley_locs_count =n_npks;
n_ratio_average =0;
n_i_ratio_count = 0;
for(k=0; k< 5; k++) an_ratio[k]=0;
for (k=0; k< n_exact_ir_valley_locs_count; k++){
  if (an_ir_valley_locs[k] > BUFFER_SIZE ){
    *pn_spo2 =  -999 ; // do not use SPO2 since valley loc is
 out of range
    *pch_spo2_valid  = 0;
    return; }}
for (k=0; k< n_exact_ir_valley_locs_count-1; k++){
  n_y_dc_max= -16777216 ;
  n_x_dc_max= -16777216;
  if (an_ir_valley_locs[k+1]-an_ir_valley_locs[k] >3){
      for (i=an_ir_valley_locs[k]; i< an_ir_valley_locs[k+1]; i
 ++){
        if (an_x[i]> n_x_dc_max) {n_x_dc_max =an_x[i];
 n_x_dc_max_idx=i;}
        if (an_y[i]> n_y_dc_max) {n_y_dc_max =an_y[i];
 n_y_dc_max_idx=i;} }
```

```
218       n_y_ac= (an_y[an_ir_valley_locs[k+1]] - an_y[
    an_ir_valley_locs[k] ] )*(n_y_dc_max_idx -an_ir_valley_locs[k])
    ; //red
          n_y_ac=  an_y[an_ir_valley_locs[k]] + n_y_ac/ (
    an_ir_valley_locs[k+1] - an_ir_valley_locs[k])  ;
220       n_y_ac=  an_y[n_y_dc_max_idx] - n_y_ac;    // subracting
    linear DC compoenents from raw
          n_x_ac= (an_x[an_ir_valley_locs[k+1]] - an_x[
    an_ir_valley_locs[k] ] )*(n_x_dc_max_idx -an_ir_valley_locs[k])
    ; // ir
222       n_x_ac=  an_x[an_ir_valley_locs[k]] + n_x_ac/ (
    an_ir_valley_locs[k+1] - an_ir_valley_locs[k]);
          n_x_ac=  an_x[n_y_dc_max_idx] - n_x_ac;      // subracting
    linear DC compoenents from raw
224       n_nume=( n_y_ac *n_x_dc_max)>>7 ; //prepare X100 to
    preserve floating value
          n_denom= ( n_x_ac *n_y_dc_max)>>7;
226       if (n_denom>0  && n_i_ratio_count <5 &&  n_nume != 0)
          { an_ratio[n_i_ratio_count]= (n_nume*100)/n_denom ; //
    formular is ( n_y_ac *n_x_dc_max) / ( n_x_ac *n_y_dc_max) ;
228         n_i_ratio_count++; } }}
  maxim_sort_ascend(an_ratio, n_i_ratio_count);
230  n_middle_idx= n_i_ratio_count/2;
  if (n_middle_idx >1)
232    n_ratio_average =( an_ratio[n_middle_idx-1] +an_ratio[
    n_middle_idx])/2; // use median
  else
234    n_ratio_average = an_ratio[n_middle_idx ];
  if( n_ratio_average>2 && n_ratio_average <184){
236     n_spo2_calc= uch_spo2_table[n_ratio_average] ;
    *pn_spo2 = n_spo2_calc ;
238    *pch_spo2_valid  = 1;//  float_SPO2 =  -45.060*
    n_ratio_average* n_ratio_average/10000 + 30.354 *
    n_ratio_average/100 + 94.845 ;  // for comparison with table
    }
240  else{
    *pn_spo2 =  -999 ;
242    *pch_spo2_valid  = 0;}}

244 #endif /* ALGORITHM_H */
```