Hochschule Bremerhaven

M.Sc Embedded System Design

PULSE OXIMETER PROJECT REPORT

WINTER SEMESTER: 2022-23

SUBMITTED BY

Abhinit Shetty	39405
Aylin Eda Üstü	39440
Lisa Margreat Rajendran	39376
Raj Choksi	39433
Vaishnavi shah	39367
Yashwant Rao	39423

UNDER THE GUIDANCE OF

Prof. Dr.-Ing. Kai Müller

handed on May 3, 2023

Declaration

We, the undersigned members of Pulse Oximeter project, hereby declare that we have worked upon the base code provided by Professor Kai Müller to develop our project and have made significant modifications to the code to achieve our desired results.We acknowledge that the base code provided by Professor Kai Müller was instrumental in helping us develop our project, and we are grateful for their contribution.

Group Members :

1. Abhinit Shetty - 39405
2. Aylin Eda Üstü - 39440
3. Lisa Margreat Rajendran - 39376
4. Raj Choksi - 39433
5. Vaishnavi Shah - 39367
6. Yashwant Rao - 39423

Date: 3rd May 2023

Contributions

Matriculation no.	Name	Chapters
39405	Abhinit Shetty	3.2.7, 3.2.8, 3.3, 5 and 6.3
39440	Aylin Eda Üstü	Programming to Control and Calibrate the Sensor (Attached Document)
39376	Lisa Margreat Rajendran	1
39433	Raj Choksi	10, 11 and 12
39367	Vaishnavi Shah	8 and 9
39423	Yashwant Rao	3, 4, 6.1, 6.2 and 7

Contents

1 Introduction

Pulse oximetry is a non-invasive medical technique that measures the oxygen saturation levels in a person's blood. It works by emitting light from a device placed on a patient's fingertip or earlobe, and measuring the amount of light that is absorbed by the blood vessels in the tissue. Based on the amount of light absorbed, the device can calculate the oxygen saturation levels in the blood.

Designing and analyzing pulse oximetry involves understanding the underlying physics and biology of the measurement technique, as well as the engineering principles behind the device. This includes knowledge of optics, signal processing, and data analysis techniques.

The design of a pulse oximeter involves selecting appropriate wavelengths of light to be used, designing the sensor that emits and detects the light, and developing algorithms for signal processing and data analysis. The accuracy and reliability of the pulse oximeter can be improved by optimizing these factors.

Analysis of pulse oximetry data involves processing the signals obtained from the device to extract relevant information about the patient's oxygen saturation levels. This may involve filtering out noise, correcting for motion artifacts, and interpreting the data in the context of the patient's overall health.

Overall, pulse oximetry is a valuable medical tool that can provide important information about a patient's health. Effective design and analysis of pulse oximeters can improve their accuracy and reliability, leading to better patient outcomes.

2 Project Lifecycle

The project lifecycle of designing and analyzing pulse oximetry can be divided into several stages, which include:

1. Requirements gathering: In this stage, the requirements for the pulse oximeter are gathered. This includes understanding the clinical requirements, such as the range of oxygen saturation levels to be measured and the accuracy required, as well as the technical requirements, such as the wavelengths of light to be used and the sensor design.

2. Design and prototyping: Based on the requirements gathered in the previous stage, the design of the pulse oximeter is developed. This includes selecting the appropriate components, such as the light sources and detectors, and designing the sensor and signal processing algorithms. A prototype of the pulse oximeter is then developed and tested to ensure that it meets the requirements.

3. Testing and validation: In this stage, the pulse oximeter prototype is tested and validated to ensure that it meets the clinical and technical requirements. This may include laboratory testing to validate the accuracy of the measurements, as well as clinical testing to evaluate the performance of the pulse oximeter in real-world conditions.

4. Regulatory approval: Before the pulse oximeter can be used in clinical practice, it must receive regulatory approval from the appropriate authorities. This includes demonstrating that the pulse oximeter is safe and effective for use in patients, and meeting the regulatory requirements for medical devices.

5. Production and deployment: Once regulatory approval has been obtained, the pulse oximeter can be manufactured and deployed for use in clinical practice. This may involve developing a manufacturing process and establishing quality control procedures to ensure that the pulse oximeters are produced to the required standards.

6. Maintenance and support: Once the pulse oximeter is in use, it may require maintenance and support to ensure that it continues to perform to the required standards. This may include regular calibration, software updates, and technical support for users.

Throughout the project lifecycle, it is important to ensure that the pulse oximeter meets the clinical and technical requirements, and that it is safe and effective for use in patients. Close collaboration between the engineering and clinical teams is important to ensure that the pulse oximeter is designed and deployed in a way that meets the needs

of patients and healthcare providers.

2.1 Significance of Pulse oximetry in Medical systems

Pulse oximetry is an essential diagnostic tool in medical systems used to measure the level of oxygen saturation in a patient's blood. It is a non-invasive method that involves the use of a pulse oximeter, which emits light through the skin to measure the amount of oxygen bound to hemoglobin in the blood.

Here are some key significances of pulse oximetry in medical systems:

1. Early detection of respiratory problems: Pulse oximetry is a quick and simple way to measure oxygen saturation levels in the blood. It allows medical professionals to detect early changes in oxygen levels, which may indicate respiratory problems, even before the patient shows symptoms of respiratory distress.

2. Monitoring of critical care patients: Pulse oximetry is frequently used in critical care units to monitor patients on mechanical ventilation or oxygen therapy. Continuous monitoring of oxygen saturation levels in the blood provides vital information for healthcare providers to adjust treatment or ventilation settings as required.

3. Evaluation of sleep disorders: Pulse oximetry is also used in sleep laboratories to diagnose and assess sleep disorders such as sleep apnea. It can detect changes in oxygen saturation levels during sleep, which can help in the diagnosis of the condition.

4. Management of chronic conditions: Pulse oximetry is useful in managing chronic conditions such as chronic obstructive pulmonary disease (COPD), asthma, and heart failure. Regular monitoring of oxygen saturation levels in the blood allows healthcare providers to evaluate the effectiveness of treatment and adjust therapy as needed.

5. Safe administration of anesthesia: Pulse oximetry is a standard practice during anesthesia to monitor the patient's oxygen saturation levels and ensure their safety during the procedure.

In summary, pulse oximetry is an essential tool in medical systems, as it allows for early detection of respiratory distress, monitoring of critical care patients, evaluation of sleep apnea, management of COPD, and safe administration of anesthesia. It is a simple and non-invasive way to measure oxygen saturation levels in the blood, and its importance cannot be overstated in ensuring patient safety and well-being.

2.2 Development of Pulse Oximetry

2.2.1 The Origins of Pulse Oximetry

The concept of pulse oximetry can be traced back to the early 1900s when German scientist Karl Matthes developed a non-invasive device that measured blood oxygen saturation levels. However, it wasn't until the 1930s that scientists began to focus on pulse oximetry as a practical tool for measuring blood oxygen levels.

In 1935, German physician Kurt Ecker developed a device that used a photometer to measure changes in light absorption in the skin caused by variations in blood oxygen levels. However, this device was invasive and required blood to be drawn from the patient.

It wasn't until the 1940s that scientists developed a non-invasive method for measuring blood oxygen saturation levels. In 1949, John Scott Haldane, a Scottish physiologist, developed a device that used a finger clip to measure changes in light absorption. This was the first pulse oximeter, and it used the principle of spectrophotometry to measure the amount of oxygen bound to hemoglobin in the blood.

2.2.2 The Evolution of Pulse Oximetry

In the 1960s, pulse oximetry technology began to evolve rapidly. In 1964, Takuo Aoyagi, a Japanese bioengineer, developed a new type of pulse oximeter that used two different wavelengths of light to measure the absorption of oxygenated and deoxygenated blood. This innovation greatly improved the accuracy of pulse oximetry readings and made the technology more reliable.

The principle behind pulse oximetry is based on the fact that hemoglobin, the protein in red blood cells that carries oxygen, absorbs light differently depending on whether it is oxygenated or deoxygenated. Hemoglobin with oxygen-bound (oxyhemoglobin) absorbs more red light and less infrared light, while hemoglobin without oxygen-bound (deoxyhemoglobin) absorbs more infrared light and less red light. By using two different wavelengths of light, pulse oximeters can determine the amount of oxygen in the blood by measuring the absorption of both red and infrared light.

In the 1970s, the first commercial pulse oximeter was introduced by Biox Corporation. This device used a single wavelength of light and was primarily used for research purposes. However, by the 1980s, pulse oximetry had become an essential diagnostic tool in the medical industry. The technology had evolved to include a range of features such as alarms to alert healthcare providers of changes in oxygen saturation levels, and

new algorithms to improve accuracy and reliability.

In the 1990s, pulse oximetry continued to evolve, with the introduction of new devices that could be used in a range of settings, from hospitals to clinics and even at home. The development of portable pulse oximeters made it easier for healthcare providers to monitor patients' oxygen saturation levels outside of a clinical setting, and the technology became more affordable and widely available.

Today, pulse oximetry is used in a range of medical settings, from emergency rooms to critical care units and even in-home care. It is a vital tool in monitoring patients with respiratory conditions such as chronic obstructive pulmonary disease (COPD) and asthma, as well as in monitoring patients undergoing surgery or receiving oxygen therapy. Pulse oximetry has come a long way since its inception in the early 1900s. The technology has evolved from invasive devices to non-invasive, portable, and highly accurate devices that are essential in modern medical care. The scientific principles behind pulse oximetry are based on the interaction of light with biological tissue, and the development of new algorithms and technologies has greatly improved the accuracy and reliability of pulse oximetry readings.

3 Hardware Interface Description and Testing

The research and development on this project have been done with only a few hardware units as a whole. However, the individual hardware unit has quite a complex structure. The two major hardware blocks are ARTY 7 development board which hists the Xilinx All programmable SoC-7000 and interface pcb developed at the IAE ,HS Bremerhaven. A detailed study has been performed before and during the project development, to effectively utilize the hardware resources. Other components are also looked at carefully to use it.

1. The ARTY 7 Development Board
2. Interface PCB
3. Probe
4. Power Adaptor
5. USB Cable

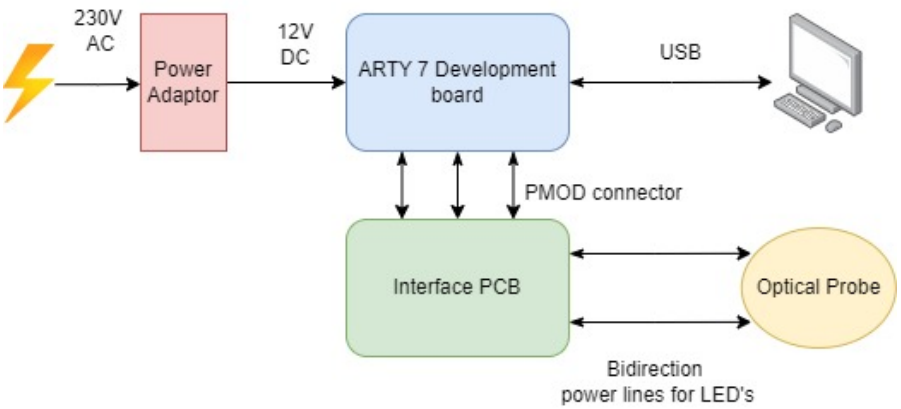


Figure 3.1: Overall Block Diagram

3.1 Arty Board

The Arty Z7 is a development board designed around the Zynq-7000™ All Programmable System-on-Chip (AP SoC) from Xilinx. The Zynq-7000 architecture tightly integrates a dual-core, 650 MHz ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic.

3.1.1 Development Board Features

1. **ZYNQ Processor**
   - 650MHz dual-core Cortex-A9 processor
   - DDR3 memory controller and 4 High-Performance AXI3 Slave ports
   - High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO
   - Low-bandwidth peripheral controller: SPI, UART, CAN, I2C
   - Programmable from JTAG, Quad-SPI flash, and microSD card
2. **Memory**
   - 512MB DDR3 with 16-bit bus @ 1050Mbps
   - 16MB Quad-SPI Flash
   - microSD slot
3. **Power**
   - Powered from USB or any 7-15V external power source
4. **USB**
   - USB-JTAG Programming circuitry
   - USB-UART bridge
5. **Switches, Push-buttons, and LEDs**
   - 4 push-buttons
   - 2 slide switches
   - 4 LEDs
   - 2 RGB LEDs
6. **Expansion Connectors**

- Two Pmod ports
- 16 Total FPGA I/O

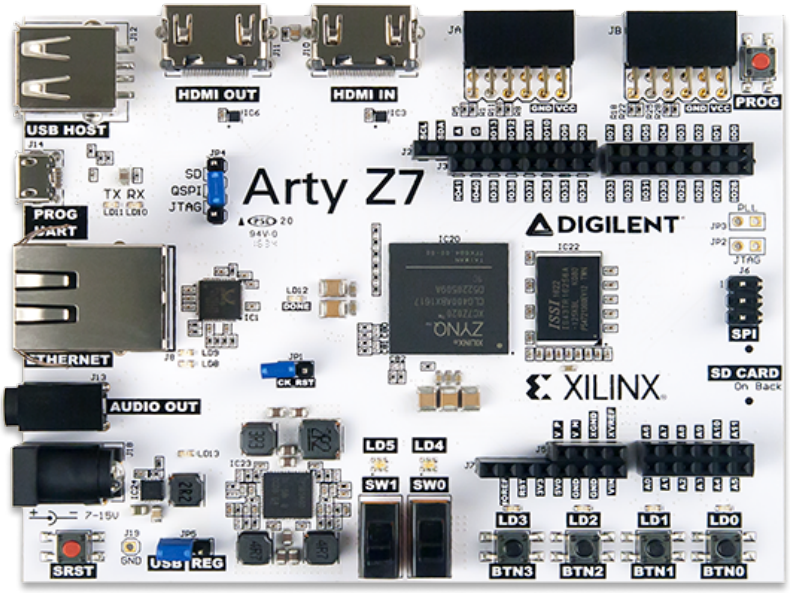


Figure 3.2: Arty Board

3.1.2 FPGA Specifications

- Logic slices 13,300
- LUTs 53,200
- Flip-Flops 106,400
- Block RAM 630 KB
- DSP Slices 220
- Clock Resources
  1. Zynq PLL with 4 outputs
  2. 125 MHz external clock

3.1.3 ZYNQ Configuration

The Zynq-7020 has a processor, which acts as a master to the programmable logic fabric and all other on-chip peripherals in the processing system. So, the Zynq boot process is similar to that of a microcontroller to an FPGA. This process involves the processor loading and executing a Zynq Boot Image, which includes a First Stage Bootloader (FSBL), a bitstream for configuring the programmable logic (optional), and a user application.

The Arty Z7 supports three different boot modes: microSD, Quad SPI Flash, and JTAG. The boot mode is selected using the Mode jumper (JP4), which affects the state of the Zynq configuration pins after power-on.

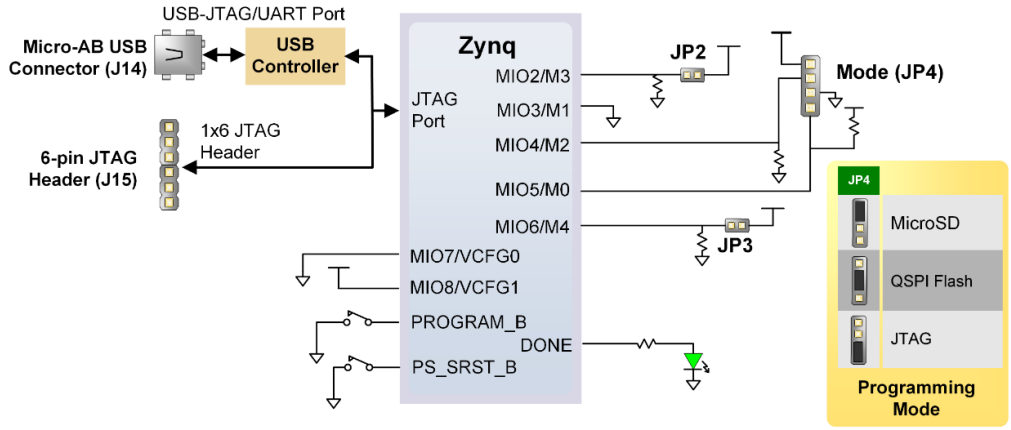


Figure 3.3: Programming Mode (Ref: ARTY Z7 reference manual)

Note : The JTAG programming is used during the project development.

3.1.4 Clock Sources

The Arty Z7 development board employs a 50 MHz clock signal as the basis for generating the necessary clocks for the Zynq Processing System subsystem.

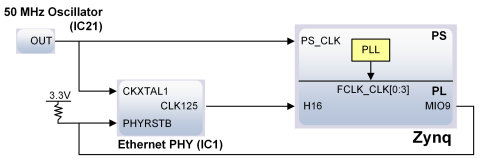


Figure 3.4: Clock Sources

This clock signal enables the processor to operate at a maximum frequency of 650 MHz. The figure 3.4 illustrates the clocking scheme employed by the Arty Z7.
The Zynq Processing System has a dedicated Phase-Locked Loop (PLL) that can generate up to four reference clocks, each with settable frequencies, to be used for clocking custom logic implemented in the Programmable Logic (PL). Moreover, the Arty Z7 provides an external 125 MHz reference clock directly to the PL via pin H16. This external reference clock allows the PL to operate independently of the PS.

3.2 Interface Board

The IAE team at HS Bremerhaven has designed a single PCB that encompasses all required external hardware components, which is referred to as the interface board. This board can be conveniently connected to the ARTY Z7 board via the Pmod connectors. Additionally, the optical finger probe can be easily connected to the interface board's DB9 connector.

3.2.1 Overview

In addition to several ICs, the interface board comprises various critical sections whose details are elaborated in subsequent sections of the report. However, for a comprehensive understanding of the board's functioning, it is important to outline the ICs' numbers and functionalities, which are described below.

S. no	IC	Description	Reference
1	AD8608	Quad Op Amp	U1
2	AD8608		U2
3	MCP6S21	Single Channel PGA	U3
4	AD5624	4 Channel 12 Bit DAC	U4
5	AD7887	12 Bit ADC	U5
6	74HC540	8-bit inverting buffer	U6
7	74HC541	8-bit non-inverting buffer	U7

Figure 3.5: Description of IC

3.2.2 Digital to Analog Converter

The interface board comes with an AD5624 Digital to Analog converter (DAC) that has four channels and a resolution of 12 bits. The DAC can be controlled using the Serial Peripheral Interface (SPI) communication protocol. The four channels are utilized to regulate the intensity of the RED/IR LED lights, facilitate visualization of the heartbeat LED, and cancel out ambient light.

1. **DAC Features**
   - SPI clock utp 50 Mhz
   - DIN - 24 bit input shift register
   - On Chip 1.25v/2.5v reference
   - Operating Voltage 2.7v to 5.5v

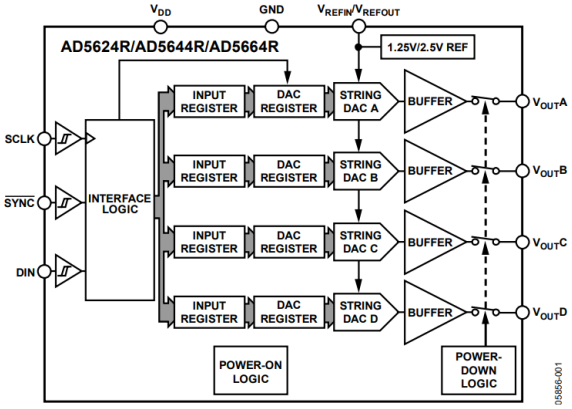


Figure 3.6: Functional Block Diagram, Ref: Datasheet

2. **DAC Architecture** The DAC architecture consists of a string DAC followed by an output buffer amplifier. The below figure shows DAC architecture.

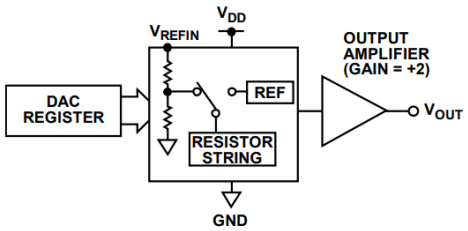


Figure 3.7: DAC stages, Ref: Datasheet

3. **DAC Serial Communication** The DAC utilizes a 3-wire serial interface which conforms to the standard SPI communication protocol. However, it is important to note that the DAC is designed to only receive data and does not transmit any information back to the master. Hence, it can be inferred that the communication between the DAC and master is unidirectional.

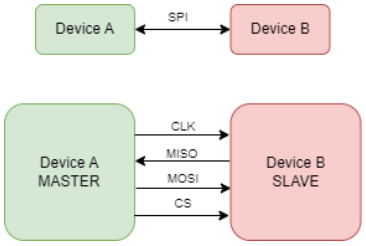


Figure 3.8: SPI protocol

- Clock Polarity = 0
- Clock Phase = 1
- SPI Mode = 1

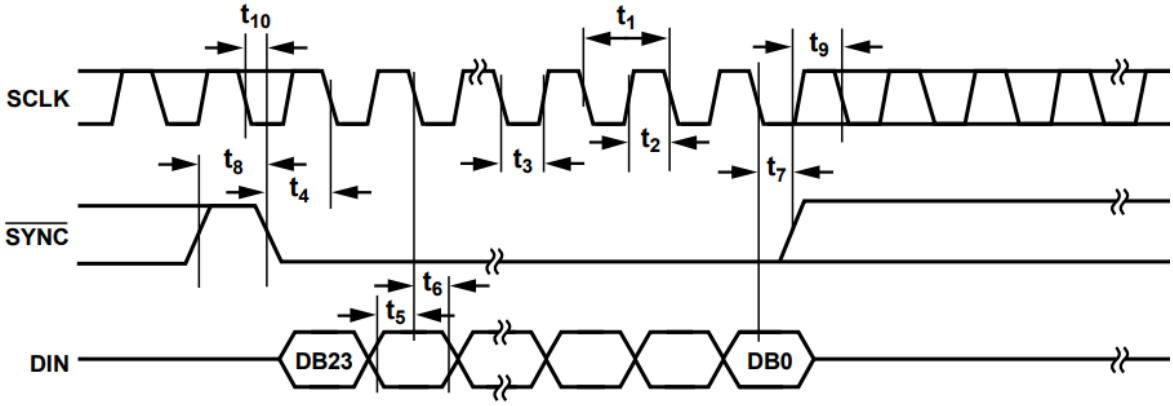


Figure 3.9: DAC Serial Communication, Ref: Datasheet

4. **DAC Commands**

- Set internal reference 1.25v
  By default, the on-chip reference is turned off when power is applied. However, there is a programmable bit called DB0 in the control register that can be used to turn the reference on or off through software.

  Bit 0 in Internal Register is used to set up the internal reference. Zero (0):

Reference off (default) and One (1): Reference on.

Command to set internal reference: 0x380001

DB23 to DB22 (MSB)	DB21	DB20	DB19	DB18	DB17	DB16	DB15 TO DB1	DB0 (LSB)
X	1	1	1	X	X	X	X	1
Don't Care	Commands bits(C2 to C0)			Address bits(A2 to A0)			Don't Care	Internal Reference

Figure 3.10: DAC Register Contents for Internal Reference set

Command to clear internal reference: 0x380000

DB23 to DB22 (MSB)	DB21	DB20	DB19	DB18	DB17	DB16	DB15 TO DB1	DB0 (LSB)
X	1	1	1	X	X	X	X	0
Don't Care	Commands bits(C2 to C0)			Address bits(A2 to A0)			Don't Care	Internal Reference

Figure 3.11: DAC Register Contents for Internal Reference clear

- Set LDAC Register

The behavior of the LDAC bit register is dependent on its setting. When set to a low state, the corresponding DAC registers are latched, allowing changes to the input registers without affecting the contents of the DAC registers. On the other hand, when set to a high state, the **DAC registers become transparent**, and the contents of the input registers are transferred to them on the falling edge of the 24th SCLK pulse.

Command to set LDAC for all 4 channels : 0x30000F

DB23 to DB22 (MSB)	DB21	DB20	DB19	DB18	DB17	DB16	DB15 TO DB4	DB3	DB2	DB1	DB0 (LSB)
X	1	1	0	X	X	X	X	1	1	1	1
Don't Care	Commands bits(C2 to C0)			Address bits(A2 to A0)			Don't Care	DAC D	DAC C	DAC B	DAC A
								1/0 for respective channel			

Figure 3.12: DAC Register Contents for LDAC set for all 4 channels

- Reset DAC outputs The DAC includes a built-in function for resetting its software, which is triggered using Command 101. This command is exclusively reserved for the software reset function. Additionally, the software reset command offers two different modes that can be selected based on the software configuration of bit DB0 in the control register.

  a) DB0 = 0 ; Reset only DAC Input Shift registers.

  b) DB1 = 1 ; Reset all registers.

  Command: Software Reset (101)

DB23 to DB22 (MSB)	DB21	DB20	DB19	DB18	DB17	DB16	DB15 TO DB1	DB0 (LSB)
X	1	0	1	X	X	X	X	1
Don't Care	Commands bits(C2 to C0)			Address bits(A2 to A0)			Don't Care	Software Reset

Figure 3.13: Software Reset Command

- Voltage out at Channel A

  Command: Write and update DAC channel n (011)
  Address: Channel A (000)
  Data : 1000mv/(2500mv/4096) = 1638, (011001100110) binary

  Command for 1000mv @ Channel A: 0x186660

DB23 to DB22 (MSB)	DB21	DB20	DB19	DB18	DB17	DB16	DB15 - DB04 D11 - D0	DB03 - DB00
X	0	1	1	0	0	0	011001100110	X
Don't Care	Commands bits(C2 to C0)			Address bits(A2 to A0)			12 bit Data	Don't Care

Figure 3.14: DAC Register Contents for 1000mv @ Channel A

- Voltage out at Channel B

  Command: Write and update DAC channel n (011)
  Address: Channel B (001)
  Data : 1000mv/(2500mv/4096) = 1638, (011001100110) binary

  Command for 1000mv @ Channel B: 0x196660

DB23 to DB22 (MSB)	DB21	DB20	DB19	DB18	DB17	DB16	DB15 - DB04 D11 - D0	DB03 - DB00
X	0	1	1	0	0	1	011001100110	X
Don't Care	Commands bits(C2 to C0)			Address bits(A2 to A0)			12 bit Data	Don't Care

Figure 3.15: DAC Register Contents for 1000mv @ Channel B

- Voltage out at Channel C

  Command: Write and update DAC channel n (011)
  Address: Channel C (010)
  Data : 1000mv/(2500mv/4096) = 1638, (011001100110) binary

  Command for 1000mv @ Channel C: 0x1A6660

DB23 to DB22 (MSB)	DB21	DB20	DB19	DB18	DB17	DB16	DB15 - DB04 D11 - D0	DB03 - DB00
X	0	1	1	0	1	0	011001100110	X
Don't Care	Commands bits(C2 to C0)			Address bits(A2 to A0)			12 bit Data	Don't Care

Figure 3.16: DAC Register Contents for 1000mv @ Channel C

- Voltage out at Channel D

  Command: Write and update DAC channel n (011)
  Address: Channel D (100)
  Data : 1000mv/(2500mv/4096) = 1638, (011001100110) binary

  Command for 1000mv @ Channel D: 0x1C6660

DB23 to DB22 (MSB)	DB21	DB20	DB19	DB18	DB17	DB16	DB15 - DB04 D11 - D0	DB03 - DB00
X	0	1	1	1	0	0	011001100110	X
Don't Care	Commands bits(C2 to C0)			Address bits(A2 to A0)			12 bit Data	Don't Care

Figure 3.17: DAC Register Contents for 1000mv @ Channel D

3.2.3 Analog To Digital Converter

The interface board is equipped with a 12-bit ADC that samples both IR and RED light data samples. Both light types are consecutively sampled on the same channel, utilizing appropriate data acquisition techniques. A comprehensive description of the ADC can be found in the attached report by Aylin Eda .

3.2.4 Programmable Gain Array

Amplifying the signals is crucial for signal processing, particularly since biomedical signals are typically low in amplitude. To achieve this, a programmable gain amplifier is utilized on the interface board. This amplifier offers eight distinct gain options, ranging from +1 to +32, including +1, +2, +4, +5, +8, +10, +16, or +32, to suit various signal amplification requirements. A comprehensive description of the PGA can be found in the attached report by Aylin Eda .

3.2.5 Buffers

Buffers are a fundamental component of electronic circuits that are designed to isolate and protect signals from various sources of interference. The primary function of a buffer is to provide high input impedance and low output impedance, which helps to prevent signal distortion and reduce the loading effect on other components in the circuit. two buffers are used in the interface board.

- Inverting Buffer (74HC540) This buffer plays a crucial role in the circuit by serving as an inverter for the 3 SPI chip select signals, namely those of the ADC, PGA, and DAC. It is also responsible for controlling the H-bridge driver of the LED's.
- Non-Inverting Buffer (74HC541) This buffer serves the dual purpose of isolating the MOSI, MISO, and SCLK signals of the SPI, as well as enabling visualization of the IR and RED LED signals.

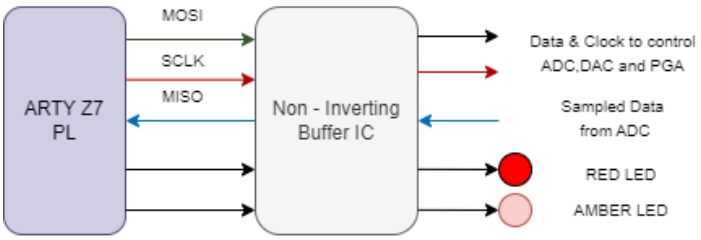


Figure 3.18: Use of buffer

3.2.6 RED/IR LED Driver

This section serves as a means to regulate the power and intensities of both the IR and RED LEDs through controlled current flow. An H-bridge configuration is employed, allowing for bi-directional current flow to the LEDs. The overall operation is explained

with the help of an example for IR LED.
Note : The MOSFETs (Q2 & Q1)are used as a switch and therefore the MOSFETs are referred as switches in the below description.

- IR LED :1, RED LED :0, Some voltage on DAC channel B - from ARTY board
- IR LED :0 , RED LED :1 - signals after inverting Buffer
- Switch Q2B ON: Input 0, Switch Q3A ON: Input 1
- Switch Q2A OFF: Input 0, Switch Q3B OFF: Input 1
- Switch Q1A ON: Input 1, Switch Q1B OFF: Input 0

When the gate of switch Q1A receives a high input, it gets closed, causing the non-inverting terminal (pin no.3) of the AD860 operational amplifier to be set to zero. This results in the output of the UA1 op-amp becoming zero, which turns off the Q2A switch and puts it in a high impedance state.

On the other hand, when the gate of switch Q1B receives a low input, the switch remains open, causing the non-inverting terminal (pin no.5) of the AD860 operational amplifier to be set to a voltage level equal to Channel b output/4.25. This results in the output of the UAB op-amp becoming high, which switches on the Q3A switch.

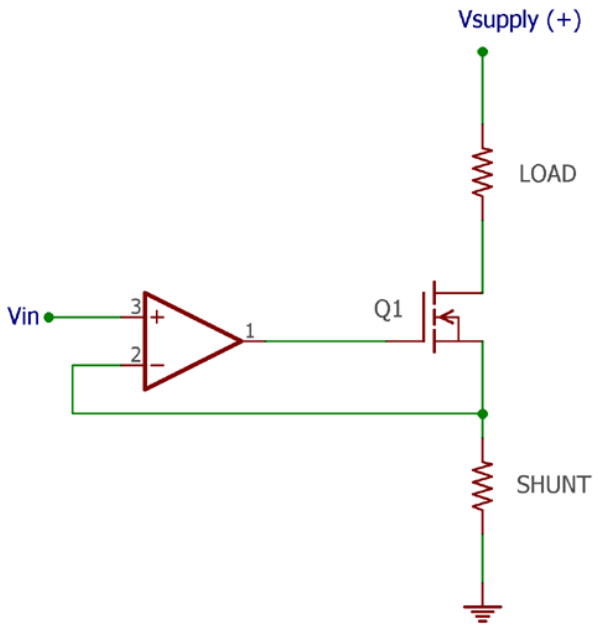


Figure 3.19: Voltage Control Current Source

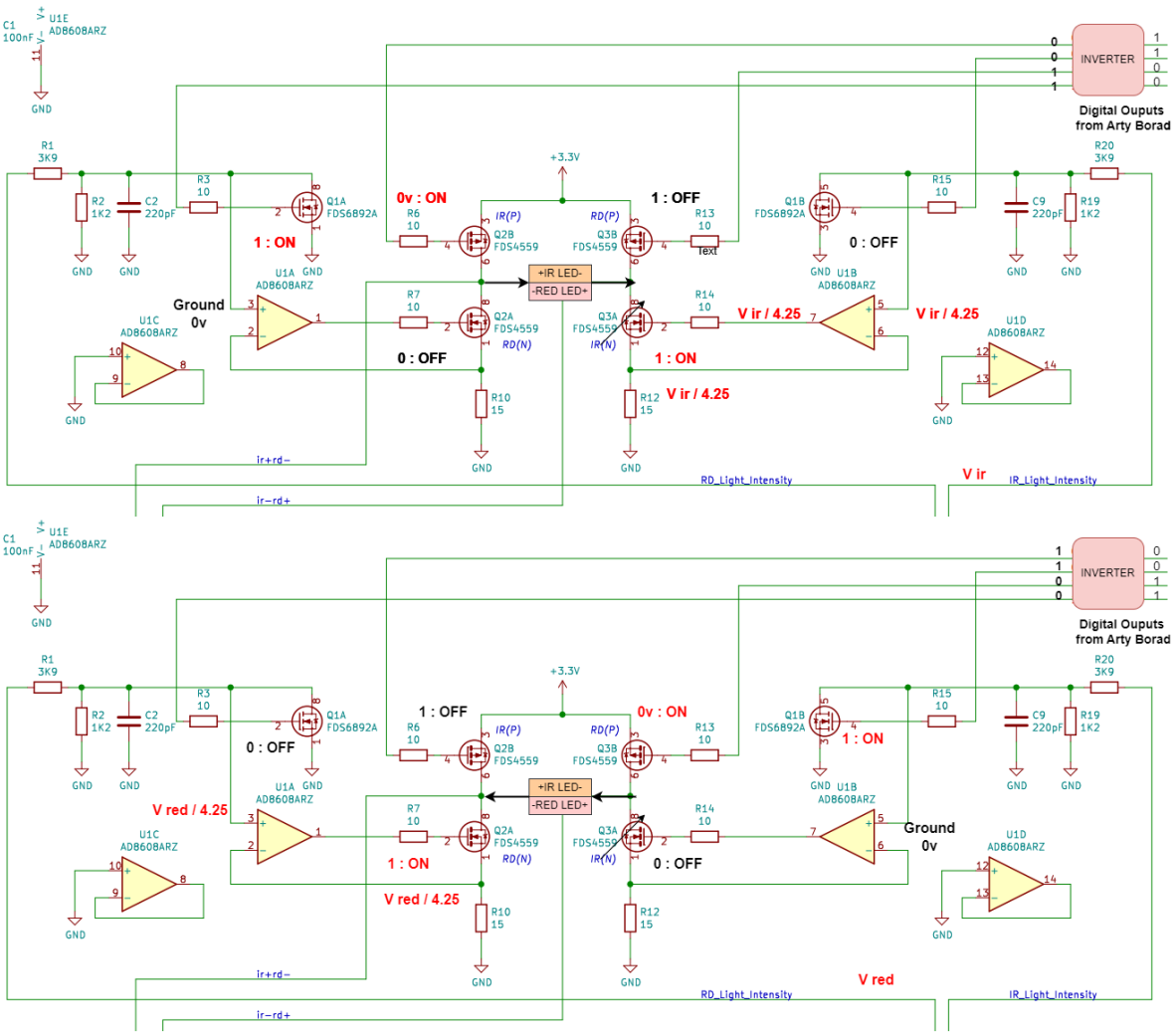


Figure 3.20: RED and IR LED on states

3.2.7 Current to Voltage Converter

A current to voltage converter will produce a voltage proportional to the given current. This circuit is required if the measuring instrument is capable only of measuring voltages and the need to measure the current output.

Note : The inverting terminal of the I-V converter Op-amp is not directly linked to the cathode terminal of the photodetector. The cathode terminal output is received via Pin 9 of the DB9 connection, which links the sensor probe to the board.

Applying KCL at node which is at inverting terminal of Op-Amp we get :

$$I_x = I + I_f \tag{3.1}$$

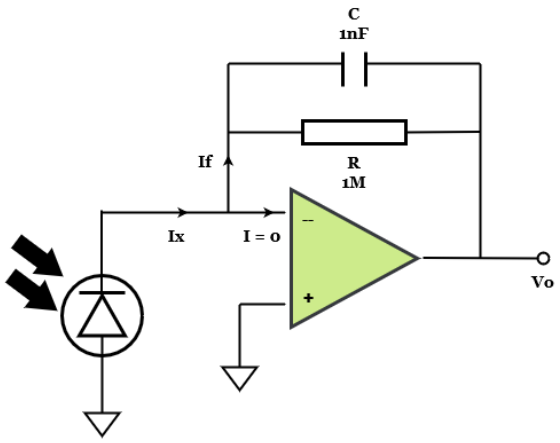


Figure 3.21: Current - Voltage converter circuit

Also current flowing into the input terminal of opamp is almost equal to zero (I = 0), hence

$$I_x = I_f \tag{3.2}$$

This current flows through the Feedback Resistor R which creates the output voltage Vo which is given by Ohm's law :

$$V_o = I_x.R \tag{3.3}$$

Feedback Resistor (R) : The input signal from the photo detector is of the order of a few micro amps. A high-value feedback resistance (R) of the order of 1M is used to convert the input current into an output signal of a few volts. At this stage, maximum gain is provided because adding more gain after the transimpedance stage generally produces poor noise performance. The signal-to-noise ratio (SNR) is improved with higher R since noise increases with the square root of R and signal increases linearly with R.

Feedback Capacitor (C) : Feedback capacitor (C) is used to minimize peak gain and improve stability. The use of Cf also limits bandwidth, reducing noise. A capacitance value of 1nF is used as Cf. Larger values of feedback capacitance limit the operating bandwidth.

3.2.8 Ambient Light Cancellation

The purpose of this section is to mitigate the impact of external lighting on the probe's photoreceptors, as this can result in inaccurate readings. While the probe is already

designed to block out external light, the implementation of ambient light cancellation allows for reliable detection of oxygen saturation and heartbeat. A basic differential amplifier is utilized to subtract the pre-determined ambient light prior to calibration.
A pulse oximeter uses a differential amplifier to cancel out the effect of ambient light. The photodetector produces a signal proportional to light intensity. This signal is converted to a voltage using a current-voltage converter and then amplified by the differential amplifier, which cancels out any common-mode signals like ambient light. Before calibration, the initial reading is taken when the finger is in the probe with LEDs off and subtracted from subsequent readings.

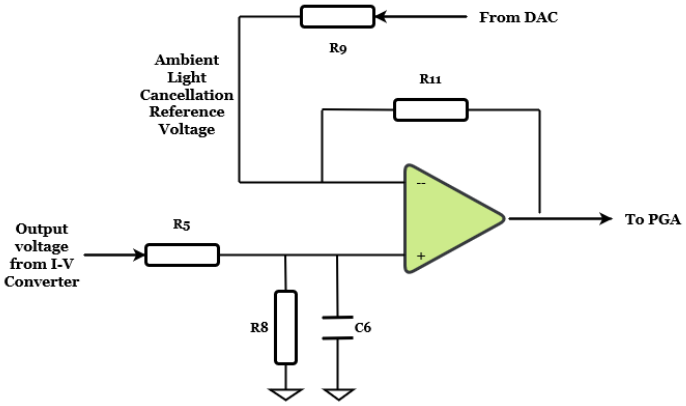


Figure 3.22: Ambient Light Cancellation

3.3 Probe

The finger probe is a device equipped with a photo detector and two Light Emitting Diodes (LEDs) - one emitting red light at a wavelength of 660 nm and the other emitting infrared light at a wavelength of 895 nm. The device is designed to be connected to a board for incoming signals, which instructs it to turn on the Red and IR LEDs at specific intervals.
The device is also equipped with a DB9 connector that allows for the transmission of signals for further processing. This interface enables the transfer of data between the finger probe and other devices or systems that are compatible with the DB9 connector. The finger probe is typically used in medical or healthcare applications for monitoring various parameters, such as oxygen saturation levels in the blood, and can be integrated into larger monitoring systems for real-time tracking of patient vital signs.

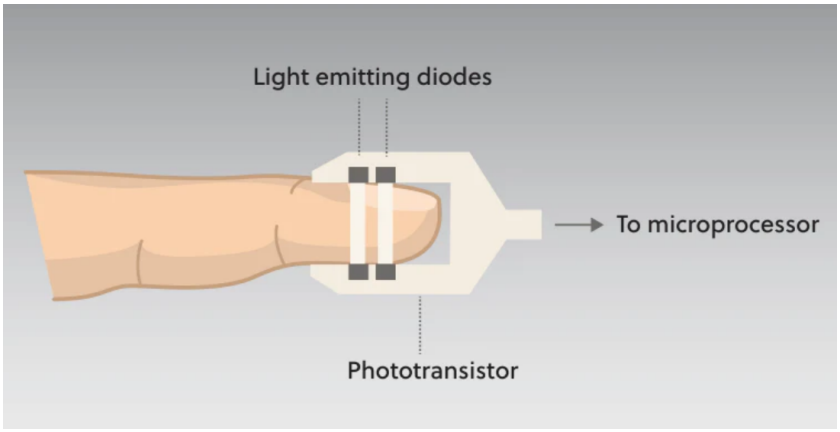


Figure 3.23: Transmissive-type Finger Probe

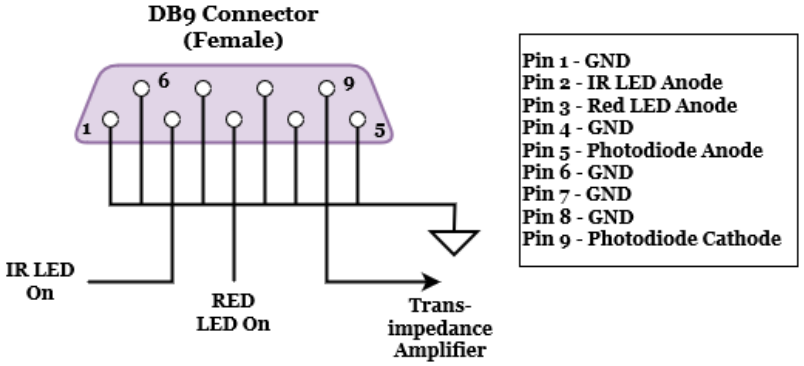


Figure 3.24: DB9 Connector Pinout

3.4 Testing and Results

3.4.1 DAC outputs

The digital-to-analogue converter (DAC AD5624) plays a crucial role in the process of data acquisition. Specifically, the reference voltage for the analogue-to-digital converter (ADC AD7887)is supplied by the output of the DAC Vref.

The DAC itself generates a voltage of 1.25V at pin no. 10, which is then further amplified by a non-inverting amplifier with a gain of 2.5, resulting in an output voltage of 3.125V. Additionally, the DAC is programmed to produce varying voltages at the four different channels.

- 1.25v Reference Voltage: DAC command : 0x380001 0

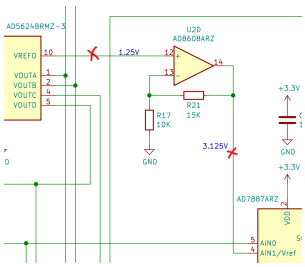


Figure 3.25: Measured points in schematic

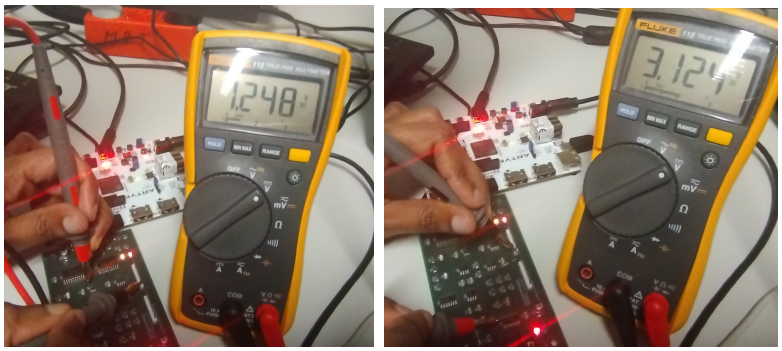


Figure 3.26: V ref 1.24v and Amplified 3.124v measured with multimeter

- Channel A: RED light intensity: DAC command- 0x1813E 318mv
- Channel B: IR light intensity: DAC command- 0x1913E 318mv

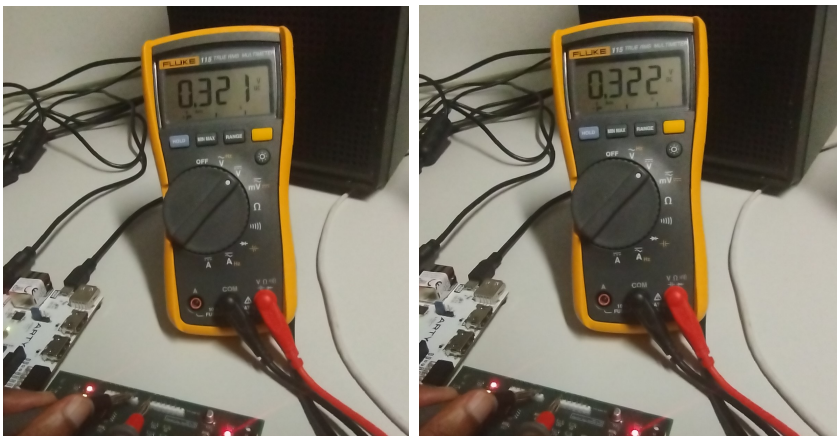


Figure 3.27: 321mv @ CH-A and 312 mv @ CH-B measured with multimeter

- Channel C: Heart Beat LED

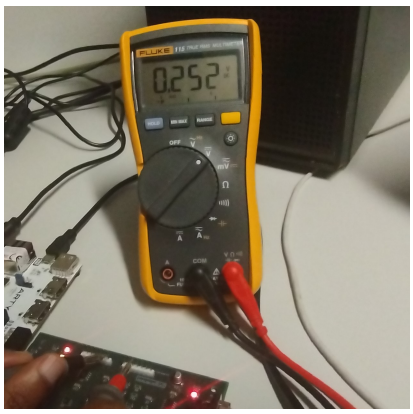


Figure 3.28: 251mv @ CH-C measured with multimeter

3.4.2 PGA I/O

The input and output test of PGA.

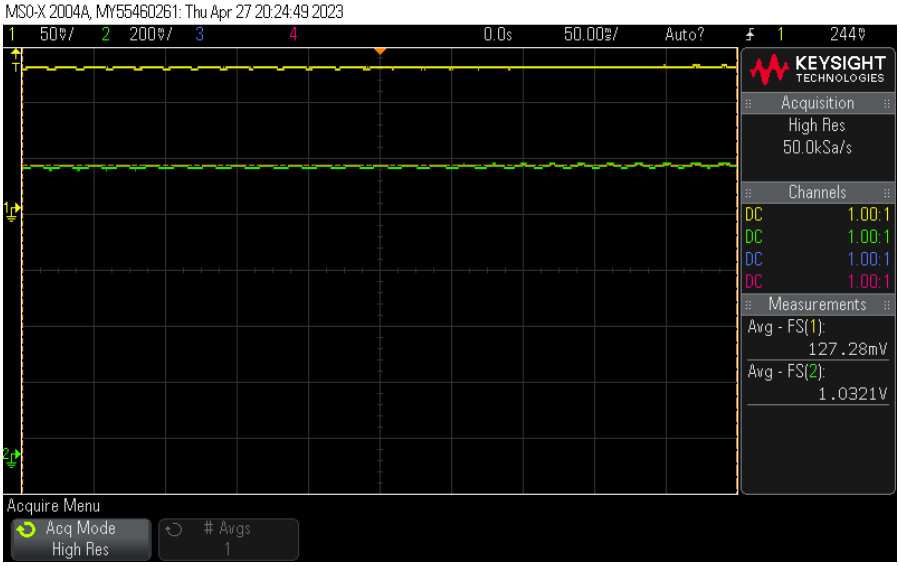


Figure 3.29: Input 127.28 mv and Output 1.0321 v measured with DSO

- 8 x amplification Test : PGA command 0x00004004
- DSO Channel 1 - Voltage per division 500mv : Input
- DSO Channel 2 - Voltage per division 200mv : Output

3.4.3 RED/IR LED Intensity Control

- The voltage output from the DAC for controlling the intensity of the IR LED is directed to a voltage divider consisting of resistors R19 (1.2k) and R20 (3.9k) in the schematic.
- This voltage divider reduces the input voltage by a factor of 0.2353 (i.e. 1200/(1200+3900)).
- For the purpose of this discussion, let's assume that Va represents the voltage output of DAC channel 2, Vb represents the voltage at resistor R12 (15 ohms), and I represent's the current flowing through resistor R12.

$$Vb = Va/4.25$$

$$I = Vb/15$$

During the experiment, the DAC voltage was altered, and Va and Vb were measured. The current was calculated using the formula Vb/15.

As shown in the graph, there is a linear correlation between the voltage across the resistor and the LED since they are in series, and the LED experiences the same current flow as the resistor. The same experiment was conducted on the IR and red LEDs, resulting in identical findings.

Va Voltage output DAC (mv)	Vb Voltage at resistor (15 ohm) (mv)	Va/Vb	I Current (mA)
53,4	12,6	4,24	0,840
103,4	24,3	4,26	1,620
153,3	36,1	4,25	2,407
203,3	47,8	4,25	3,187
253,2	59,6	4,25	3,973
503,1	118,3	4,25	7,887
752,0	176,8	4,25	11,787

Figure 3.30: Experiment Data

- The current across the LED's should not exceed **20mA**.
- So, voltage across current limiting resistor(15 ohm) is = 20mA*15 ohm = **300mv**
- The voltage from DAC is reduced(due to the voltage divider) by a factor of **4,25**.

- So, Maximum voltage allowed from DAC output = 300mv*4,25 = **1275mv**
- 1275mv/ (2500mv/4096)= 2088, we can **limit to 2000** integer value
- Maximum integer value form DAC is 1275/ (2500/4096)= 2088, we can limit to **2000** integer value
- The **2000** integer value corresponds to a DAC voltage (2000* (2500/4096)) = 1200.7mv. This voltage is reduced by 4.25, 1200./4.25 = 287.22 mv across resistor and current of (287.22/15) **19.1 mA.**

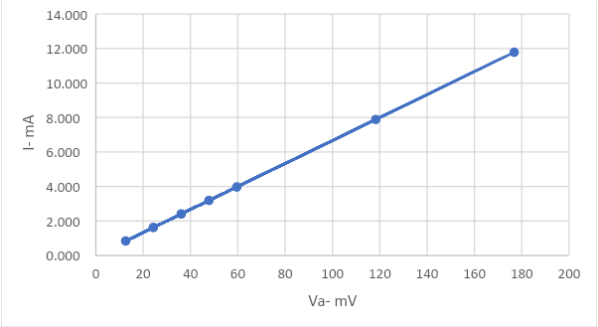


Figure 3.31: R12 resistor VI curve

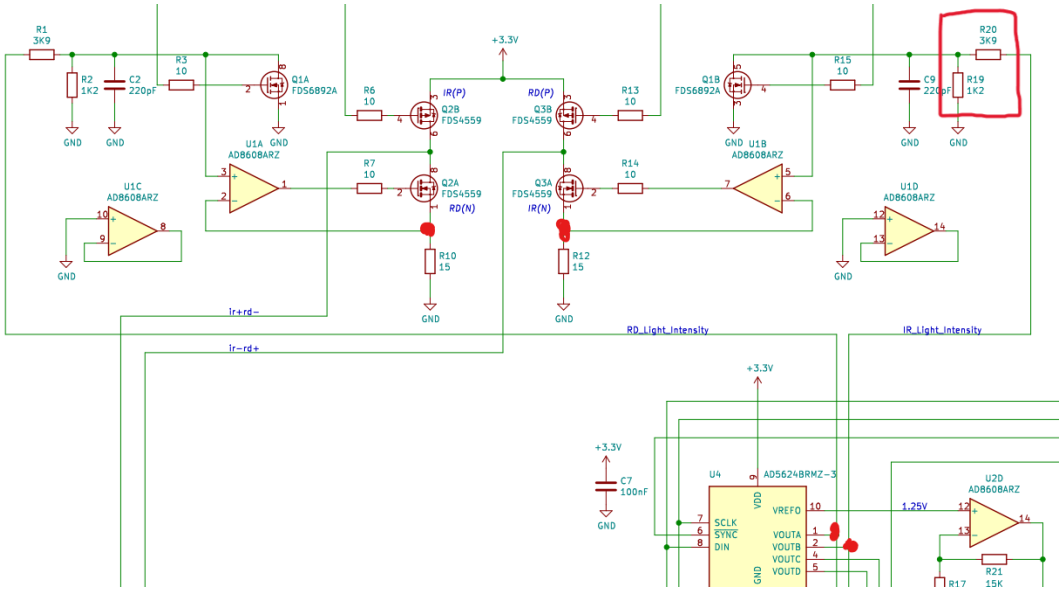


Figure 3.32: Part of interface board schematic

4 IP Design and Development

4.1 Introduction

This chapter outlines the custom-built IP used for the SPI communication and control of digital input/output. Initially, a basic SPI state machine was developed for the DAC during the project's early stages. This state machine was later modified to enable communication with all three peripherals, including the DAC, ADC, and PGA. Since the reference for the ADC was produced by the DAC, the project started with the DAC interface.

Explicit use of registers was included in the design to store and retrieve data. The ZYNQ architecture and AXI bus are also explained. The simulation was carried out to test the design's functionality before implementation. Additionally, testing was done to ensure that the design met the project requirements and specifications.

A hazard analysis was also conducted to identify static hazards associated with the SPI lines in the design. Finally, an implementation report analysis was conducted to get insight into the use of PL resources.

4.2 Block Design: Top Level Interface

The top-level design is a high-level representation of the blocks that configure the System on Chip (SoC). The block design consists of four important blocks, as listed below:

1. **ZYNQ7 Processing System** - this block is responsible for executing software and managing the peripherals within the system.
2. **Process System Reset** - this block generates a reset signal that ensures the system starts up in a known state
3. **AXI Interconnect** - this block enables communication between different modules and components within the system using the AXI bus.

4. **IP** - this block is the custom built IP used for SPI communication and control of digital inputs and outputs.

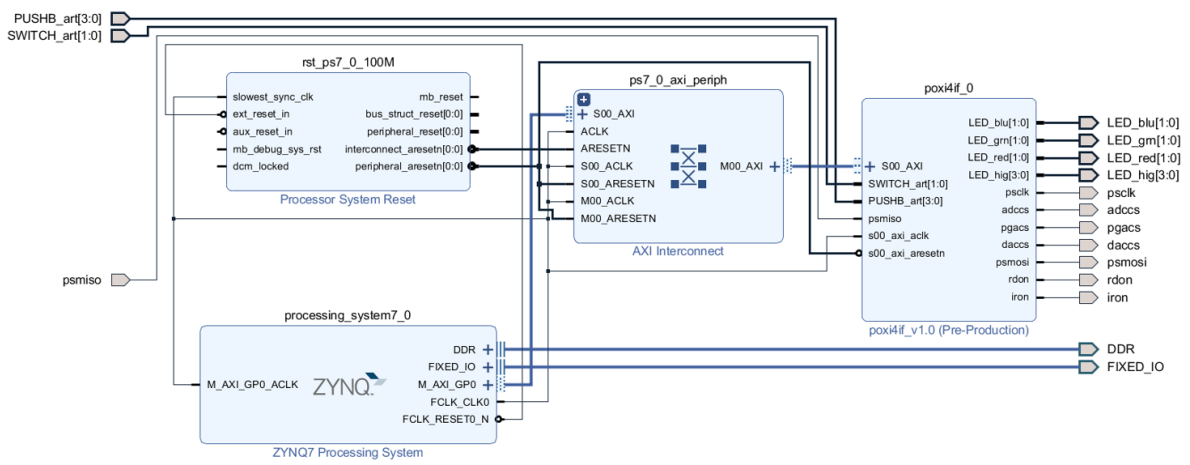


Figure 4.1: Block Design

4.2.1 Clock Configuration

The processing system's clock source is provided by a **50MHz** external source, which is scaled up to **650MHz**. This 650MHz clock is used as the base clock for the processor. On the other hand, the clock for the custom-built IP in the programmable logic (PL) is selected as **100MHz**.

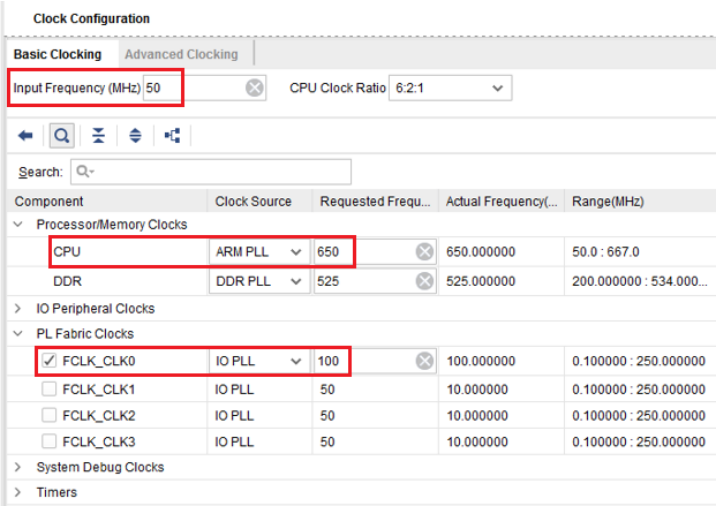


Figure 4.2: Clock Configuration

4.2.2 UART Configuration

To enable communication between the GUI and the SoC, the serial communication interface UART is used. UART0 is selected for the available peripherals, with a baud rate of 115200.

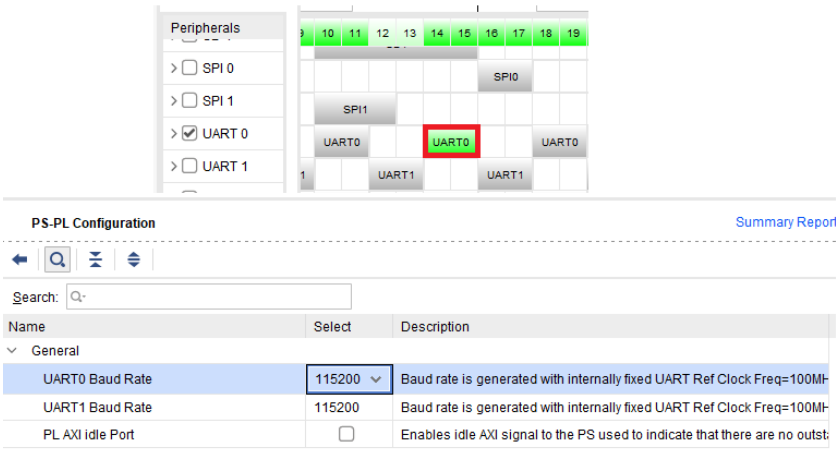


Figure 4.3: UART Configuration

4.2.3 ZYNQ

The Zynq-7000 family of devices from Xilinx features a dual-core ARM Cortex-A9 processor coupled with programmable logic, making it a powerful platform for implementing complex system designs. The processor runs at a clock speed of up to 1 GHz and provides a rich set of peripherals including Gigabit Ethernet, USB 2.0, SDIO, UART, SPI, and I2C. The programmable logic consists of configurable logic blocks (CLBs), digital signal processing (DSP) slices, and embedded memory blocks, which can be interconnected to create custom digital circuits.

4.2.4 AXI

AXI stands for Advanced eXtensible Interface, and the current version is AXI4, which is part of the ARM AMBA® 3.0 open standard. Many devices and IP blocks produced by third party manufacturers and developers are based on this standard. The AMBA standard was originally developed by ARM for use in microcontrollers, with the first version being released in 1996.

4.3 AXI Slave Registers Description

The 4 registers (each 4 bytes) has been defined for data transfer between PS(C code) & IP(VHDL).

- Slave Register 0 - SPI configurations
- Slave Register 1 - SPI tx data to DAC,ADC and PGA
- Slave Register 2 - RED,b IR and uLED control
- Slave Register 3 - SPI rx data from ADC

The VHDL code for the definition of slave registers is defined below.

```
---- Number of Slave Registers 4
signal slv_reg0 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg1 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg2 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
signal slv_reg3 :std_logic_vector(C_S_AXI_DATA_WIDTH-1 downto 0);
```

Figure 4.4: Slave reg Definitions

4.3.1 Slave Register 0

This register is used to start and stop the SPi communication. The diffrent SPI modes, clock rerquncy, chip sellect option and data length are configured with SL0

- Bit 0 - start SPI : WRITE from C
- Bit 2 & 1 - SPI modes : WRITE from C
  (00->mode0, 01 ->mode1, 10->mode2->2, 11->mode3)
- Bit 4 & 3 - data length: WRITE from C
  (00->8bit, 01 ->16bit, 10->16->24bit, 11->32bit)
- Bit 6 & 5 - chip select : WRITE from C
  (00->DAC_CS, 01 ->PGA_CS, 10->16->ADC_CS, 11->IC_shutdown)
- Bit 7 - clock configuration : WRITE from C
  (0->1Mhz, 1->2Mhz)
- Bit 8 - stop SPI : WRITE from VHDL

```
-------------------------------------------
---- Clock Polarity select loops ----
-------------------------------------------
   IF cPol='0' THEN
     sclk_last <= sclk_i;
   ELSIF cPol='1' THEN --CPOL=1
     sclk_last <= NOT sclk_i;
   END IF;
-------------------------------------------
---- Data Length select loops ----
-------------------------------------------
   IF dataLength= "00" THEN
     USPI_SIZE <= 8;
   ELSIF dataLength= "01" THEN
     USPI_SIZE <= 16;
   ELSIF dataLength= "10" THEN
     USPI_SIZE <= 24;
   END IF;
-------------------------------------------
---- IC's chip select loops ----
-------------------------------------------
   IF ic_slct="00" THEN --DAC
     dac_i <= not scsq_i;
     adc_i <= '0';
     pga_i <= '0';
   ELSIF ic_slct="01" THEN --PGA
     pga_i <=  not scsq_i;
     dac_i <= '0';
     adc_i <= '0';
   ELSIF ic_slct="10" THEN --ADC
     adc_i <=  not scsq_i;
     dac_i <= '0';
     pga_i <= '0';
   ELSE --Shut Down All IC's
     adc_i <= '0';
     dac_i <= '0';
     pga_i <= '0';
   END IF;
-------------------------------------------
   ---- Clock Frequency select loops ----
   -------------------------------------------
      IF clk_freq='0' THEN
        CLK_GENERATER <= 25;
      ELSIF clk_freq='1' THEN
        CLK_GENERATER <= 50;
      END IF;

END PROCESS scmb_proc;
```

Figure 4.5: Slave reg 0 VHDL code

4.3.2 Slave Register 1

This register is used to store MOSI(sdo) data. For DAC it is 24 bytes.

- trn_data => slv_reg1(23 downto 0)

4.3.3 Slave Register 2

This register is used to control LED's.

- slv_reg2(1)= iron
- slv_reg2(0)= rdon
- slv_reg2(5 downto 2)= uLED_hig

4.3.4 Slave Register 3

This register is used to receive MISO(sdi) data. For testing is kept 24 bytes.

- reg_data_out <= slv_reg3(C_S_AXI_DATA_WIDTH-1 downto 24) & rcvData;

4.3.5 Slave Register Read and Write

The AXI VHDL file for our project already defines the read and write operations for the slave. The process responsible for implementing these operations can be viewed below.

```
-- Implement memory mapped register select and read logic generation
-- Slave register read enable is asserted when valid address is available
-- and the slave is ready to accept the read address.
slv_reg_rden <= axi_aready and S_AXI_ARVALID and (not axi_rvalid) ;

process (slv_reg3, slv_reg2, slv_reg1, slv_reg0, axi_araddr, S_AXI_ARESETN, slv_reg_rden,
         uSWITCH_art, uPUSHB_art, spi_done, rcvData)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    -- Address decoding for reading registers
    loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
    case loc_addr is
      when b"00" =>
        reg_data_out <= slv_reg0(C_S_AXI_DATA_WIDTH-1 downto 9)& spi_done & slv_reg0(7 downto 0);
      when b"01" =>
        reg_data_out <= slv_reg1;
      when b"10" =>
        reg_data_out <= slv_reg2;
      when b"11" =>
        reg_data_out <= slv_reg3(C_S_AXI_DATA_WIDTH-1 downto 24) & rcvData;
      when others =>
        reg_data_out  <= (others => '0');
    end case;
end process;
```

Figure 4.6: Slave reg write

```
process (S_AXI_ACLK)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
  if rising_edge(S_AXI_ACLK) then
    if S_AXI_ARESETN = '0' then
      slv_reg0 <= (others => '0');
      slv_reg1 <= (others => '0');
      slv_reg2 <= (others => '0');
      slv_reg3 <= (others => '0');
    else
      loc_addr := axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
      if (slv_reg_wren = '1') then
        case loc_addr is
          when b"00" =>
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if ( S_AXI_WSTRB(byte_index) = '1' ) then
                -- Respective byte enables are asserted as per write strobes
                -- slave registor 0
                slv_reg0(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
              end if;
            end loop;
          when b"01" =>
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if ( S_AXI_WSTRB(byte_index) = '1' ) then
                -- Respective byte enables are asserted as per write strobes
                -- slave registor 1
                slv_reg1(byte_index*8+7 downto byte_index*8) <= S_AXI_WDATA(byte_index*8+7 downto byte_index*8);
              end if;
            end loop;
          when b"10" =>
            for byte_index in 0 to (C_S_AXI_DATA_WIDTH/8-1) loop
              if ( S_AXI_WSTRB(byte_index) = '1' ) then
```

Figure 4.7: Slave reg read

4.4 Serial Peripheral Interface: DAC

SPI stands for serial peripheral interface and is widely used for interfacing I/O devices. It is a synchronous and Full-duplex protocol. The multiple devices can be connected in an SPI bus, but there can be only one Mater & one slave at a given communication time. The SPI works on the master-slave concept. The device which initiates the data transfer is called the Master, whereas the device with which the master communicates is called a slave. One master can be connected with multiple devices via SPI, but it can communicate with only one slave at a given time.

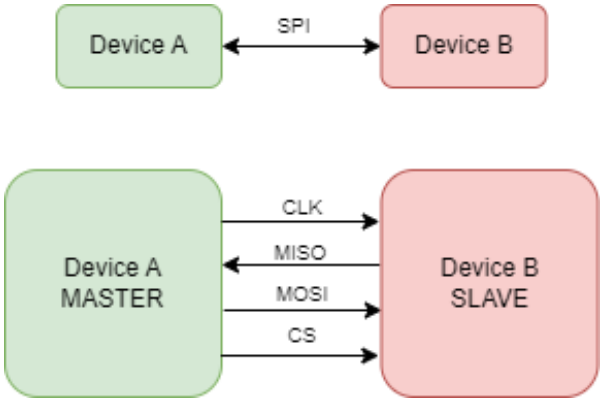


Figure 4.8: SPI protocol

The SPI bus contains 4 lines namely as:

1. Clock: The master provides the clock for communication and it is synchronous with MISO & MOSI lines.

2. Master Out Slave In (MOSI): The master writes the data on the MOSI line with respect to the clock and slaves reads the data from the MOSI line with respect to the clock

3. Master In Slave Out (MOSI); The salve writes the data on the MISO line with respect to the clock and the master reads the data from the MISO line with respect to the clock

4. Chip Select: This line is controlled by the master to select one device (among multiple connected devices) for communication.

The SPI has 4 modes of communication. The mode of communication depends on the clock polarity (CPOL) clock phase (CPHA). **The data should be stable and must not change, during the sampling by the slave**.

SPI Mode	CPOL	CPHA	Clock Polarity in Idle State	Data written at	Data sampled on
0	0	0	Logic Low	Falling Edge	Rising Edge
1	0	1	Logic Low	Rising Edge	Falling Edge
2	1	0	Logic High	Falling Edge	Rising Edge
3	1	1	Logic High	Rising Edge	Falling Edge

Figure 4.9: SPI Modes

4.4.1 SPI State Machine

A state machine has been designed to carry out all the functionalities required and it consists of total 7 states. The state machine is similar to what has been developed in the VHDl lab. few modification have been made to utilize it with DAC IC.

- SPI Mode : 1 (CPOL =0 and CPH =1)
- Base Clock : 100 Mhz item SPI State machine clock : 4 Mhz
- SPI Clock : 2 Mhz
- Date Length : 24 Bytes

1. The actions within the states are sequential and marked as red.
2. The actions during the state transition and conditions for state transitions are combinational and marked as green and blue.
3. The data length(USPI_SIZE) is defined as generic to design the modular SPI master, which can be used for transmitting any no. of data.

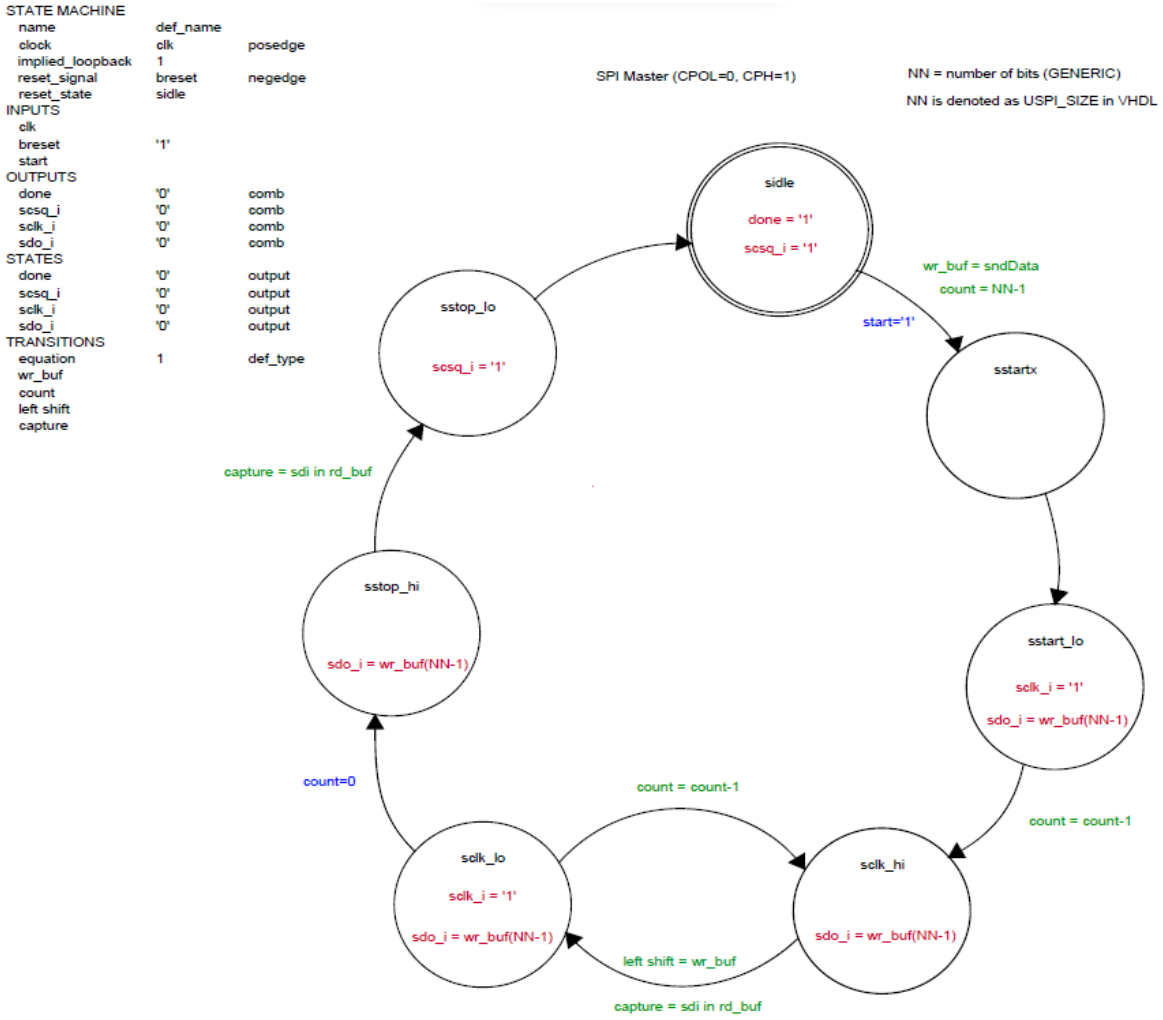


Figure 4.10: SPI Modes

State Name	sidle
Description	initial state
Action During the state	done = '1'
	SET chip select pin
Next State Transition Condition	IF start is SET
Action During Next State Transition	Load data into write buffer
	Assign data length to a variable counter
Next State	sstartx

Figure 4.11: SPI state 1

State Name	sstartx
Description	To start the SPI communication
Action During the state	None
Next State Transition Condition	None
Action During Next State Transition	None
Next State	sstart_lo

Figure 4.12: SPI state 2

State Name	sstart_lo
Description	To initiate the data transfer
Action During the state	SET SPI clock
Next State Transition Condition	write data(1 byte) in serial data output(SDO)
Action During Next State Transition	None
Next State	Decrement the counter variable by 1 count

Figure 4.13: SPI state 3

State Name	sclk_hi
Description	SPI clock high
Action During the state	write data(1 byte) in serial data output(SDO)
Next State Transition Condition	None
Action During Next State Transition	Left shift the data in write buffer
Next State	sclk_low

Figure 4.14: SPI state 4

State Name	sclk_low
Descriptions	SPI clock low
Action During the state	SET SPI clock
	write data(1 byte) in serial data output(SDO)
Next State Transition Condition	None
sclk_ni	None
sstop_hi	IF counter is zero
Action During Next State Transition	Decrement the counter variable by 1 count
Next State	sclk_hi
	sstop_hi

Figure 4.15: SPI state 5

State Name	sstop_hi
Descriptions	To Transmit the last byte
Action During the state	Write data(1 byte) in serial data output(SDO)
Next State Transition Condition	None
Action During Next State Transition	store data from serial data input (SDI) pin
Next State	sstop_lo

Figure 4.16: SPI state 6

State Name	sstop_lo
Descriptions	To stop the SPI communication
Action During the state	SET SPI clock
Next State Transition Condition	None
Action During Next State Transition	None
Next State	sidle

Figure 4.17: SPI state 7

4.4.2 SPI Algoritm

Two separate processes is included in the design

1. PROCESS A (sseq_proc): This process is to handle the sequential tasks. The sequential tasks are the actions during the transitions between the states. Such tasks are highlighted in green colour in the tables describing each state functionality.

2. PROCESS B (scmb_proc): This process is designed for combinational tasks.

   The combinational tasks include actions in the state and conditions for state transitions. Such tasks are highlighted in red and blue in colour in the tables describing each state functionality.

3. PROCESS C (clk_4MHz): This process is designed 2 generate a 4 Mhz clock for the SPI state machine from the 100 Mhz AXI clock.

4.4.2.1 PROCESS A: sseq_pro : sensitivity - bclk

This process is executed on every clock. The main function of the process is to determine the actions during the transitions between the states.
The algorithms used in the sequential process :

1. Technique to send the data: byte by byte from send data buffer.
   The data which is to be sent by SPI is read from sndData input port; stored in the write buffer and the count variable is stored with the total no. of bytes to be sent. Two actions take place after sending a byte.

   a) Counter is decremented

   b) Write buffer data is shifted left

   Algorithm :
   Step 0 : Read data from the input port and store it in a local write buffer
   Step 1: Length of data is stored in local variable count
   Step 2: Send a byte (MSB first) via sdo
   Step 3: Decrement the variable count by 1
   Step 4: Rotate Left the data in the write buffer
   Step 5: Go to Step 2 until the counter becomes zero

2. Technique to receive the data: byte by byte from SDI.
   The received byte is stored from sdi and stored in LSB position and then left

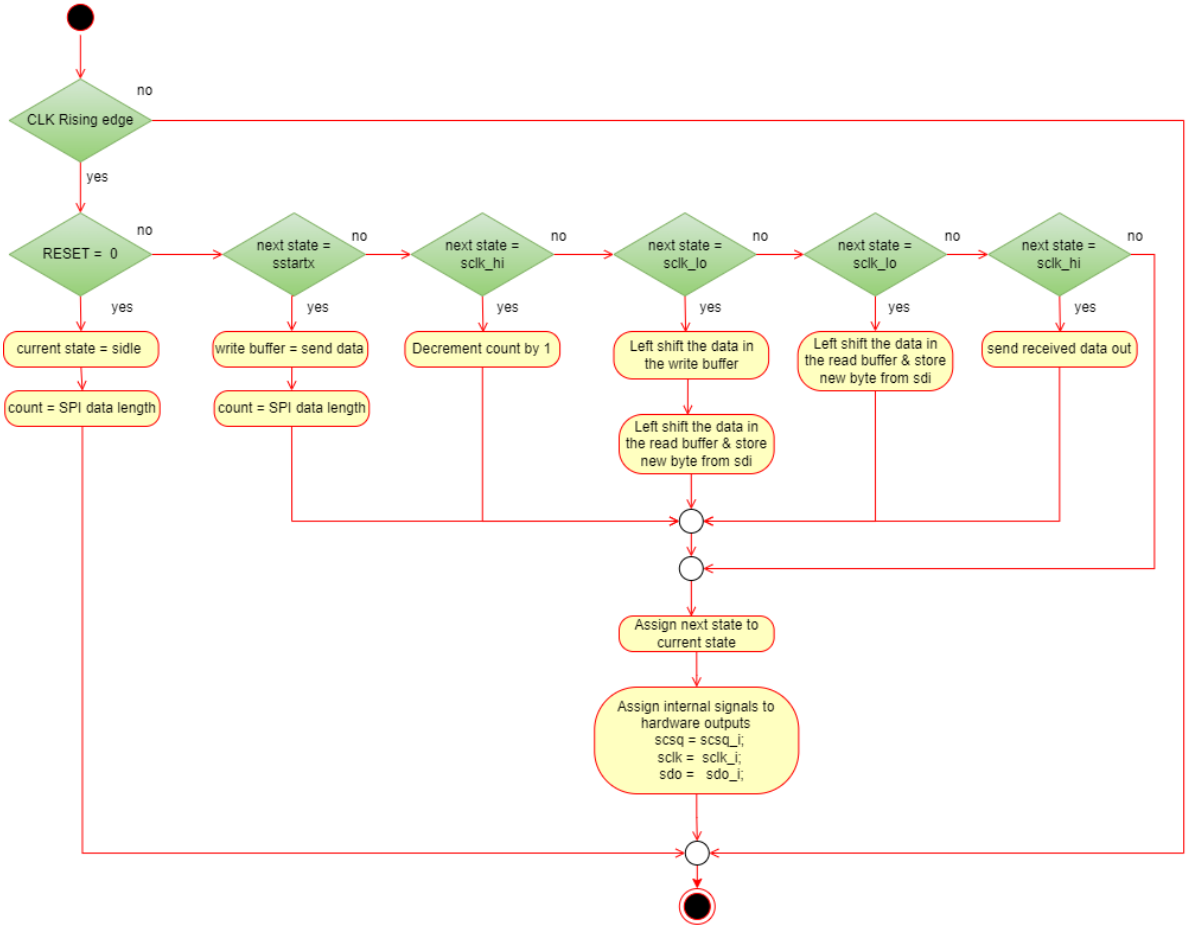


Figure 4.18: PROCESS A

shifted every time a new byte is received.

Algorithm:

Step 0: Read data from sdi and store in LSB of local receive data buffer & left shift

bit0 - bit22 to bit1 - bit23

Step 1: Repeat Step 0 until full 24-byte data is received.

Step 2: After receiving all 24 bytes, send the entire data to the received data output port.

4.4.2.2 PROCESS B: scmb_proc : sensitivity – state and dstart

The main function of the process is to determine the next states, based on the current state of the system. The tasks included in this process are combinational.

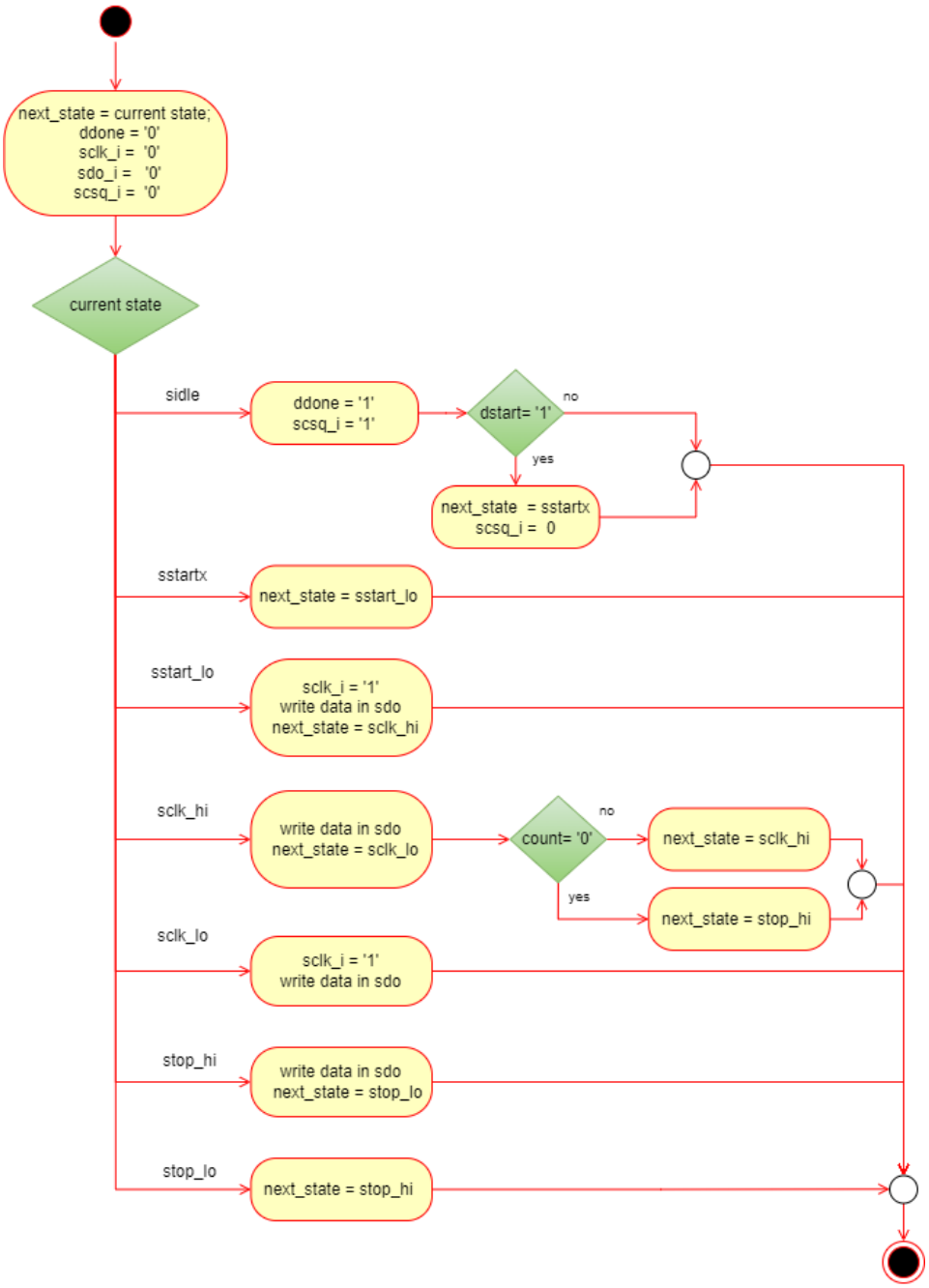


Figure 4.19: PROCESS B

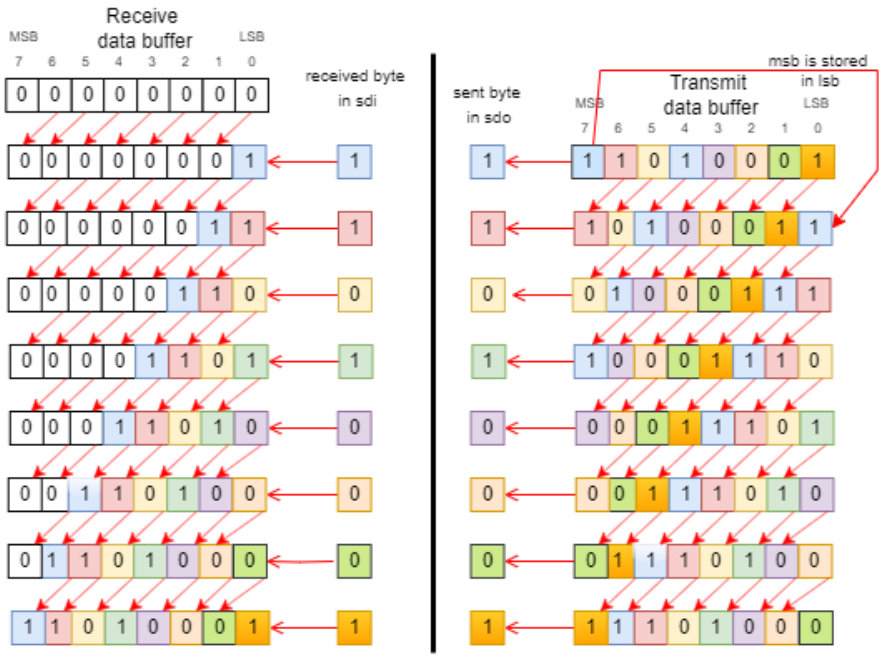


Figure 4.20: Transmission & Reception of serial data(8 Bit) byte by byte

4.4.2.3 PROCESS C: clk_4MHz : sensitivity – AXI_CLK

```
clk_2MHz : PROCESS(AXI_CLK)
BEGIN
  IF rising_edge(AXI_CLK)  THEN
    IF count_clk >= 25 THEN
        IF b_clk = '1' THEN
          b_clk <= '0';
        ELSE
          b_clk <= '1';
        END IF;
        count_clk <= 0;
    ELSE
        count_clk <= count_clk+1;
    END IF;
  END IF;
END PROCESS clk_2MHz;
```

Figure 4.21: VHDL code for SPI state machine clock.

4.4.3 Simulation

The source code is simulated on the same software – VIVADO. The simulation allows to provide the inputs to the Unit Under Test (Source code) and produce the outputs according to the Unit Under Test design. The simulation code is also written in VHDL.

- bclk is the clk to the SPI state machine clock
- SDI (sndData) 0x380001 to set the internal reference of DAC 1.25v
- SDO (rcvData) is inverted sdi. Therefore 0x380001 inverted to 0xc7fffe.

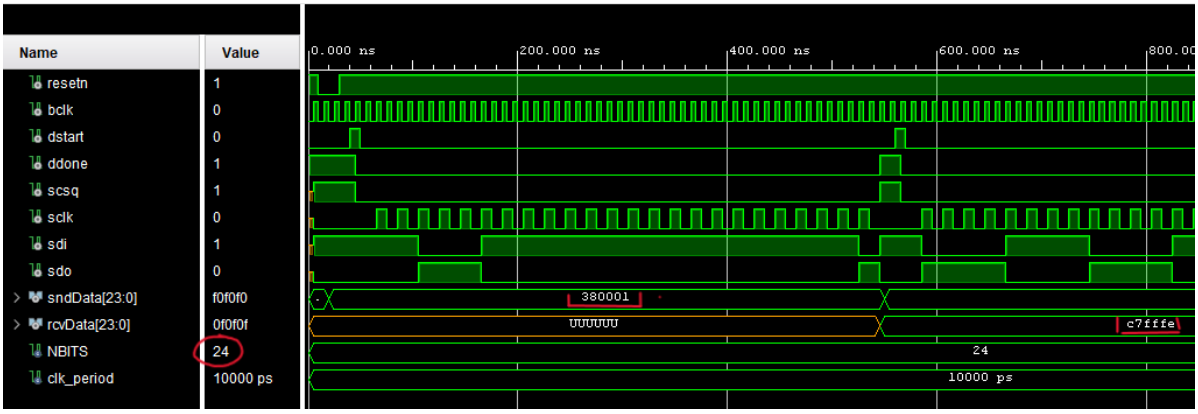


Figure 4.22: Simulation Result

4.5 Hazard Analysis

Hazards refer to unwanted changes in output caused by changes in input signals, which can result in false outputs for a signal that should be stable. Two types of hazards are static-0 and static-1 hazards, which respectively represent false outputs for signals that should be at a stable 0 or 1 level. Timing hazards arise due to disparities in propagation delays in combinational logic circuits. It can be solved by adding extra min terms as per the K-map.

- A static-0 Hazard creates a '1', a1though it shou1d stay '0'.
- A static-1 Hazard creates a '0', a1though it shou1d stay '1'.

Hazards do not occur in digital outputs that come out of flip-flops. Flip flops are edge-triggered devices that latch the input signal at the clock edge and hold the output stable until the next clock edge. Since flip flops have a single clock input, there is no possibility of propagation delays or glitches that might cause hazards. Therefore, any digital output that comes out of a flip-flop is free from hazards, making it a reliable and safe output signal.

The design in our project has addressed the issue of hazards in SPI communication by ensuring that the SPI lines are connected to flip-flops. This arrangement eliminates the possibility of hazards. A schematic showing this implementation can be viewed below.

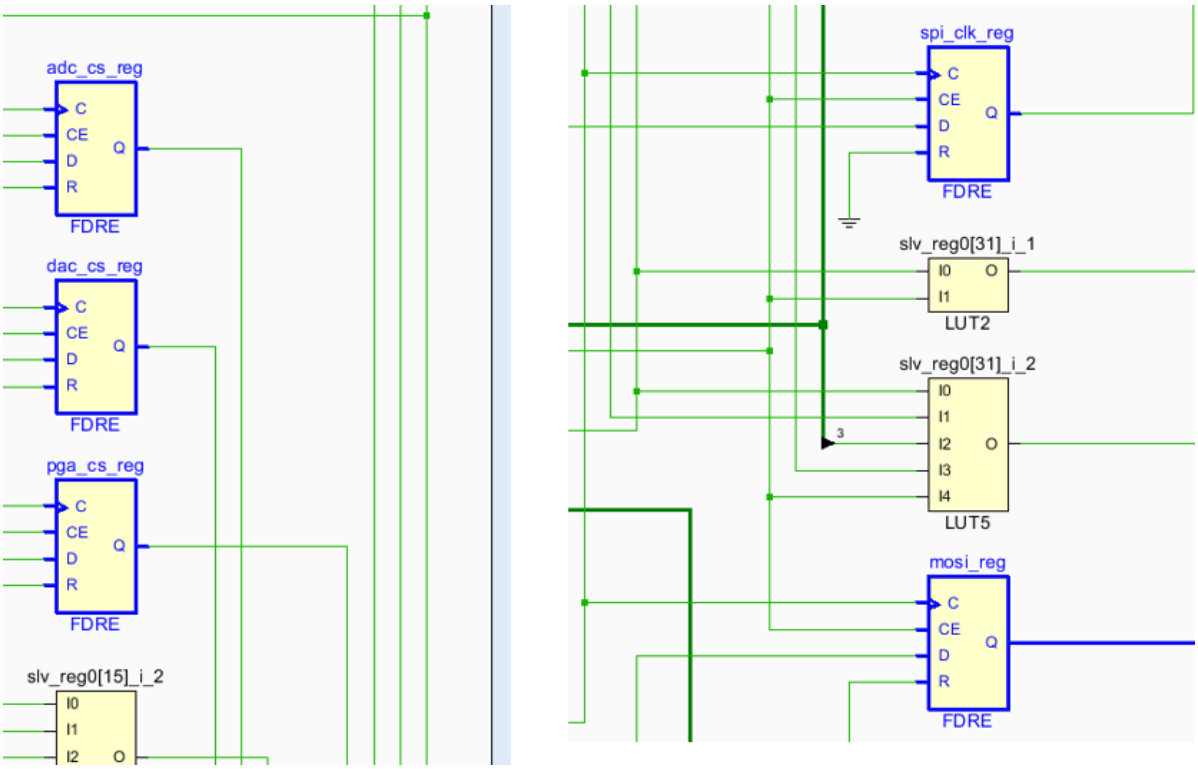


Figure 4.23: Part of schematic

4.6 Implementation Report

The section describes the utilization of PL resources like LUT, Flip flop, LUTRAM, IO and BUFG.

Utilization with initial version

Utilization | Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	409	53200	0.77
LUTRAM	60	17400	0.34
FF	591	106400	0.56
IO	24	125	19.20
BUFG	1	32	3.13

Utilization with final version

Utilization | Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	536	53200	1.01
LUTRAM	60	17400	0.34
FF	743	106400	0.70
IO	15	125	12.00
BUFG	1	32	3.13

Figure 4.24: Utilization of PL resources

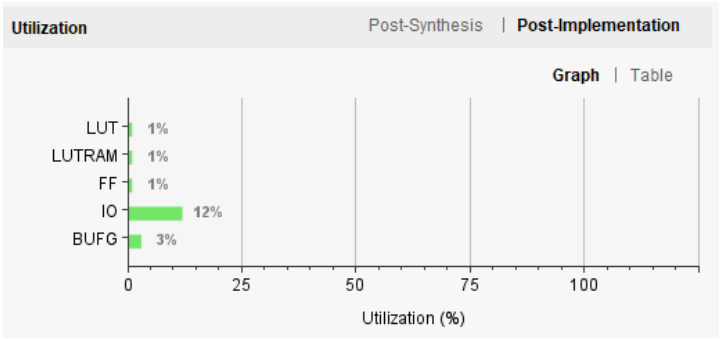


Figure 4.25: Graphical Representation of final version

5 Filter Design and Implementation (Abhinit - 39405)

5.1 Introduction

The signal path of pulse oximeters contains one or more filters, which are used to filter out the noise and unwanted signal components from the received signal. These filters can be analog, digital, or both. The narrower the bandwidth of the filter, the lower the noise content of the filtered signal. Noise comes from external sources and the photodiode of the pulse oximeter also senses environmental light, which flickers in the case of fluorescent lighting.

Implementing a signal filter with a narrow bandwidth appears to be an obvious solution for separating the pulse signal from other signal components, such as noise. However, by lowering the corner frequency of a low-pass filter (LPF) in the signal path, we also decrease the information content of the useful signal. The narrow bandwidth of the filter causes the dicrotic notch to disappear, and the filtered signal resembles a sinusoidal wave rather than a pulse wave. Cutting off the harmonics of the measured pulse signal significantly distorts the shape of the signal and may affect the R ratio, and consequently, the oxygen saturation measurement.

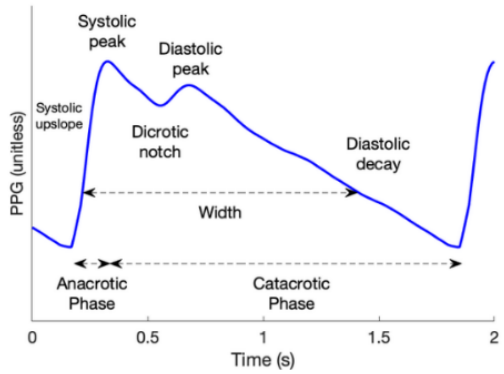


Figure 5.1: Photoplethysmography Signal (PPG)

5.2 Filter Types

When dealing with noisy data, there are various ways of going about filtering that data. The two overarching categories in which one can filter data are analog filtering and digital filtering. Analog filtering involves physical hardware that alters analog signals before they are passed off to other components to be processed. Digital filtering involves passing analog data to a processor that then runs code to digitally filter the data.

5.2.1 Analog Filters

Analog filters have the main advantage of speed. Filtering with hardware means that the signal coming out of the physical filter is the final signal. Analog filters also provide a greater dynamic range for frequency. It is relatively easy to design a frequency filtering circuit with an operational amplifier (op amp) that can handle signals that have frequencies between 0.01 Hz and 100 kHz Analog filters require physical space, so they must be used sparingly if space is an issue. As with any physical hardware, if there is an issue with its design, it is much harder to fix once a product is deployed, as hardware cannot be altered over the air.

5.2.2 Digital Filters

The most apparent advantage of digital filtering is that digital filters require less hardware, as they are done on a processor. This makes them very versatile and applicable in any system with a processor. In addition to lowering the cost, extra hardware has the added disadvantages of being affected by external factors such as temperature, humidity, and general wear and tear. Digital filters are software programmable, which makes them easy to bring up and test. Being able to quickly program and prototype digital filters also contributes to their versatility.

The standard disadvantage of a digital filter is that digital filters are significantly slower than analog filters. Digital filters introduce additional latency into a system, as the analog data that comes out of the hardware must be processed on a computer before it is filtered as desired. It is also difficult to handle large frequency ranges with digital filters. The sampling rate to capture one cycle at 0.01 Hz must be extremely high (20 million points).

For the reasons stated above, we will use Digital Filtering for our project.

5.3 Digital Filtering

Digital filtering consists of convolving the input signal with filter coefficients to produce a filtered signal. The transfer function of a filter can be expressed in the z-domain as:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{M} b_i.z^-i}{1 + \sum_{k=0}^{N} a_k.z^-k} \tag{5.1}$$

where b(i) and a(k) are the coefficients of the numerator and denominator, respectively. The coefficients are determined in the design process according to the type of filter and the cut-off frequencies required. This transfer function can also be expressed as a difference equation, which can be easily applied to the original signal in the time domain, in order to obtain the filtered signal, y. The difference equation can be expressed as:

$$y(n) = \sum_{i=0}^{M} b_i.x(n-i) - [1 + \sum_{k=1}^{N} a_k.y(n-k)] \tag{5.2}$$

The design process of a digital filter consists simply on finding the b(i) and a(k)coefficients that give the desired response of the filter.

5.4 Steps for designing a Digital Filter

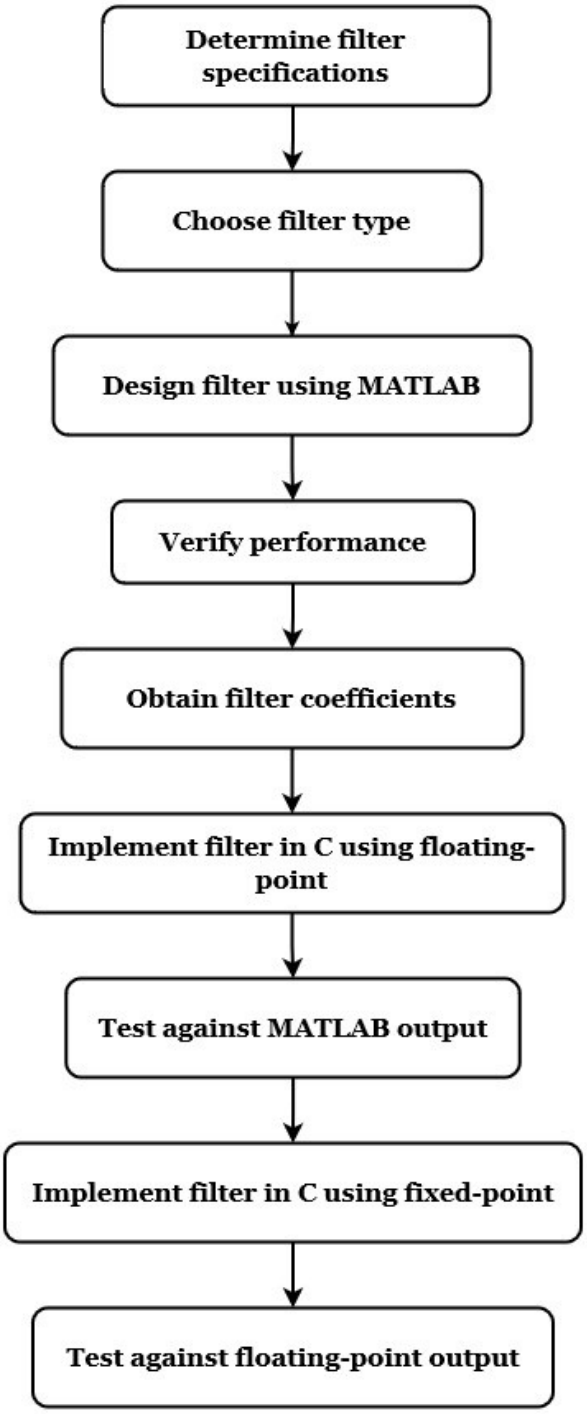


Figure 5.2: Flowchart for designing a Digital Filter

5.4.1 Determine Filter Specifications

Human heart rate can vary widely, ranging from a very low 30 beats per minute (corresponding to a frequency of 0.5Hz) to an extremely high 200 beats per minute (corresponding to a frequency of 3.33Hz). It is important to note that the normal respiration rate for a person at rest typically falls within the range of 12 to 16 breaths per minute, which corresponds to a frequency of 0.26Hz. When designing a filter for physiological

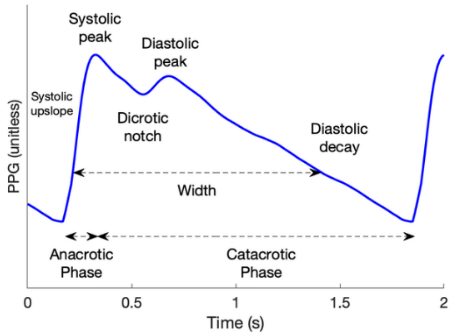


Figure 5.3: Photoplethysmography Signal (PPG)

signals, it is crucial to ensure that it can effectively attenuate the breathing frequency of 0.26Hz while still allowing the lower heart rate of 0.5Hz to pass through.
Most of the frequency content of the PPG signal is below 15 Hz, but a lower low-pass cut-off frequency helps make individual pulse waves more recognisable but distorts their shape. A lower low pass filter cut-off frequency cound possibly eliminate the Diastolic peak and would result in errorneous calculations further on.

Subject and PPG probe movements are likely to result in significant alterations to the PPG waveform, with resultant timing, amplitude and/or shape measures affected across a number of heart beats. Furthermore, sudden breathing changes - a cough, even talking - can also impact on the PPG signal. The goal is to try to reduce noise in the first place by collecting signals of high quality and to a well-considered protocol whereby a subject is at rest and their body movements minimised. So keeping all the above mentioned points in check, we decided to design filters with the following specifications :

Filter Type	Cut-off Frquency
Highpass Filter	0.5 Hz
Lowpass Filter	9Hz

5.4.2 Choose Filter Type

Because of the advantages that Digital Filters have over its counterparts, we decided to move forward with designing a Digital Filter.
There are two main families of digital filters which differ according to their transfer function:

- **Finite Impulse Response** (FIR) filters and
- **Infinite Impulse Response** (IIR) filters.

As the terminology suggests, these classifications refer to the filter's impulse response. By varying the weight of the coefficients and the number of filter taps, virtually any frequency response characteristic can be realized with an FIR filter. FIR filters can achieve performance levels which are not possible with analog filter techniques (such as perfect linear phase response). However, high performance FIR filters generally require a large number of multiply-accumulates and therefore require fast and efficient DSPs.
On the other hand, IIR filters tend to mimic the performance of traditional analog filters and make use of feedback. Therefore their impulse response extends over an infinite period of time. Because of feedback, IIR filters can be implemented with fewer coefficients than for an FIR filter.
The equations for both an IIR and FIR filter are shown. The input to the filter is x(n), and the output of the filter is y(n).

$$\textbf{FIR Filter Equation} : y(n) = \sum_{i=0}^{M} b_i.x(n-i). \tag{5.3}$$

$$\textbf{IIR Filter Equation} : y(n) = \sum_{i=0}^{M} b_i.x(n-i) - \sum_{k=0}^{N} a_k.y(n-k). \tag{5.4}$$

The mathematical difference between the IIR and FIR implementation is that the IIR filter uses some of the filter output as input. This makes the IIR filter a 'recursive' function.

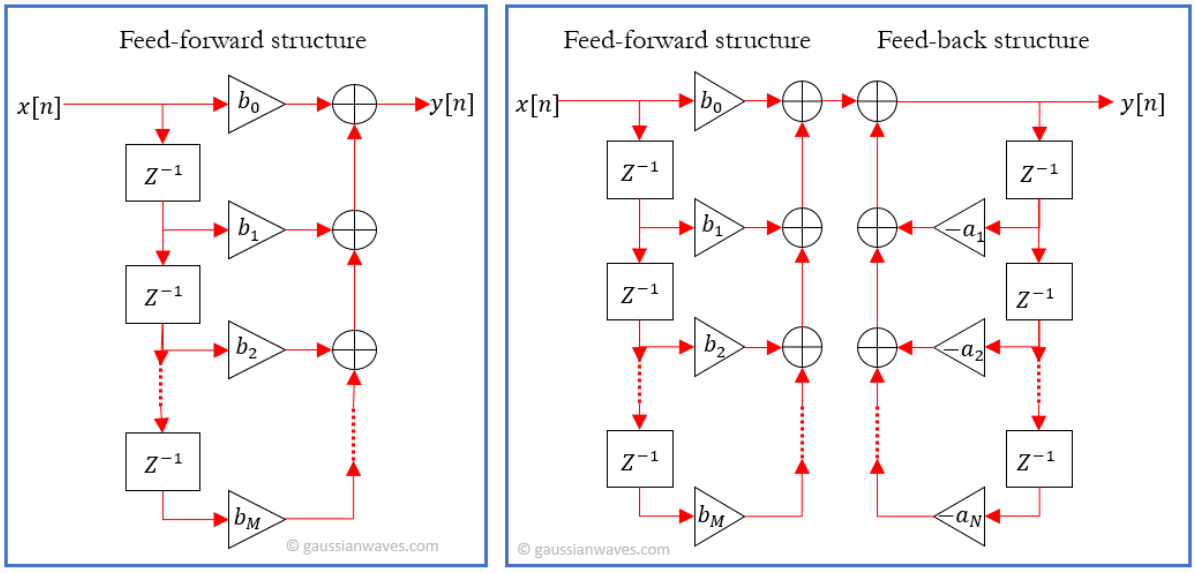


Figure 5.4: (a) FIR Filter Structure & (b) IIR Filter Structure

5.4.3 Comparison of FIR & IIR Filter based on their characteristics

5.4.3.1 Frequency Response & Order of Filter

An IIR filter has an advantage that for a similar filter roll off as a FIR, a lower order or lower number of terms can be used. This means that less computations are needed to achieve the same result, making the IIR faster computationally. However, an IIR has nonlinear phase and stability issues.

A filter with more terms (i.e a higher order) has a sharper transition between the frequencies being passed and the frequencies being stopped. Making a filter sharper is done by increasing the order. This requires more calculations, and has an impact on the time delay introduced by the filter.

The sharpness of FIR and IIR filters is very different for the same order as shown in the figure below. Because of the recursive nature of an IIR filter, where part of the filter output is used as input, it achieves a sharper roll off with the same order filter. This makes IIR filters faster from a computational point of view than FIR filters. If a filter has to be implemented in real-time application, it is typically done with an IIR filter.

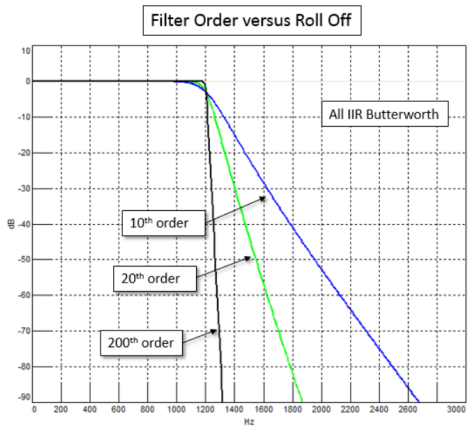


Figure 5.5: Filter Order & Roll off

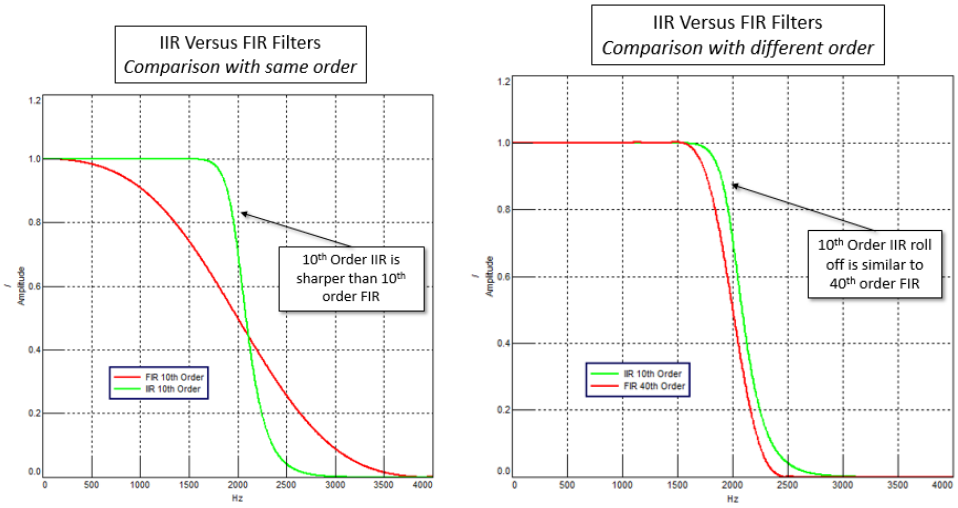


Figure 5.6: Comparison of FIR v/s IIR Filter Order Response

5.4.3.2 Time Delay

The time delay of an IIR filter is determined by the phase response of the filter, which is a function of the filter's transfer function.

$$GroupDelay = -\frac{d\theta(\omega)}{d\omega} \tag{5.5}$$

Where, $\theta(\omega)$ is the phase response of the filter.
The significance of the group delay is that it is a measure of average time delay taken by the input signal to pass through the filter.
At the cutoff frequency of an IIR filter, the filter's phase response undergoes a shift of 90 degrees. This means that the output signal of the filter is delayed by a maximum amount of time, which is equal to one-quarter of the period of the input signal at the

cutoff frequency.
This delay occurs because the filter is designed to attenuate or pass frequencies above or below the cutoff frequency, and the attenuation or amplification of these frequencies causes a phase shift in the output signal. At the cutoff frequency, the phase shift is maximum, resulting in the maximum time delay.

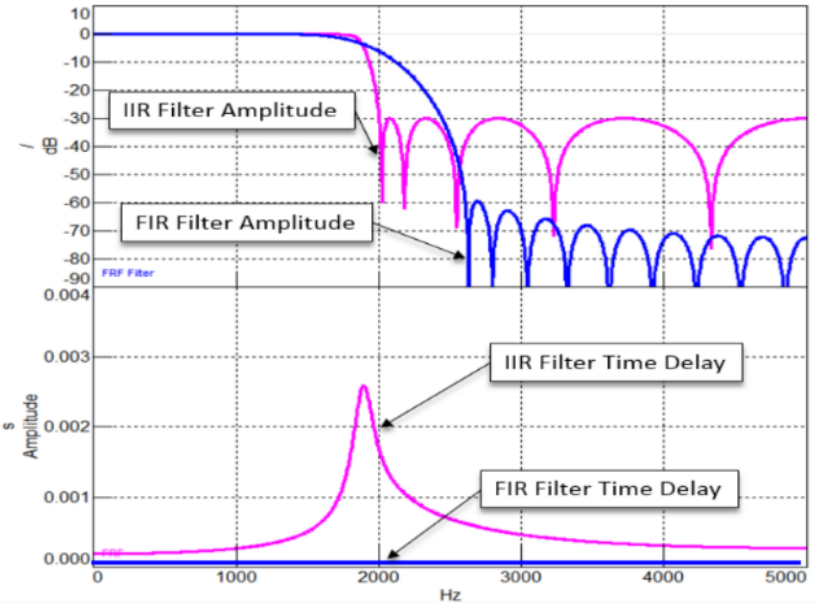


Figure 5.7: Time Delay comparison

5.4.3.3 Stability

For a causal LTI digital filter will be BIBO (Bounded Input Bounded Output) stable, if and only if the impulse response h[n] is absolutely summable.

$$\sum_{n=-\infty}^{n=-\infty} |h[n]| < \infty \tag{5.6}$$

Impulse response of FIR filters are always bounded and hence they are inherently stable. On the other hand, an IIR filter may become unstable if not designed properly.
Example : Consider an IIR filter implemented using a floating point processor that has enough accuracy to represent all the coefficients in the transfer function below :

$$H_1(z) = \frac{1}{1 - 1.845.z^{-}1 + 0.850586.z^{-}2} \tag{5.7}$$

The corresponding impulse response h1[n] is plotted below (Impulse response-h1(n)). The plot shows that the impulse response decays rapidly to zero as 'n' increases. For

this case, the sum in equation 7.7 will be finite. Hence this IIR filter is stable. Suppose, if we were to implement the same filter in a fixed point processor and we are forced to round-off the co-efficients to 2 digits after the decimal point, the same transfer function looks like this

$$H_2(z) = \frac{1}{1 - 1.85.z^{-}1 + 0.85.z^{-}2} \tag{5.8}$$

The corresponding impulse response h2[n] plotted in (Impulse response-h2(n)) shows that the impulse response increases rapidly towards a constant value as 'n' increases. For this case, the sum in equation (7.8) will tend to infinity. Hence the implemented IIR filter is unstable.

Therefore, it is imperative that an IIR filter implementation need to be tested for stability. To analyze the stability of the filter, the infinite sum in equation (7.8) need to be computed and it is often difficult to compute this sum. Analysis of pole-zero plot is an alternate solution for this problem. To have a stable causal filter, the poles of the transfer function should lie completely strictly inside the unit circle on the pole-zero plot.

If we plot the pole-zero plot for the above given transfer functions H1[z] & H2[z] , it shows that for the transfer function H1[z], all the poles lie within the unit circle (the region of stability) and hence it is a stable IIR filter. On the other hand, for the transfer function H2[z], one poles lie exactly on the unit circle (i.e, it is just out of the region of stability) and hence it is an unstable IIR filter.

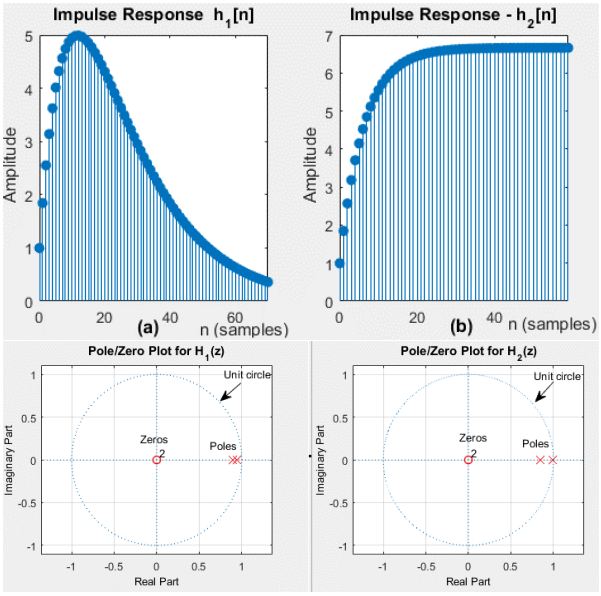


Figure 5.8: Instability due to Coefficient Quantization Error

5.4.3.4 Phase Response

Linear phase digital filters allow all the frequency components of an input signal to pass through the filter with the same delay, which means that the group delay through the filter is a constant value independent of the frequency. Linear phase filters are useful in filtering applications in which you want to minimize signal distortion and spreading over time.

Nonlinear phase response, or dispersion, can be harmless in audio and other applications in which mild phase distortion is often imperceptible to humans. However, phase distortion can be harmful in some applications. For example, in digital communications applications, signal spreading caused by phase distortion can cause interference between time concentrated information symbols. Digital filters with linear phase have the ad-

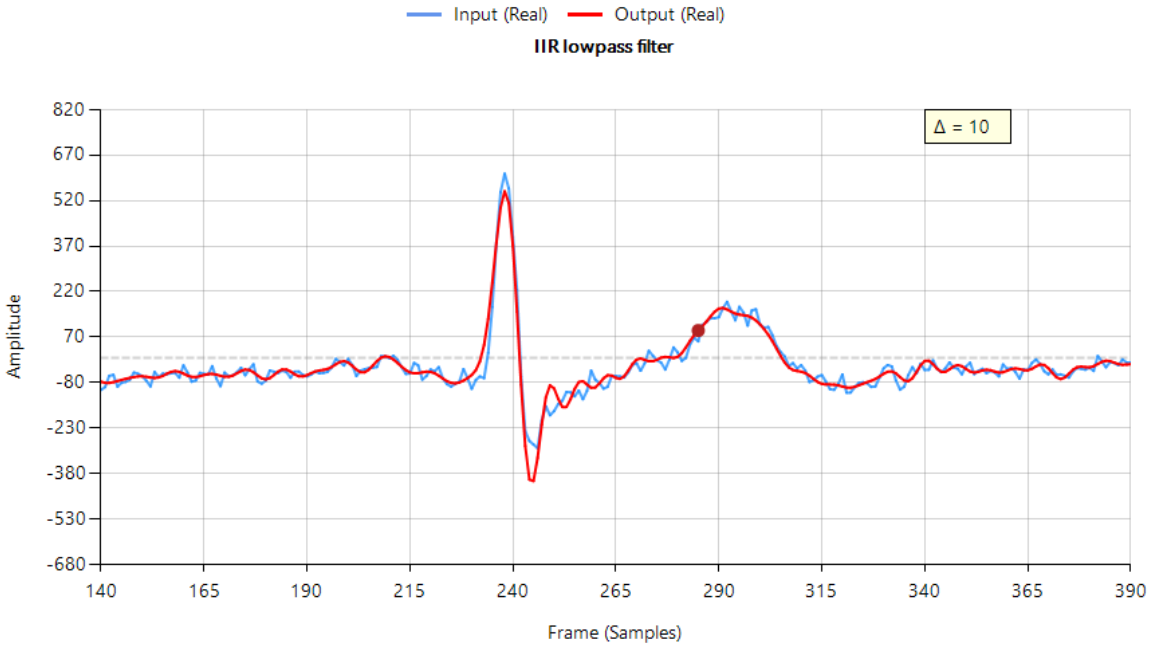


Figure 5.9: Phase Response

vantage of delaying all frequency components by the same amount, i.e. they preserve the input signal's phase relationships. This preservation of phase means that the filtered signal retains the shape of the original input signal. This characteristic is essential for ECG biomedical waveform analysis, as any artefacts introduced by the filter may be misinterpreted as heart anomalies.

5.4.4 FIR v/s IIR Filter

Parameter	Finite Impluse Response	Infinite Impulse Response
Nature	Non-recursive	Recursive
Computational Efficiency	Less	Comparatively more
Usage	Difficult	Quite easy
Feedback	Absent	Present
Stability	More	Less
Requirement to generate current output	Present and past samples of input.	Present and past samples of input along with past output.
Delay offered	High	Comparatively lower
Transfer function	Only zeros are present.	Both poles and zeros are present.
Memory requirement	More	Less
Sensitivity	Less	Comparatively more
Controllability	Easy	Quite Difficult

Figure 5.10: Summary of FIR v/s IIR Comparison

After thorough consideration of all of the mentioned factors, we chose to construct a Lowpass & Highpass IIR Filter.

5.5 Design Filter using MATLAB

5.5.1 Raw Sample Data

To start with, a sample signal was provided with a :

- Respiratory frequency = 0.11Hz
- DC offset = 1200
- Heartbeat frequency = 80bpm (1.33Hz)
- AC Component of 3 different magnitudes
- Added noise constant of 0.01
- Sampling frequency = 43Hz and,
- Total number of samples = 10000

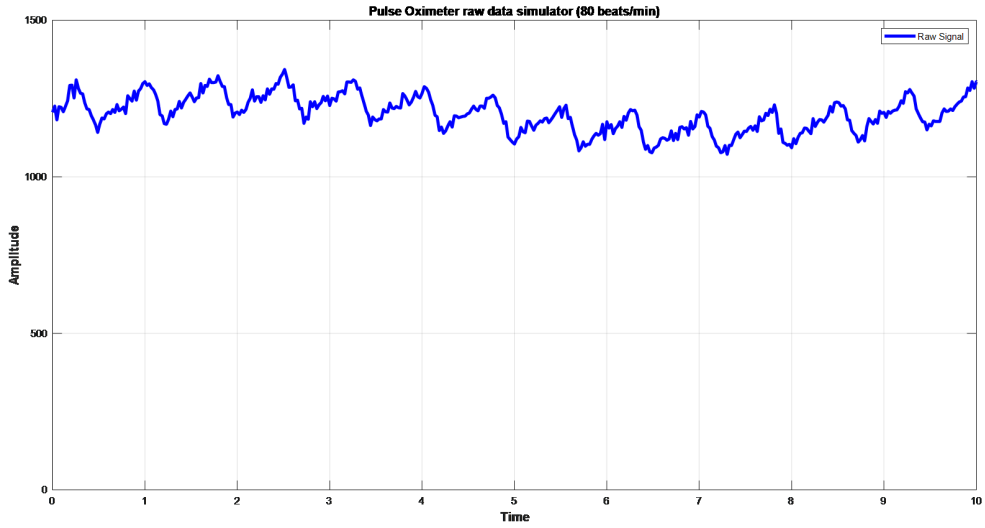


Figure 5.11: Raw PPG Signal

5.5.2 Lowpass Filter

To remove noise from the raw signal, a Low pass IIR Butterworth filter with cut-off frequency of 9Hz was designed.

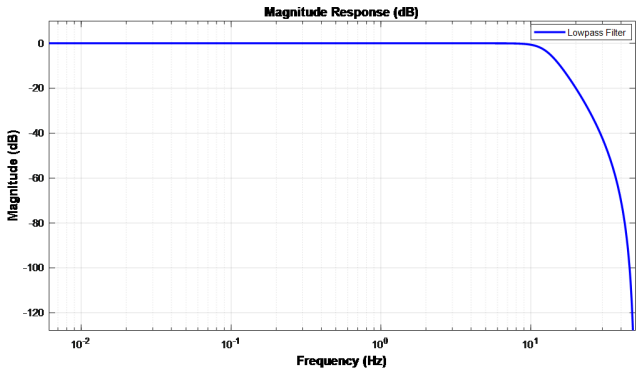


Figure 5.12: Frequency Response of Lowpass Filter with cut-off frequency = 9Hz

Lowpass Butterworth Filter specifications are as follows :

- Filter Structure : Direct Form II Transposed
- Numerator Length : 5
- Denominator Length : 5
- Stable : Yes
- Phase : Nonlinear

After applying LPF on Raw signal the final output looks like :

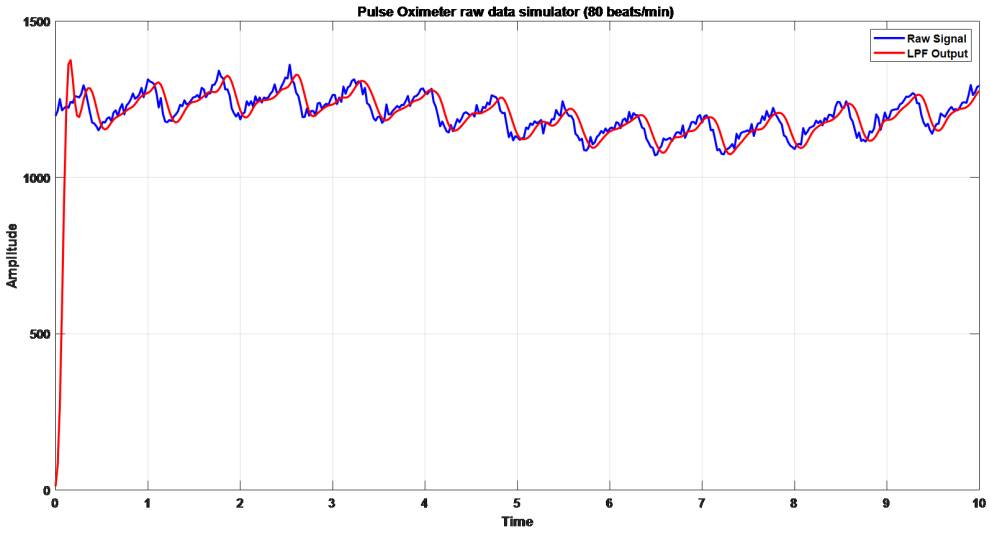


Figure 5.13: Lowpass Filter Output

5.5.3 Highpass Filter

After removing noise from the signal, the DC Offset caused by changes in other tissue components such as venous and capillary blood, bloodless tissue, etc and also the low frequency respiratory is to be attenuated. Hence, for this purpose a Highpass Butterworth IIR Filter with cutoff frequency of 0.5Hz was designed. Highpass Butterworth

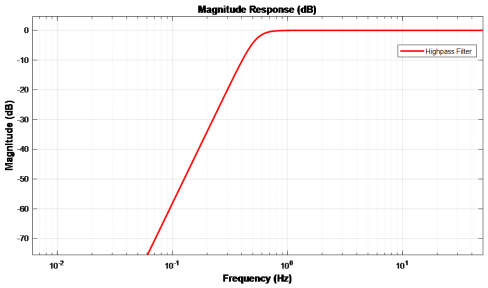


Figure 5.14: Frequency Response of Highpass Filter with cut-off frequency = 0.5Hz

Filter specifications are as follows :

- Filter Structure : Direct Form II Transposed
- Numerator Length : 5
- Denominator Length : 5
- Stable : Yes
- Phase : Nonlinear

After applying HPF on output of Lowpass Filter, the final output looks like :

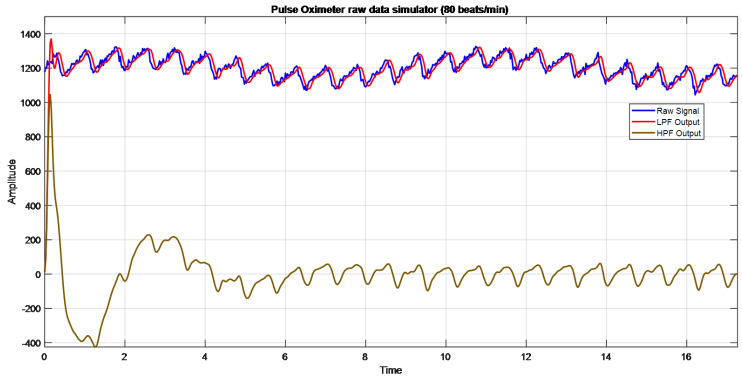


Figure 5.15: Highpass Filter Output

5.5.4 Filter Coefficients

After achieving satisfactory output from the designed filters, we obtained the filter coefficients and moved on to implementing the filter on C.

Lowpass Butterworth Filter Coefficients
Order of Filter : 4

	Numerator		Denominator
b0	0.003227021246802	a0	1
b1	0.012908084987208	a1	-2.548097716243580
b2	0.019362127480811	a2	2.609926241233029
b3	0.012908084987208	a3	-1.237146989281122
b4	0.003227021246802	a4	0.226950804240504

Highpass Butterworth Filter Coefficients
Order of Filter : 4

	Numerator		Denominator
b0	0.971224810242597	a0	1
b1	-3.884899240970387	a1	-3.941607386855387
b2	5.827348861455580	a2	5.826520971017184
b3	-3.884899240970387	a3	-3.828190973978209
b4	0.971224810242597	a4	0.943277632030769

Figure 5.16: Lowpass & Highpass Filter Coefficients

5.5.5 Filter Results

- Frequency Response

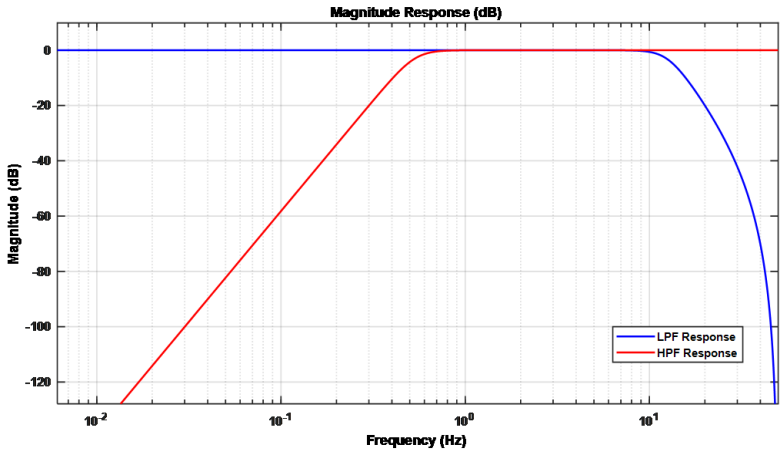


Figure 5.17: Frequency Response

- Pole-Zero Plot

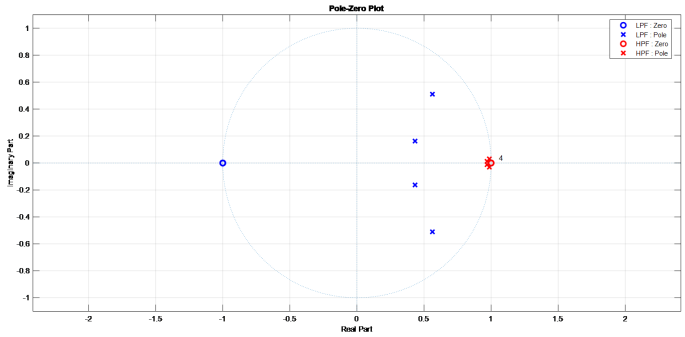


Figure 5.18: Pole-Zero Plot

- Phase Delay

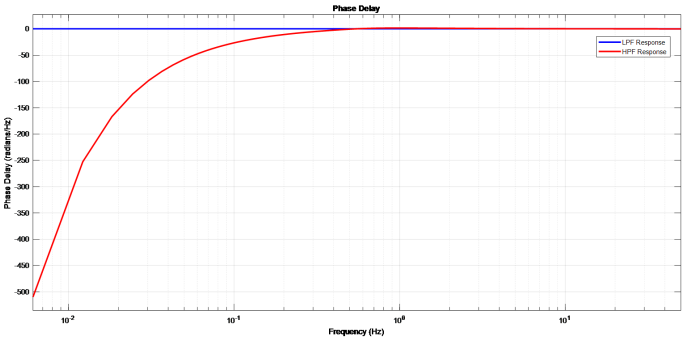


Figure 5.19: Phase Delay

- Group Delay

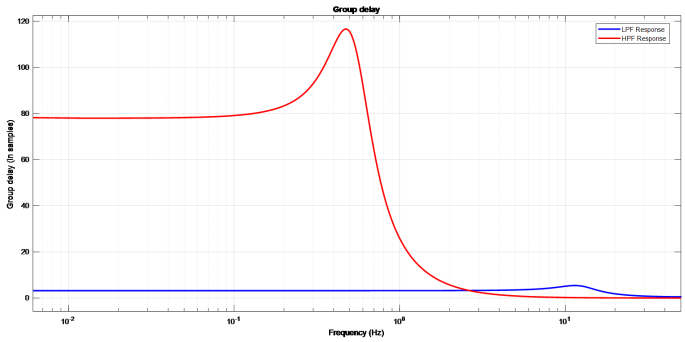


Figure 5.20: Group Delay

- Phase Response

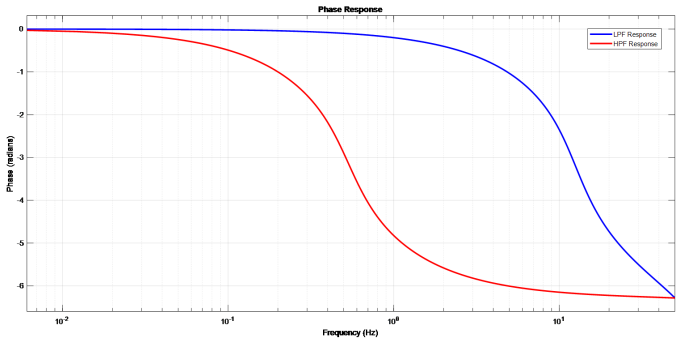
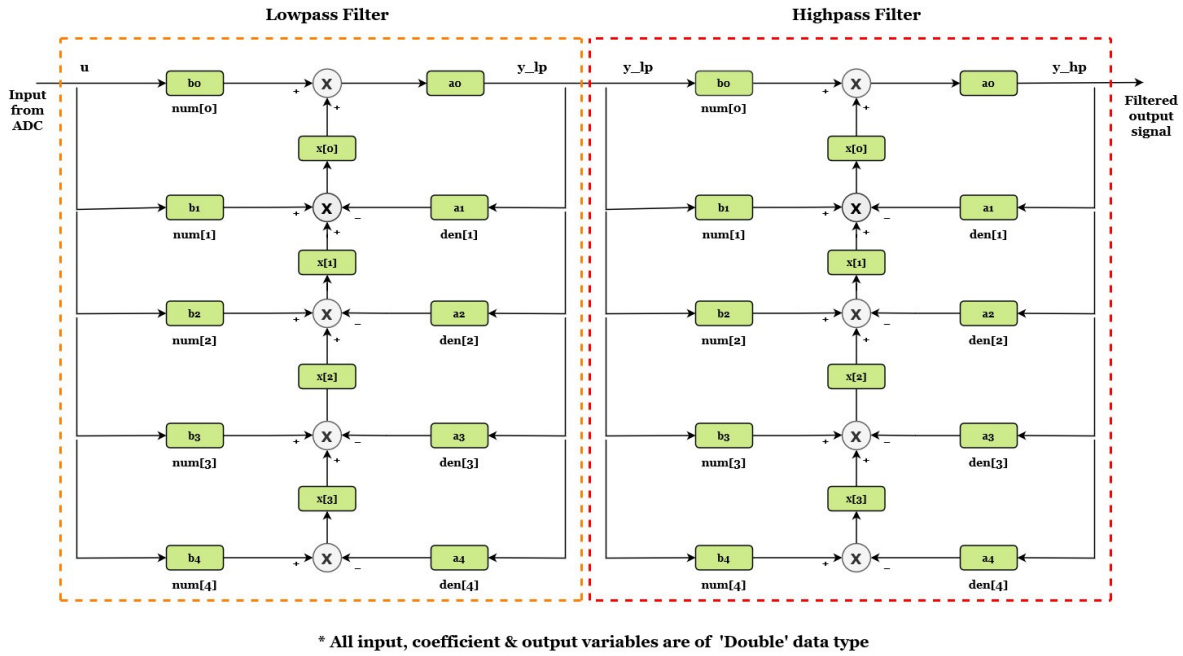


Figure 5.21: Phase Response

5.6 Implement Filter in C using Floating Point

After obtaining a satisfactory response on MATLAB from the filter on the raw sample data provided, the filter coefficients were carefully noted and a digital filter was implemented using the C programming language with floating point computation. The resulting output shown below was then compared to the filter response obtained from MATLAB to verify the accuracy of the output. This process involved rigorous testing and analysis to ensure that the C implementation of the filter was reliable.

5.6.1 Filter structure for Floating point computation



* All input, coefficient & output variables are of 'Double' data type

5.6.2 Output and Delay State Equations

```
y = filtobj -> num[0] *u + filtobj -> x[0] ;                    ..... (1)
for (k=1; k<n; k++)
{
   filtobj -> x[k-1] = filtobj->num[k] *u + filtobj->x[k] - filtobj->den[k] * y;        ..... (2)
}
filtobj -> x[n-1] = filtobj->num[n] *u  - filtobj->den[n] *y ;          ..... (3)
```

Equation 1 → Output State Equation

Equation 2 → Delay State Equation

Equation 3 → Delay State Equation for the last order

5.6.3 Filter Output

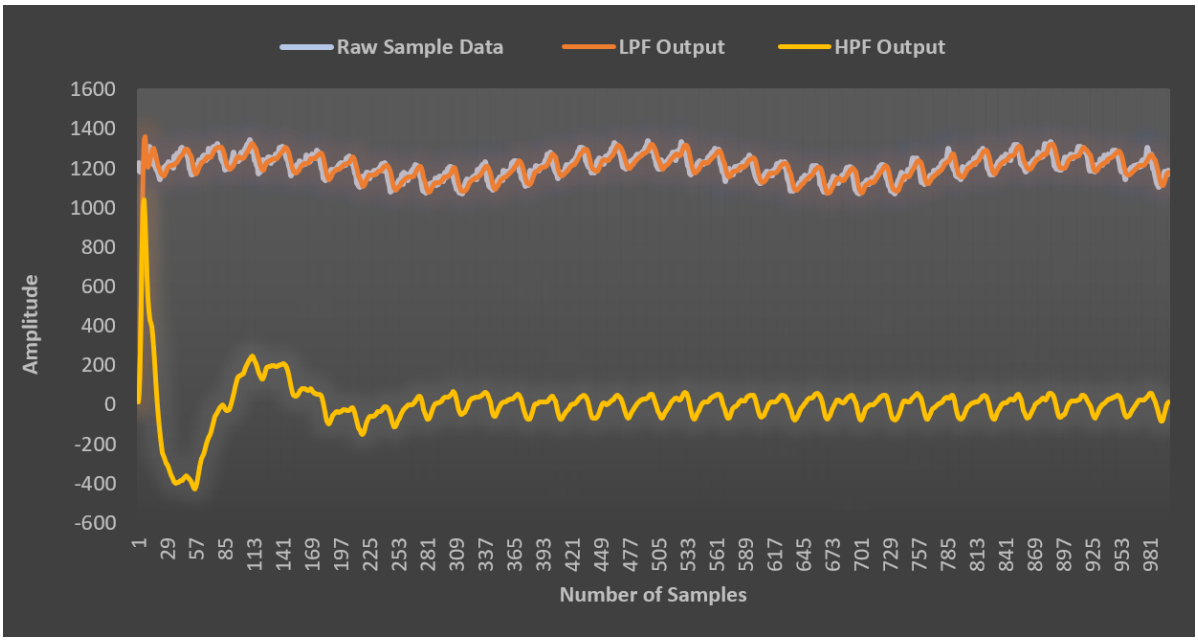


Figure 5.22: Filter Output using Floating point arithmetic

After successfully implementing the filter using floating point arithmetic in the C programming language, we proceeded to integrate it into the larger system to test its functionality. This involved connecting the filter code to the existing codebase and

ensuring that all inputs and outputs were properly interfacing with the filter. Through this integration process, we were able to verify that the filter was operating correctly and providing the expected results.

5.7 Implement Filter in C using Fixed Point Arithmetic

5.7.1 Introduction

There are several reasons why we chose to use fixed-point arithmetic over floating-point arithmetic. Here are some key points:

- **Efficiency** : Fixed-point arithmetic can be more efficient than floating-point arithmetic, particularly on embedded systems or devices with limited processing power. This is because fixed-point arithmetic can be implemented using integer arithmetic, which is typically faster than floating-point arithmetic, which requires more complex operations.
- **Memory usage** : Fixed-point arithmetic can require less memory than floating-point arithmetic, particularly if you only need to represent numbers with a limited range and precision. This can be an important consideration for applications with limited memory resources.

After achieving satisfactory output using floating point computation, we proceeded to implement the filters using fixed point arithmetic. While we were able to achieve satisfactory output for the low pass filter using fixed point arithmetic, we encountered difficulties in generating fixed point output for the high pass filter. Despite our best efforts, we were unable to achieve the desired output using fixed point arithmetic.

5.7.2 Filter Structure

For implementing the Lowpass filter using Fixed point arithmetic, depending upon the input values from ADC and the coefficient values obtained by filter we calculated the fixed point data format at every point.
For example :

- Input from ADC = 12 bits (0 to 4095)
  Since C language does not have a specific data type for 12-bit integers, we decided

to use signed integer 16 to represent the input values. Signed int 16.4 format, enables us to maintain the input and output data formats of the filter consistent while providing maximum precision.

- Coefficient values = Ranges from -2.55 to +2.6 They range from -2.55 to +2.6, and require a maximum of 3 bits to represent the integer part. For achieving maximum precision in representing the fractional part of the coefficients, we decided to use signed integer 16 as the data format for all the coefficients of the Lowpass filter. Signed int 16.13 format, ensures that the coefficients can be represented with the highest possible precision.

Filter structure with data formats is shown below :

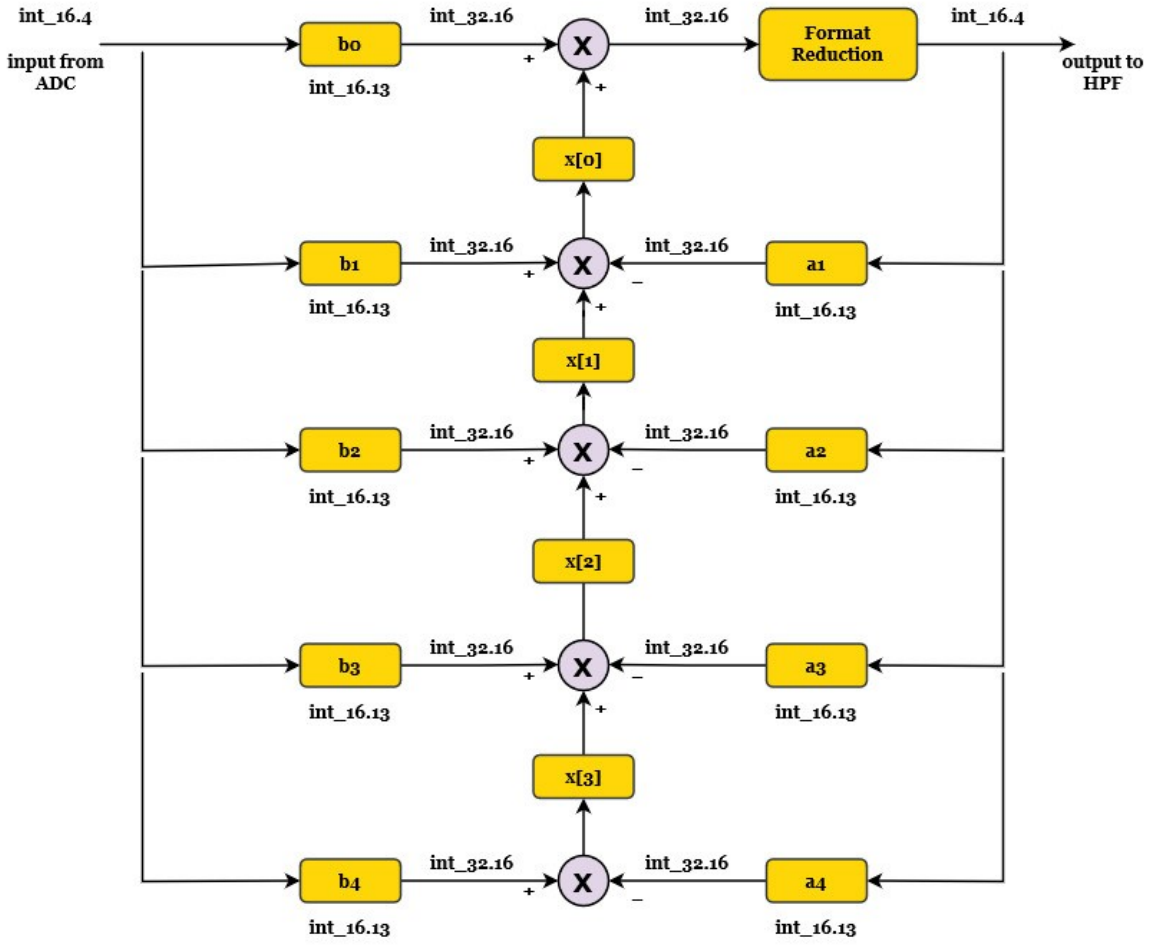


Figure 5.23: Filter Structure - Fixed Point Arithmetic

5.7.3 Double to Fixed point integer conversion

To convert a floating point number into fixed point data format, we utilized a function that takes two input arguments: the first argument is the input data which is represented in floating point format, and the second argument is a value that determines the precision of the fixed point data format.
The function first checks the sign of the input data and determines whether it is positive or negative. If the input data is positive, the function shifts the bits of the input data to the left by N positions to generate the fixed point data format. On the other hand, if the input data is negative, the function shifts the bits of the input data to the right by N positions to generate the fixed point data format.
By using this function, we can accurately convert floating point numbers into fixed point data format with a specific precision. This enables us to maintain the consistency of the data format throughout the processing of the signal.

```
inline signed int double_to_fixed(double input, int num) {
        return (input * pow(2,num));
}
```

Input: **IIR_Lpf.num[0] = double_to_fixed(0.003227021246802, 13);**

Function performs: **(0.003227021246802<<13)**

Ouput: **26**

Figure 5.24: Double to Fixed point data type conversion

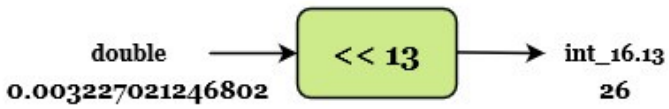


Figure 5.25: Double to Fixed point data type conversion

5.7.4 Format Reduction

After performing the computations, the result's data format is signed int 32.16. However, to maintain the consistency of the input and output data formats, we need to down-convert the result to a signed int 16 format. However, down-converting the result to a

signed int 16 format will cause us to lose the precision bits.
To maintain a 4-bit precision, we first left shift the result since we know that the integer part of the result will not exceed beyond 12-bits (ADC value ranges from 0 to 4096). By left shifting the result, we preserve the precision bits that would have been lost in the down-conversion process.
After the left shift, we down-convert the result to a signed int 16.4 format. This format enables us to maintain a 4-bit precision while ensuring that the input and output data formats of the system remain consistent.

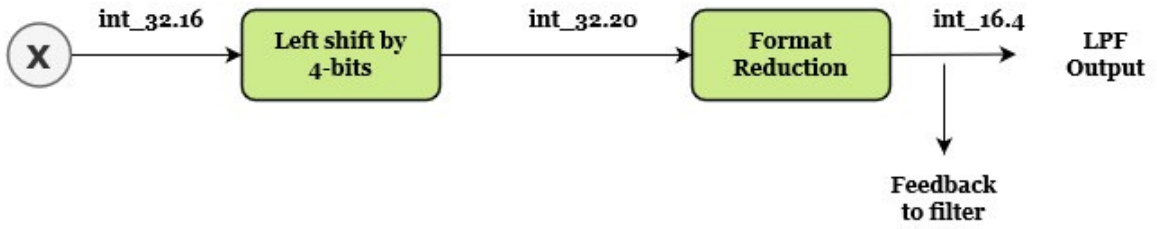


Figure 5.26: Format reduction at Lowpass Filter Output

5.7.5 Filter Output

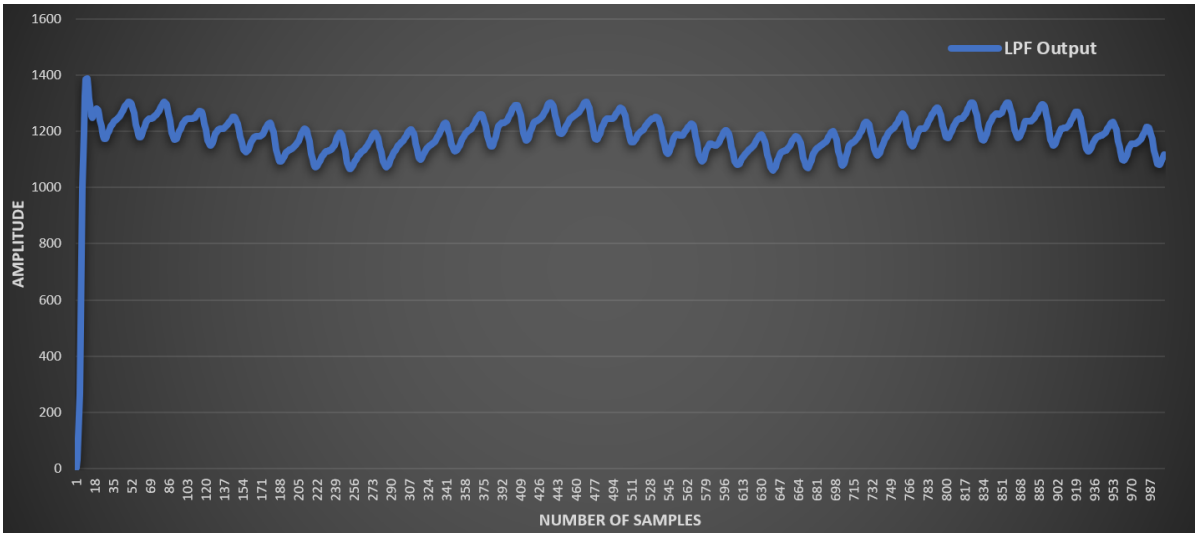


Figure 5.27: Filter Output using Fixed Point Arithmetic

6 Blood Oxygen Saturation and Heart Rate

6.1 Beer-Lamber Law

Pulse oximetry is a method of measuring the impact of arterial blood on tissue's transmitted light intensity. The volume of blood in the tissue depends on the arterial pulse, with systole having a greater volume and diastole having a smaller volume. Since blood mostly absorbs the light passing through the tissue, the light's intensity that emerges from the tissue is inversely proportional to the volume of blood present. The light intensity varies with the arterial pulse and can be utilized to determine a patient's pulse rate. Furthermore, oxyhemoglobin's absorbance coefficient differs from that of deoxygenated hemoglobin for most light wavelengths. Variations in the amount of light absorbed by the blood at two different wavelengths can be used to determine hemoglobin oxygen saturation, which is equivalent to

$$\%S_aO_2 = [HbO_2]/([Hb] + [HbO_2]) \times 100\% \tag{6.1}$$

The absorbance of light by uniform absorbing substances is governed by the Beer-Lambert law. Incident light with an intensity of I_0 is directed towards an absorptive substance with a characteristic absorbance factor A that measures its attenuating effect and a transmittance factor T that is the inverse of the absorbance factor (1/A). The intensity of the light emerging from the substance, I_1, is lower than the incident light I0 and can be expressed as the product TI_0. The intensity of the emergent light, I_n, passing through a medium divided into n identical parts, each with a thickness of one unit and the same transmittance factor T, can be expressed as T^n I_0. This equation can be made more convenient by equating T^n to $e^{-\alpha n}$, where α is the absorbance of the medium per unit length and is also known as the relative extinction coefficient. The relative extinction coefficient α is linked to the extinction coefficient ϵ, with $\alpha = \epsilon$C, where C is the concentration of the absorbing material. The Beer-Lambert law is the general equation used to calculate the intensity of the light T^n emerging from a medium.

$$I_n = I_0 e^{-\alpha d} \tag{6.2}$$

The intensity of the emergent light, I_n, is determined by the incident light intensity, I_0, the absorbance coefficient of the medium per unit length, α, and the thickness of the medium, d, which is expressed exponentially in base e. This equation, also known as equation (2), illustrates the exponential decay of light through a uniform absorbing medium, and is commonly known as the Beer-Lambert law (as shown in Figure 1.1).

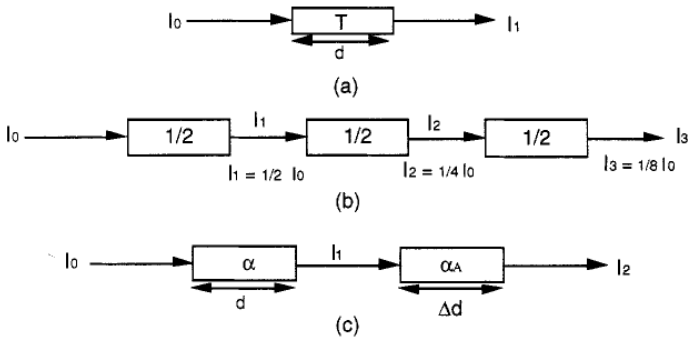


Figure 1.1. A block diagram illustrating the transmittance of light through a block model of the components of a finger.(a)When light with an initial intensity of I_0 interacts with an absorptive medium, it passes through a medium that has a characteristic transmittance factor T. (b) When incident light with an intensity of I_0 interacts with a medium that is divided into n identical components, each with a unit thickness and the same transmittance factor T, its intensity is affected.(c) In a model of a finger, both the unchanging absorptive elements and the changing absorptive portion that pulsates are represented, forming a baseline component and a pulsating component, respectively.

6.1.1 Estimation of oxygen saturation using the Beer-Lambert law

The absorbance coefficients of oxygenated and deoxygenated hemoglobin differ at most wavelengths, except for the isosbestic wavelength. When a finger is exposed to incident light, the emergent light intensity can be measured to determine the amount of light absorbed, which provides information about the oxygenated hemoglobin content in the blood of the finger. Since the volume of blood in the finger changes with each arterial pulse, and the thickness of the finger also changes slightly with each pulse, the path length for transmitted light through the finger also changes. Additionally, determining the precise intensity of the incident light applied to the finger is difficult. Therefore, it

is desirable to eliminate the effects of incident light intensity and path length thickness when estimating oxygen saturation. Modifying the Beer Lambert law is necessary to remove these variables.

6.1.1.1 Eliminating the input light intensity as a variable.

The amount of light that can pass through a finger depends on the absorption coefficient of fixed components such as bone, tissue, skin, and hair, and variable components such as the volume of blood. The light transmitted through the tissue can be expressed over time, with a slowly varying baseline component that represents the effect of fixed components and a rapidly changing pulsatile component that represents the effect of changing blood volume. The baseline component has a thickness of d and an absorbance coefficient of α, while the pulsatile component has a thickness of Δd and an absorbance coefficient of α_{A}, representing arterial blood absorbance (as shown in figure 1.l(c)).

The light emerging from the baseline component can be written as a function of the incident light intensity I_0 as follows

$$\mathrm{I}_1 = I_0 e^{-\alpha d}$$

Likewise, the intensity of light I_2 emerging from the pulsatile component is a function of its incident light intensity I_1 and can be written as follows

$$I_2 = I_1 e^{-\alpha_A \Delta d} \tag{6.3}$$

Substituting the expression of I_1 in the expression for I_2, the emerging from the finger as a function of the incident light intensity I_0 is as follows

$$I_2 = I_0 e^{-[\alpha d + \alpha_A \Delta d]} \tag{6.4}$$

The relationship between the emergent light intensity I_2 and the incident light intensity I_1 reflects the impact of the arterial blood volume on the light. This effect can be

expressed as the change in transmittance caused by the arterial component, which is represented as $T_{\Delta A}$.

$$T_{\Delta A} = I_2/I_1. \tag{6.5}$$

Substituting the expressions for I_1 and I_2 in the above equation yields the following:

$$T_{\Delta A} = (I_0 e^{-[\alpha d + \alpha_A \Delta b]})/(I_0 e^{-\alpha d}) \tag{6.6}$$

The input intensity as a variable in the equation can be removed, which results in the cancellation of the term I_0 in both the numerator and the denominator. Therefore, the change in arterial transmittance can be expressed as

$$T_{\Delta A} = e^{-\alpha_A \Delta d} \tag{6.7}$$

A device that uses this principle is self-calibrating in its operation and does not rely on the incident light intensity I_0.

2.1.2 Eliminating the thickness of the path as a variable. The changing thickness of the finger, Δd, produced by the changing arterial blood volume remains a variable in equation (8). To further simplify the equation, the logarithmic transformation is performed on the terms in equation (8) yielding the following

$$InT_{\Delta A} = In(e^{-\alpha_A \Delta d}) = -\alpha_A \Delta d. \tag{6.8}$$

To remove the variable Δd from the equation, arterial transmittance can be measured at two different wavelengths. These two measurements will yield two equations with two unknowns. The choice of wavelengths to be used is influenced by a more comprehensive representation of the arterial absorbance α_A

$$\alpha_A = (\alpha_{OA})(S_a O_2) - (\alpha_{DA})(1 - S_a O_2) \tag{6.9}$$

The variables α_{OA}, α_{DA}, and $S_a O_2$ represent the oxygenated arterial absorbance, deoxygenated arterial absorbance, and oxygen saturation of arterial Hb, respectively. Except for the isosbestic wavelength of 805 nm, α_{OA} and α_{DA} have significant differences at all red and near-infrared wavelengths. If $S_a O_2$ is approximately 90%, then α_{OA}

accounts for 90% of arterial absorbance α_A, while α_{DA} accounts for the remaining 10%. At the isosbestic wavelength, the contribution of α_{OA} and α_{DA} to α_A is almost negligible, as both coefficients are the same.

The chosen wavelengths are not near the estimated isosbestic point, ensuring that the two signals can be easily differentiated. Typically, it is desirable for the selected wavelengths to be in the red and infrared parts of the electromagnetic spectrum. Equation (9) provides the ratio of the arterial blood component's transmittance at red and infrared wavelengths.

$$\frac{InT_{\Delta AR}}{InT_{\Delta AIR}} = \frac{-\alpha_A(\lambda_R)\Delta d}{-\alpha_A(\lambda_{IR})\Delta d} \tag{6.10}$$

$T_{\Delta AR}$ denotes the shift in the amount of red light passing through the artery at wavelength λ_R, while $T_{\Delta AIR}$ refers to the variation in the amount of infrared light passing through the artery at wavelength $\lambda_I R$. If both light sources are placed near the same spot on the finger, the distance that the light travels through the finger is roughly equal for the light emitted by each LED.Therefore, the modification in the distance traveled by light due to the blood flow (Δd) is almost identical for both red and infrared light sources. This is why the Δdterm in both the numerator and denominator of equation (11) cancel each other out, resulting in a simplified outcome.

$$\frac{InT_{\Delta AR}}{InT_{\Delta AIR}} = \frac{\alpha_A(\lambda_R)}{\alpha_A(\lambda_{IR})} \tag{6.11}$$

Equation (12) is not affected by the intensity of the incoming light (I_0) or the difference in finger thickness (Δd) caused by arterial blood flow. However, due to the complicated nature of the physiological processes, the ratio given in equation (12) cannot provide an accurate measurement of oxygen saturation directly. Instead, the correlation between the ratio calculated using equation (12) and actual arterial blood gas measurement is relied upon to determine the oxygen saturation level. Therefore, if the ratio of arterial absorbance at red and infrared wavelengths is known, the oxygen saturation level of arterial blood flow can be determined by using empirically calibrated curves, which depend on I_0 and Δd. To simplify this process, a measured ratio called R_{OS} is defined using equation (12).

$$Ratio = R_{OS} = \frac{\alpha_A(\lambda_R)}{\alpha_A(\lambda_{IR})} \tag{6.12}$$

6.2 RATIO OF RATIOS

The Ratio of Ratios by using the distinct AC and DC elements of the recorded signal. To derive this mathematical expression, Yorkey employs the Beer-Lambert equation.

$$I_1 = I_0 e^{-\alpha L} \tag{6.13}$$

This technique involves the use of the Beer-Lambert law, where I_1 represents the intensity of the light after passing through a material, I_0 is the initial light intensity, α is the relative extinction coefficient of the material, and L is the distance the light travels. To determine the Ratio of Ratios, the derivatives are utilized. It is assumed that the alteration in the distance traveled by the light is identical for both wavelengths during the sampling period. Therefore, the instantaneous change in the length of the path (dL/dt) must be the same for both wavelengths.

We can apply the process of finding the derivative of e^U to our particular scenario.

$$\frac{de^u}{dt} = e^u \frac{du}{dt} \tag{6.14}$$

$$\frac{dI_1}{dt} = I_0 e^{-\alpha L} - (\alpha \frac{dL}{dt}) \tag{6.15}$$

Therefore,

$$\frac{(dI_1/dt)}{I_1} = -\alpha \frac{dL}{dt} \tag{6.16}$$

In this context, I_1 refers to the summed AC and DC component of the waveform, while dI_1/dt refers to the rate of change of the AC component of the waveform. Using two wavelengths we have

$$RofR = \frac{(dI_R/dt)/I_R}{(dI_{IR}/dt)/I_{IR}} = \frac{-\alpha(\lambda_R)}{-\alpha(\lambda_{IR})} \tag{6.17}$$

Rather than relying on the earlier approach of computing the Ratio of Ratios by taking the logarithm of the maximum and minimum values of the red and infrared signals, we can determine the R of R by utilizing the derivative value of the AC component of the waveform.

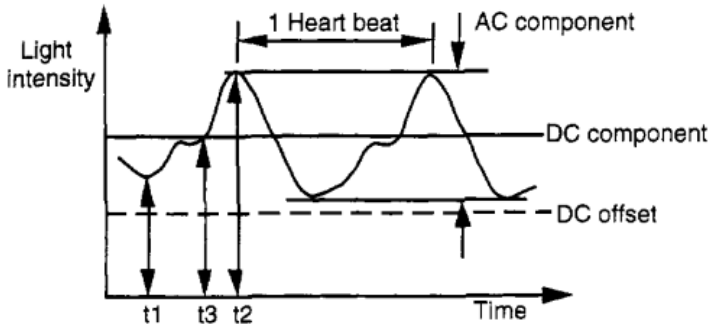


Figure 1.3. A waveform of the transmitted light intensity through a finger showing the AC component, the DC component and DC offset.

Note in discrete time

$$\frac{dI_R(t)}{dt} = I_R(t_2) - I_R(t_1) \tag{6.18}$$

By selecting the highest value of the waveform as I_2 and the lowest value as I_1, we can define the difference between them as the AC value. The denominator can be calculated at a specific time point I_3 , which lies between I_2 and I_1, and we can refer to this value as the DC value. Therefore, the expression becomes:

$$\frac{\frac{dI_R(t)/dt}{I_R}}{\frac{dI_{IR}(t)/dt}{I_{IR}}} = \frac{\frac{I_R(t_2)-I_R(t_1)}{I_R(t_3)}}{\frac{I_{IR}(t_2)-I_{IR}(t_1)}{I_{IR}(t_3)}} = \frac{\frac{AC_R}{DC_R}}{\frac{AC_{IR}}{DC_{IR}}} = R. \tag{6.19}$$

6.3 Heart-rate Calculation using PPG Signal

PPG sensors measure changes in light absorption caused by variations in blood volume in the microvascular bed of tissue. When the heart beats, it creates a pressure wave that propagates through the arteries and causes a pulsatile change in blood volume.
By placing a PPG sensor on the skin over an artery, such as the radial artery in the wrist, it is possible to measure the pulsatile changes in blood volume caused by each heartbeat. These changes in blood volume can be used to calculate heart rate by analyzing the timing between successive peaks in the PPG waveform.

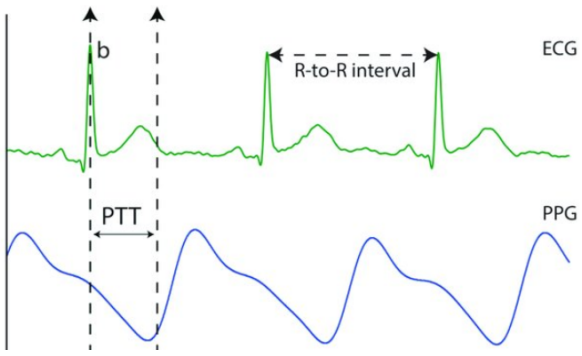


Figure 6.1: Heart-rate calculation using PPG signal

Pulse transit time (PTT) is the time it takes for a pressure wave, generated by the heartbeat, to propagate through the arterial system from one point to another. PTT is typically measured as the time difference between the R wave of an electrocardiogram (ECG) and the arrival of the corresponding pulse wave at a peripheral artery, such as the radial artery in the wrist. PTT is affected by the elasticity and stiffness of the arterial walls, as well as the diameter and length of the arterial segments. When the arterial system is more elastic, the pressure wave propagates more quickly, resulting in a shorter PTT.

The PPG signal is typically processed by an algorithm that detects the peaks in the waveform and calculates the time intervals between them. The time interval between two consecutive peaks corresponds to the duration of one heartbeat, also known as the R-R interval. The heart rate can then be calculated by taking the reciprocal of the R-R interval, which gives the number of heartbeats per minute.

7 C Code Development

7.1 Overall Project Development

The overall project development was planned prior to starting the project, and the various tasks and responsibilities were shared among the team members.

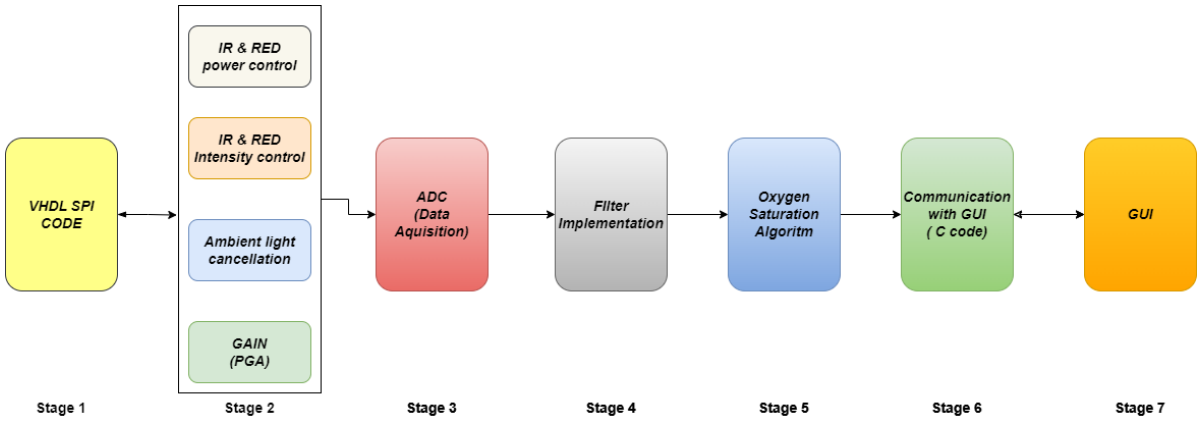


Figure 7.1: Project Development Chart

- Stage 1 : VHDL Code
- Stage 2 to 6 : C code
- Stage 7 : Java code

7.2 Main Code Flow

The primary focus is to keep the main flow of the program simple, with repetitive functions being moved to separate functions that are then called in the main flow. This approach aims to reduce complexity and make the code more modular and easier to maintain.

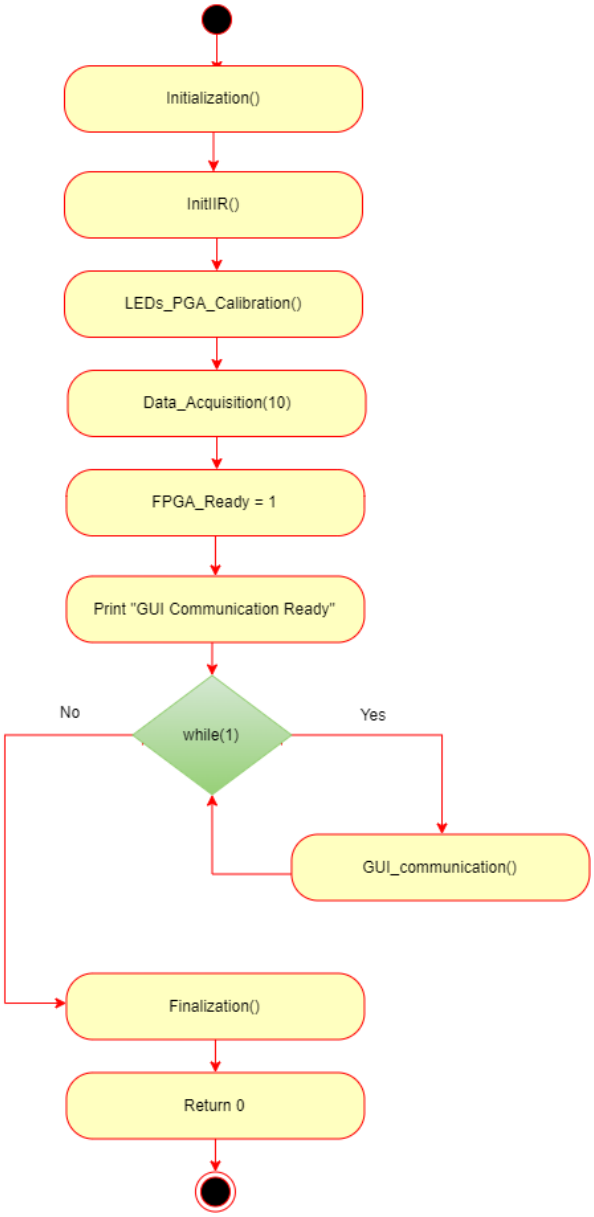


Figure 7.2: Main Code Flow

- Initialization() : This function runs when the system first starts up or is reset, and is responsible for configuring things like timers, interrupts, and peripheral devices such as analog-to-digital converters (ADCs).
- InitIIR() : The InitIIR() function is used to define the coefficients of a Lowpass and Highpass filter that have been designed to meet the requirements of a particular application. These coefficients are used to calculate the filter's output based on

its input signal, using a recursive algorithm that incorporates feedback from the filter's previous outputs.

- LEDs_PGA_Calibration() : The LED PGA Calibration function is a critical part of an Automatic Finger Detection system. This function is responsible for varying the intensity of LEDs by adjusting the gain of the programmable gain amplifiers (PGAs) depending on the user's needs. This process is ensures that each LED is calibrated to the correct intensity level for the user's needs. Once the LED PGA Calibration function is complete, the system is ready for data acquisition and GUI communicaion.

- Data_Acquisition() : . This function is responsible for collecting n no. data samples for both the red and IR LEDs, after the LED PGA calibration has been performed.The purpose of collecting these data samples is to measure the values obtained from the LEDs after calibration, and to ensure that they fall within the expected range.

- GUI_communication() : The GUI Communication function is responsible for handling the real-time communication between the system and the GUI, which provides a graphical interface for the user to interact with the system. This function is responsible for sending and receiving data between the system and the GUI, including information about the blood oxygen levels measured by the system, as well as any calibration or configuration settings that may need to be adjusted.

- Finalization() : The Finalization function is a critical part of the system that is responsible for clearing all registers, timers, counters, interrupts, and other system resources that were allocated during the initialization process. This function is called at the end of the system's operation, when it is no longer needed. The purpose of the Finalization function is to ensure that all system resources are properly released and that the platform is left in a clean and stable state.

7.3 Firmware

Real-time firmware has been implemented in order to collect data from a finger probe in real-time. The collected data is then filtered in real-time and displayed in a Graphical User Interface (GUI), providing users with up-to-date and accurate information.

- 4 Layer Architecture The 4-layer architecture has been designed to separate different layers of the code, with the aim of improving the maintainability and resilience

of the system. By dividing the code into distinct layers, it becomes easier to isolate and resolve issues, and to make changes or updates to specific components without affecting the entire system.

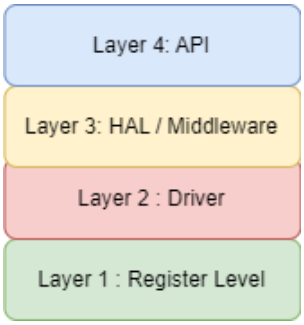


Figure 7.3: Multi-layered Firmware

7.3.1 Layer 1 : Register Level Functions

The functions directly access the registers of the slaves, enabling read and write operations to be performed on all four slave registers.

- void clear_reg(void)
- unsigned int Read_SR(POXI_REG REG)
- void Write_SR(POXI_REG REG, unsigned int SR_data)

7.3.2 Layer 2 : Driver functions

A SPI driver has been developed to facilitate communication between the IP in PL (Programmable Logic) and external devices such as DAC, ADC, and PGA. The driver has been designed to be simple to implement and utilize, making it easier for developers to integrate SPI communication into their future projects. This layer has SPI functions, which can be used with any device.

- void SPI_start(void)
- void SPI_Tx(unsigned int data)
- unsigned int SPI_Rx(unsigned int data)
- void SPI_config(void)

7.3.3 Layer 3 : HAL / Middleware functions

These functions are hardware-specific and utilize layer 2 functions.

- void DAC_Control(unsigned char channel, unsigned short int data)
- void DAC_SPI_config(void)
- void PGA_SPI_config(void)
- void ADC_SPI_config(void)
- void Send_Data_SPI(IC Device, unsigned int data)

7.3.3.1 Description of DAC_Control

- converts the received voltage into corresponding integers as per DAC
- frame DAC command as per channel no.
- send command to DAC

```
void Send_Data_SPI(IC Device, unsigned int data){

    if (Device == DAC){
        DAC_SPI_config();
    }
    else if(Device == PGA ){
          PGA_SPI_config();
    }
    else if (Device ==ADC){
          ADC_SPI_config();
        }

    SPI_config();


    SPI_Tx(data);


}
```

Figure 7.4: Send_Data_SPI

```
void DAC_Control(unsigned char channel, unsigned short int data){

    unsigned int vol_dec,dac_cmd=0;
    float temp=0;

       temp =  (((data*4096.0)/2500) - 1); // convert to integer
       vol_dec =  0x00000FFF & (unsigned int) temp; //masking
       dac_cmd = ( (0<< 20) | ((channel-1) <<16) | (vol_dec<<4)); // cmd frame

       Send_Data_SPI(DAC,dac_cmd); // send cmd to DAC

    #ifdef DEBUG1
            //printf(" cmd => 0x%x %d\n",dac_cmd,vol_dec);
    #endif

}
```

Figure 7.5: DAC_Control

7.3.4 Layer 4: API functions

Theses are application-specific functions built upon layer 3 functions

- unsigned int ADC_Read(void)
- void Ambient_light_cancellation(unsigned short dc_mv)
- void Heart_Beat_LED(unsigned short heartbeat_mv)
- void Data_Acquisition(unsigned int Sample_No)
- void Reset_ADC_buffer(void)
- void print_data(bool data_type)
- void REDOn(void)
- void REDOff(void)
- void IROn(void)
- void IROff(void)
- void Probe_LED_Intensity_Control(float current)
- void Initialize_DAC(void)

7.3.4.1 Description of Initialize_DAC

1. clear data command
2. set internal reference command
3. set LDAC command
4. set the light intensity of RED and Ir as zero
5. set heartbeat LED output zero
6. set ambient light channel output zero

7.3.4.2 Description of Probe_LED_Intensity_Control

1. converts the current into the corresponding voltage
2. sends the voltages to IR and RED intensity control channel

```
void Probe_LED_Intensity_Control( float  current){

    float temp =0;
    unsigned int data =0;
    unsigned int dac_voltage =0;

    if(current < MAX_CURRENT){          // to avoid access current

    dac_voltage = current * (DIV_FACTOR * RESISTANCE);

    // printf ("\r\n I : %3.3f mA  V : %d mv ",current,dac_voltage);

     DAC_Control(CH1,dac_voltage); // RED light intensity control

     DAC_Control(CH2,dac_voltage); // IR light intensity control
    }

}
```

Figure 7.6: Probe_LED_Intensity_Control

The detailed interlinking of function: Probe_LED_Intensity_Control

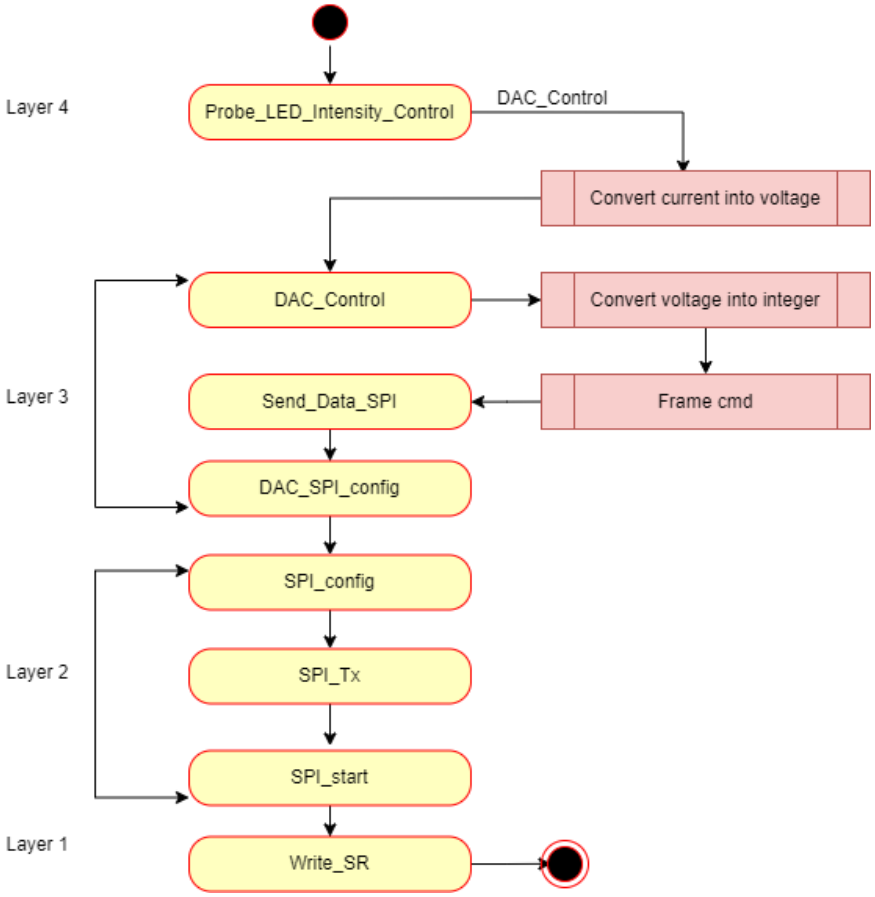


Figure 7.7: interlinking of function: Probe_LED_Intensity_Control

7.4 GUI Communication

The communication between GUI and SoC has been done via UART @ 115200 baud rate, 1 stop bit and 1 start bit.

- The UART frame contains 10 bit: 1 start, 1 stop and 8-bit data
- Time to send 1 bit = 1/115200 = 8.68 microseconds.
- Time to send 10 bits(1 byte of data) = 86.8 microseconds.

Figure 7.8: UART Frame

7.4.1 Transmission Packets

1. FPGA Ready packet
    - byte 1: start_byte
    - byte 2: data_length
    - byte 3: FPGA_Start_Success
    - byte 4: checksum
    - byte 5: stop_byte
    - byte 6: cr
    - byte 7: lf
2. Data Packet
    - byte 1: start_byte
    - byte 2: data_length
    - byte 3: FPGA_Data
    - byte 4: high_byte
    - byte 5: low_byte
    - byte 6: checksum
    - byte 7: stop_byte
    - byte 8: cr
    - byte 9: lf

7.4.2 Explantion

- The GUI initiates the communication
- The GUI send the start query and Soc shall respond (PASS/FAIL)
- IF SoC sends PASS, GUI shall next send start data query once
- Thereafter the SoC sends data in sync with ISR
- The SoC continues to send real time data untill GUI sends *
- The SoC reset all variables, stop the timer and ready for the next connection with GUI
- After this GUi can again communicate with SoC same as earlier.

7.4.3 Communication failure

- Absent of start and stop byte
- Wrong position of stop byte
- wrong checksum
- unrecognised command
- wrong baud rate

7.4.4 Flowchart

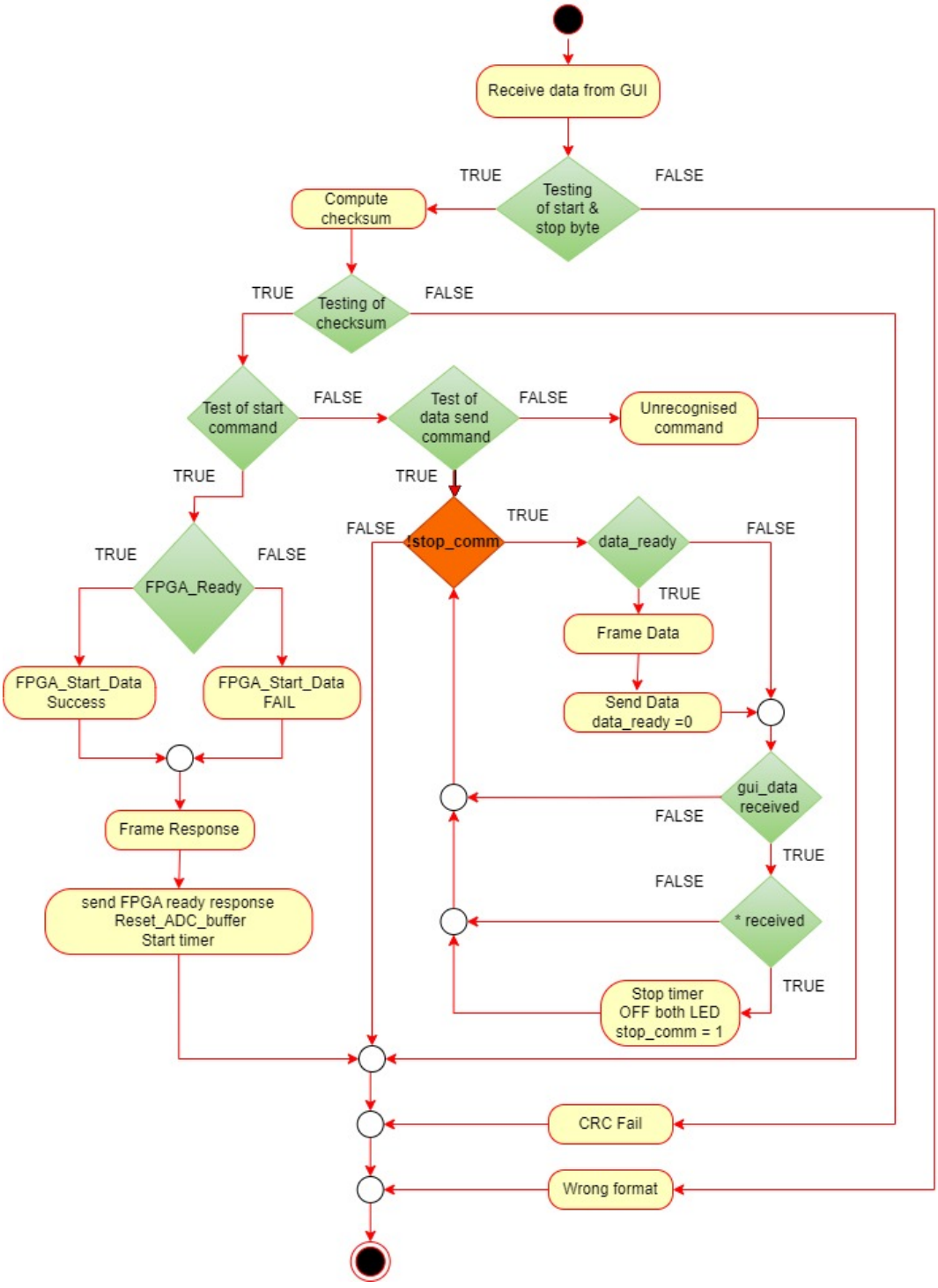


Figure 7.9: GUI Communication

7.5 Timer ISR and Data Acquisition

An interrupt has been used for data acquisition to ensure a consistent sampling rate. The Timer0 has been configured to generate an interrupt every 1.453 milliseconds, and data sampling is performed only within the Timer ISR.

7.5.1 How Interrupt Works !

Interrupts are events that deviate from the regular flow of a program's execution. When an interrupt occurs, the program halts its normal execution flow and starts executing a different program stored at a specific memory location. After the interrupt program finishes its execution, the main program flow resumes from where it was interrupted.

The Xilinx 7000 series FPGA includes several timer modules that can generate interrupts at specific time intervals. The timer module can be configured to count clock cycles and generate an interrupt when a specified count is reached.When the timer module reaches the specified count value, it generates an interrupt that halts the main program flow and calls the registered ISR. The ISR can perform the required operations such as data acquisition, processing or communication, and then return control back to the main program flow.

When multiple interrupts occur simultaneously in a system, it is essential to prioritize them to ensure that critical tasks are executed first. The nested vector interrupt controller (NVIC) is a hardware component that manages the priority and handling of interrupts

7.5.1.1 General Actions on Interrupts

1. PUSH Flags
2. PUSH Program Counter (PC) into the STACK
3. Identify Interrupt Sources
4. Read ISR address assigned for the raised interrupt and set to PC
5. Execute instructions from ISR
6. Clear and Exit ISR
7. POP PC value from STACK

8. POP Flags

7.5.1.2 Importance of volatile keyword

The importance of using volatile in interrupts lies in the fact that interrupts can modify data at any time, including during the execution of a program. If a variable is not marked as volatile, the compiler may optimize the code by caching the variable in a register or optimizing away some read or write operations. This can lead to unexpected behaviour, as the code may not reflect the latest value of the variable.

By using the volatile keyword, we ensures that the variable is always read or written from its actual memory location.

Note: All the variables accessed in the timer ISR have been declared as volatile.

7.5.2 Sampling Rate Calculation

- Fs = 43 Hz ; from Matlab
- Ts = 1/(4* Fs) = 1/(4*43) = 5.813ms (Frequency 1/5.813ms = 172.02 Hz)
- Tint = Ts/4 = 5.813ms/4 = 1.453ms

7.5.3 Timer Interrupt Calculation

- Timer clock is half of CPU clock (650 Mhz) => 650/2 => 325Mhz
- Timer Load value = Interrupt time * Timer frequency
- Timer Load value = (1.453* 10e-3) * (325*10e+6) = **472225**

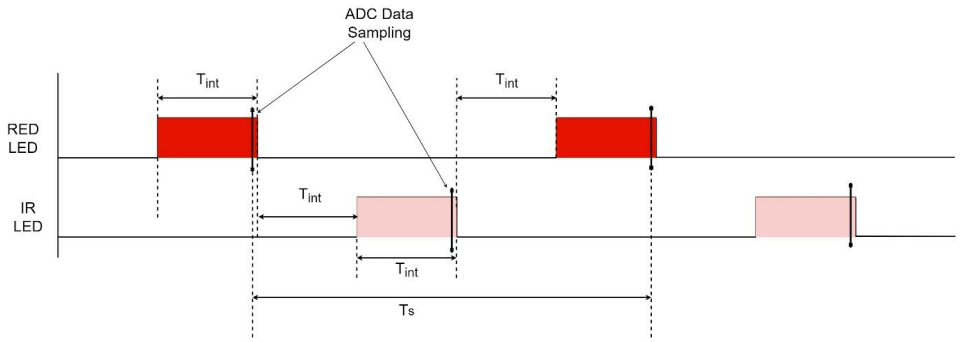


Figure 7.10: LED's control graph

7.5.4 Control of Timer

- Start timer function : **XScuTimer_Start(&pTimer)**
- Stop timer function : **XScuTimer_Stop(&pTimer)**

7.5.5 ISR Task Sequence

A sequence of operations is performed in the interrupt service routine and described below.

1. Control of RED and IR LEDs
2. Data Acquisition
3. Filtering of data
4. Synchronization with GUI communication

7.5.6 ISR Code

The Timer ISR code is shown below.

```
static void TimerIntrHandler(void *CallBackRef)
{

    XScuTimer *TimerInstance = (XScuTimer *)CallBackRef;
    XScuTimer_ClearInterruptStatus(TimerInstance);

        switch (ISR_Count) {
          case 0:
              REDOn();                                                              // switch ON RED LED
              break;
          case 1:
              ADC_Samples.RED[ADC_Samples.Count] = ADC_Read();                      // Capture RED data
              REDOff();                                                             // switch OFF RED LED
              Filter_Samples.RED[ADC_Samples.Count] = filter(ADC_Samples.RED[ADC_Samples.Count]);  // filter RED data
              break;
          case 2:
              IROn();                                                               // switch ON IR LED
              break;
          case 3:
              ADC_Samples.IR[ADC_Samples.Count] = ADC_Read();                       // Capture IR data
              IROff();                                                              // switch OFF IR LED
              Filter_Samples.IR[ADC_Samples.Count] = filter(ADC_Samples.IR[ADC_Samples.Count]);    // filter IR data
              ADC_Samples.Count++;
              data_ready =1;                                                        // flag for use in main
              break;
        }

        ISR_Count = (ISR_Count + 1) % 4;                                           // Reset counter to 0

        if (ADC_Samples.Count == No_of_samples ){

            ADC_Samples.Ready =1;
            ADC_Samples.Count =0;                                                  // Reset the array index
        }

}
```

Figure 7.11: ISR Code

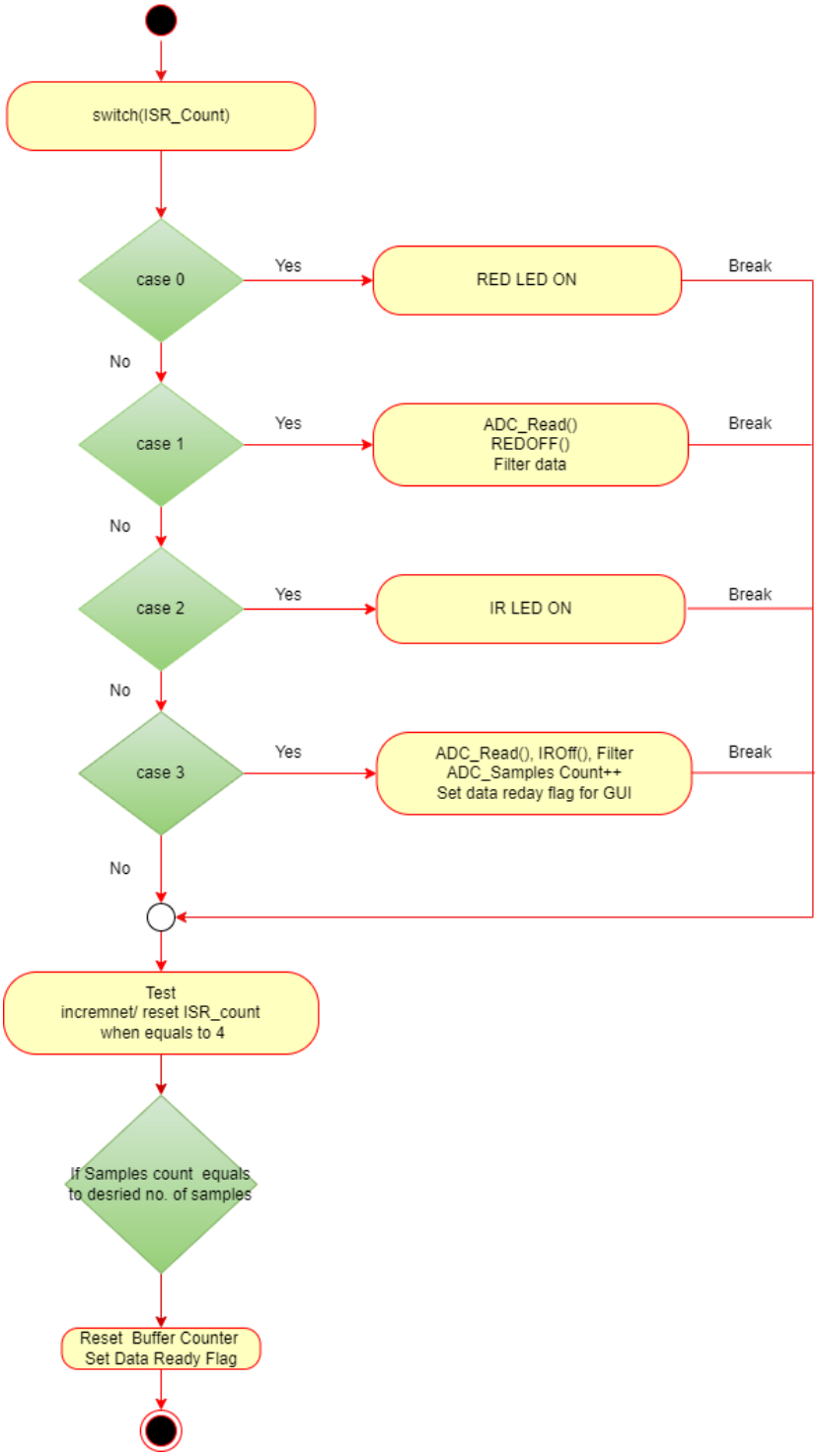


Figure 7.12: ISR Flowchart

7.5.7 Testing and Results

The red and infrared (IR) LED signals have been measured using a digital storage oscilloscope (DSO) and found to have timing values that are very close to the expected values(ON time 1.453 ms). This suggests that the LED signals are being generated correctly and are functioning as intended. **IR LED** Channel 1 , Yellow trace. **RED LED** Channel 2 Green Trace.

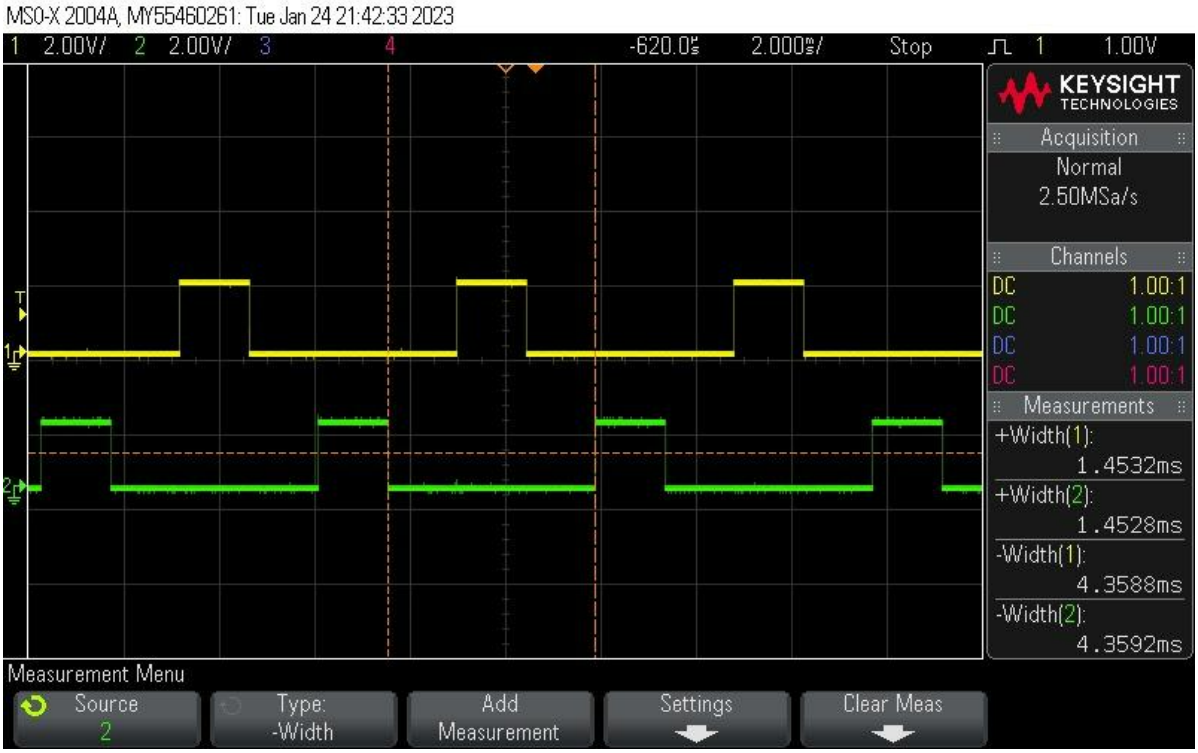


Figure 7.13: DSO captured image

7.6 ELF and Linker Script Explanation

An executable file is created by concatenating input sections from the object files (files) being linked together. The compiler, by default, creates code across standard and well-defined sections. Each section is named based on its associated meaning and purpose. The various standard sections of the object file are displayed in the following figure.

- Section - .text | Read Only: 89,948 Bytes
- Section - .data | Read and Write: 5188 Bytes

- Section - .bss | Read and Write: 60,720 Bytes
- Total Read and Write (.data + .bss) = 65,908 Bytes = 64.36 Kilo Bytes
- Total ELF size = 89940+65908 = 155856 = 152.2 Kilo Bytes

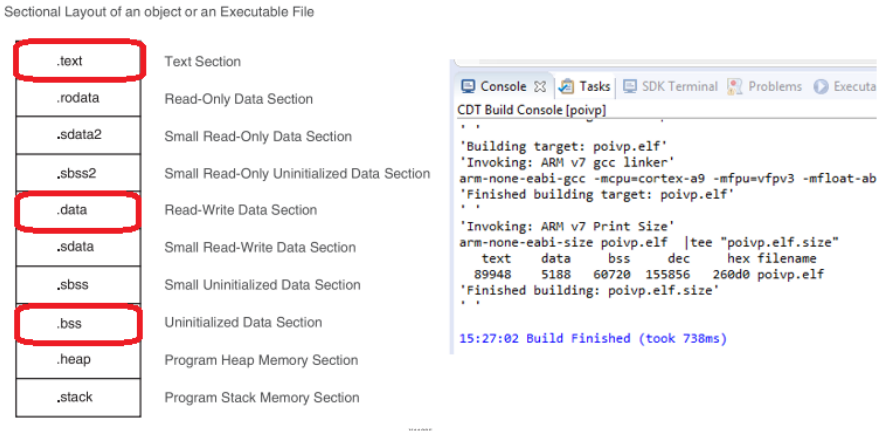


Figure 7.14: ELF sections

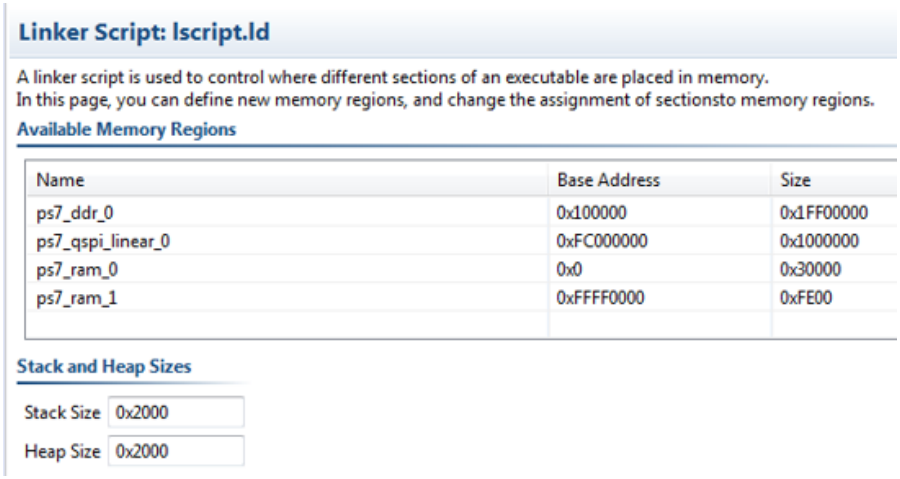


Figure 7.15: Linker Script

8 Communication Protocol

The device communicates with the GUI via a UART communication protocol. For understanding purpose, we will refer the setup of sensor + microcontroller as device and the GUI as host. The software development is done in JAVA for GUI and in C for FPGA. After the sensor has been connected, Communication is established using the following parameters.

- Bits per second—115,200
- Start Bits – 1
- Data bits – Depends on command
- Parity—None
- Stop Bits—1

Communication starts when the host (PC) sends a start packet to the device. The device then responds with a confirmation packet indicating to host that the command has been received and it is ready to send data. Then, the host sends a command packet to device, asking to send the data. At this point, the host must be prepared to receive data packets from the microcontroller and show the data received on the GUI.

Packets sent between host and device have a specific structure. The Packet is divided in four main parts:

- Packet Type
- Data length
- Data
- CRC (Cyclic Redundancy Check)

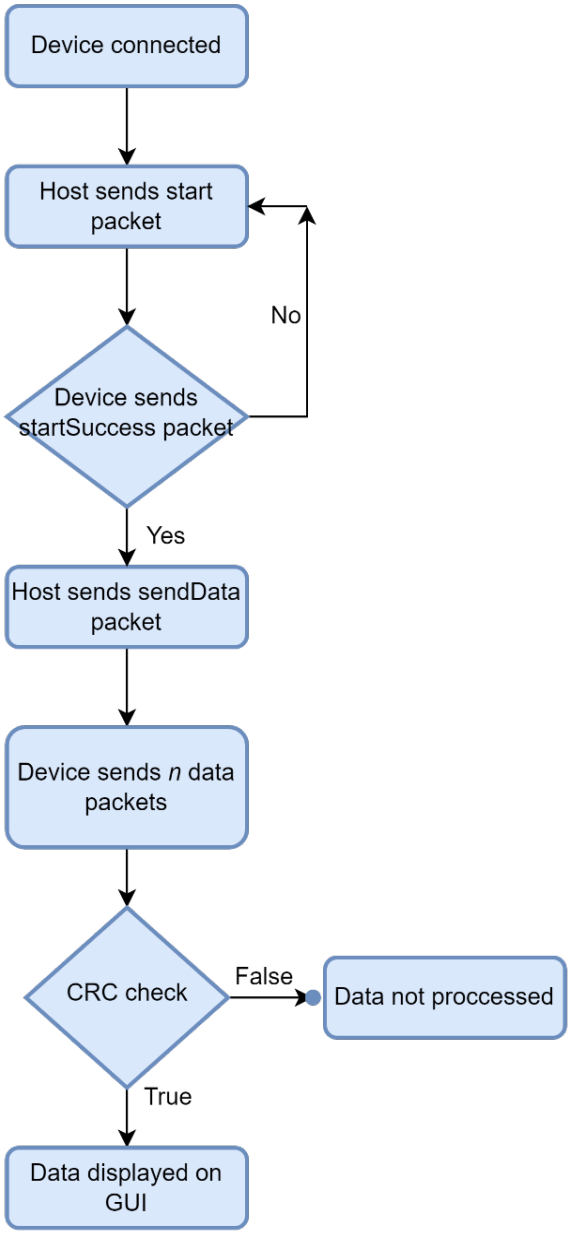


Figure 8.1: Communication Protocol Data Flow

The following block diagram describes the data flow. Each packet starts with a Start Byte (0x24) '$' and ends with an End Byte (0x23)'#'.

Data length is the total number of bytes including the Packet Type and the Data to be sent (As mentioned in the figure below).

The CRC byte XORs all bytes from start type to CRC byte. This is used to detect errors if there is any data loss in communication. It is checked for every incoming packet and the packet is processed only if CRC matches.

CRC = (Start Byte ^ Data length ^ Packet Type^ Data0 ^ Data1^.....^Data n) ;
(^ implies XOR)

The image below shows the packet structure.

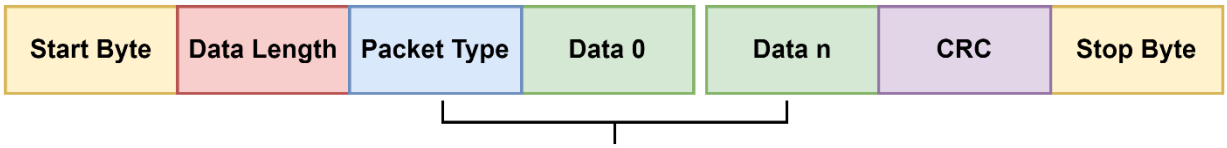


Figure 8.2: Packet Structure

8.1 Packet Type and size

The Packet Type byte defines the kind of packet sent. There are eight kinds of packets that can be sent between host and device.

Sr. No.	**Packet Type**	**Hex Code**	**Mode**	**Data Length**	**Packet Length**	**Description**
1	Start	0x25	Command	1	5	From GUI to device if connection success
2	Start Success	0x26	Command	1	5	From device to GUI if device is Ready to send data
3	Start Fail	0x27	Command	1	5	From device to GUI if device is not Ready to send data
4	Send Data	0x28	Command	1	5	From GUI to device if Device is ready
5	Error	0x32	Error	1	5	Error packet
6	Data	0x29	Data	3	7	From device to GUI after receiving send data

Table 1.1 Packet Type Details

For command Packet types, Data length will always be one, i.e. only the packet type. Hence, the size will be of 5 bytes which is better explained in below example for **Start Packet type:**

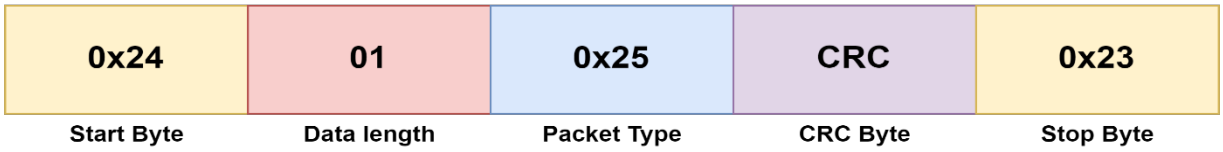


Figure 8.3: Start Packet

The **Data packet** will be as below with size of 7 bytes:

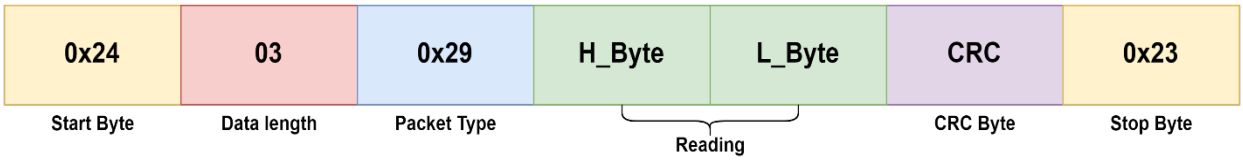


Figure 8.4: Data Packet

8.2 Functional description

Upon starting the GUI, all available ports are displayed and the user selects the respective port of device for communication. If connection is successful, it enables further communication, else it gives error.

When the device is connected successfully and host sends Start Packet to device. Now if the device is ready to send the data, it will send Start Success Packet to host. If device is not ready, it will send Start Fail Packet to host, and host will keep resending the Start Packet until Start Success Packet is not received.

For all packets received, CRC is checked and packet is processed only if CRC matches. If the received packet contains any Packet Type other than that defined in Packet type table (Table 1.1), it will not process that packet and wait till a valid packet is received.

After host receives Start Success Packet from device, it sends Send Data Packet to

device, which instructs the device to start sending the available data to host. Once the host starts receiving the Data Packets, it will decode the packet and display the readings on GUI. After the each Data Packet is received, host has to send Start Packet again to device to get new data.

When the user disconnects the device, communication between device and host is ended and host becomes idle. Hence terminating the communication.

9 Graphical User Interface (GUI)

9.1 Introduction

A graphical user interface (GUI) is an interface through which a user interacts with electronic devices such as computers and smartphones through the use of icons, menus and other visual indicators or representations (graphics). GUIs graphically display information and related user controls, unlike text-based interfaces, where data and commands are strictly in text. GUI representations are manipulated by a pointing device such as a mouse, trackball, stylus, or by a finger on a touch screen.

The user interacts with these elements of the interface (such as buttons, icons and menus), that respond in accordance with the programmed script, supporting each user's action.

For our project, as per the requirement, GUI displays real time data in the respective data fields and plots graph for IR in real time. GUI uses jSerialComm external library to connect the Host with UART port of FPGA (jSerialComm is a Java library designed to provide a platform-independent way to access standard serial ports without requiring external libraries, native code, or any other tools.) and jChart2D external library for plotting the graph. (jChart2D is a Java class library for visualizing quantitative data using two-dimensional category charts)

9.2 Implementation

In Pulse Oximeter project we require different JAR files namely jSerialComm.jar and JChart2D.jar. In order to add the mentioned External Dependency JAR files to the project we have to configure build path and add External JARS as follows:

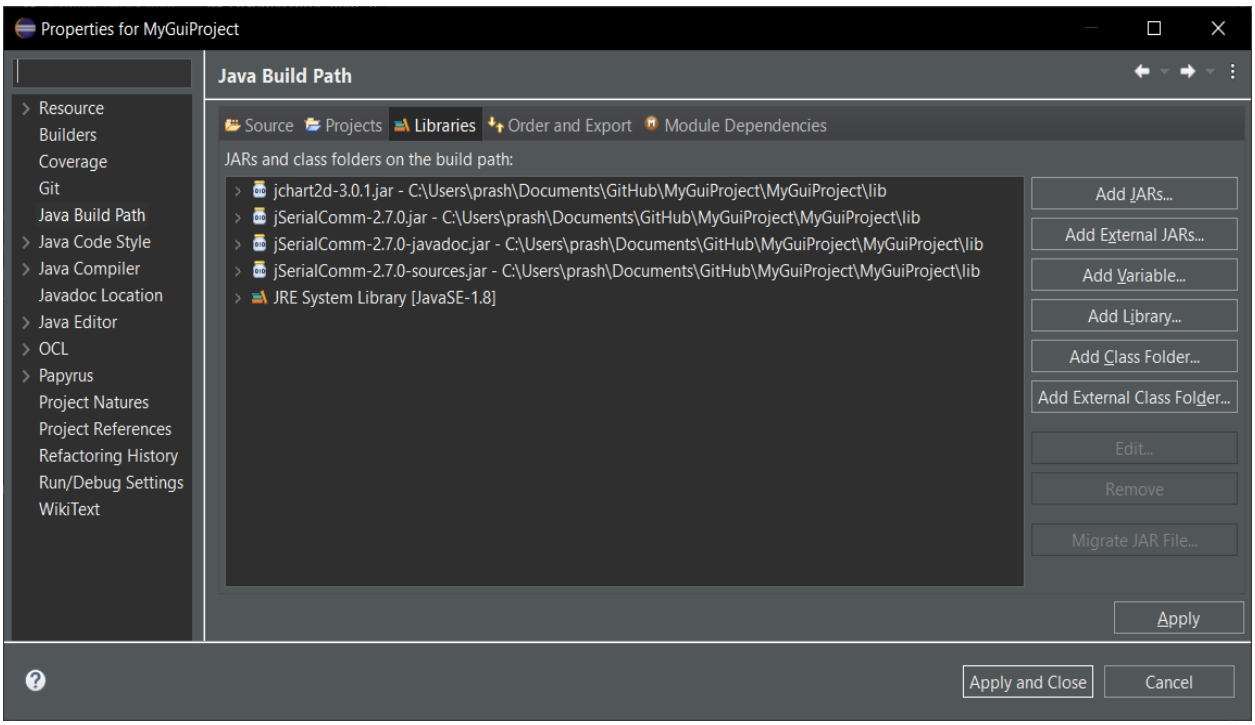


Figure 9.1: Added JAR files

This GUI contains two java files named SerialNetwork.java and DisplayFrame.java. SerialNetwork.java file is to establish Serial Communication between hardware and PC using the jSerialComm library. The code for communication protocol and CRC check is included in this file. This file receives the data, checks it, processes it and returns respective data to the FPGA.

DisplayFrame.java file is to design Front End, initialize the GUI, display readings and plot IR graph. The front end is designed using the FlowLayout and Spring Layout of Java.swing framework.

As soon as the GUI is opened, an internal timer will be activated which will check for any incoming commands to the GUI. The commands may be in the form of Button click or command written in the Command panel. Button click or a command works based on the code in respective Action Listener.

9.3 GUI Contents

The GUI for Pulse Oximeter is designed to look as follows. The Main frame is divided into 5 panels as numbered in image. They are

1. Menu Bar – Contains different options for user.
2. Readings Panel – Contains all buttons for interaction and displays the readings
3. Scroll Panel – will display messages and information for user as per command
4. Graph Panel – Displays the graph when respective command is given
5. Command Panel – Contains an editable text field where user can give command

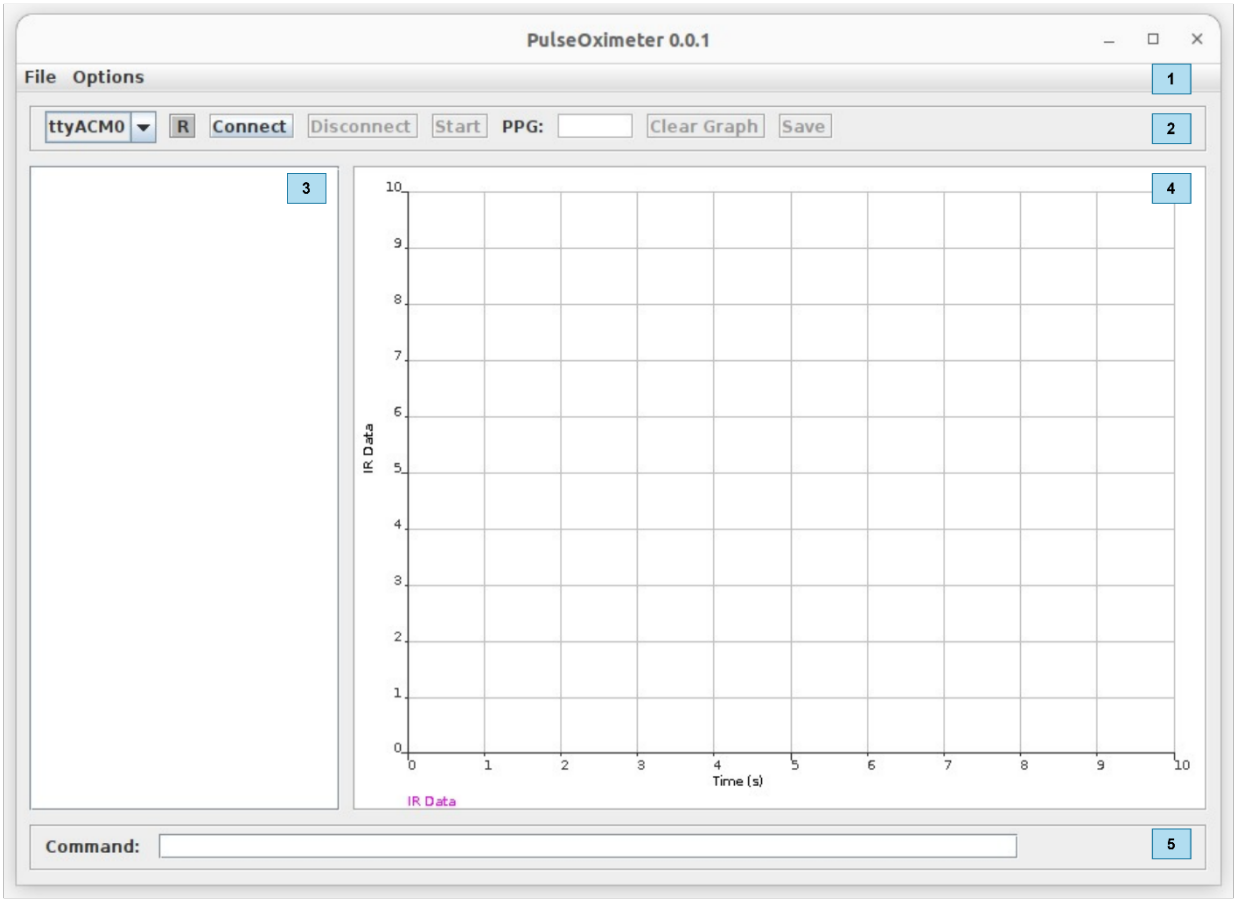


Figure 9.2: GUI Components

At start, all available ports will be displayed in the drop-down menu in Readings Panel. The user has to select respective port connected to device from the drop-down menu. A Refresh ports button "R" is provided to refresh the available ports in case of any new connection. At first only port selection, Refresh and Connect button are enable, all other functions are disabled.

The Command handling with buttons and command input or Menu Bar input will be discussed in coming sections.

9.4 Working of GUI with Buttons

9.4.1 Connect Button

Once the respective port is selected by user and "Connect" Button is clicked, application will check if there is a device available on the mentioned port and if connection is possible. If connection is made, application will be connected to device and a message will be displayed on Scroll panel. After connection, "Start" and "Disconnect" button will be enabled and "Connect" will be disabled. If connection is not successful, it will display error message on Scroll panel and no new buttons will be enabled.

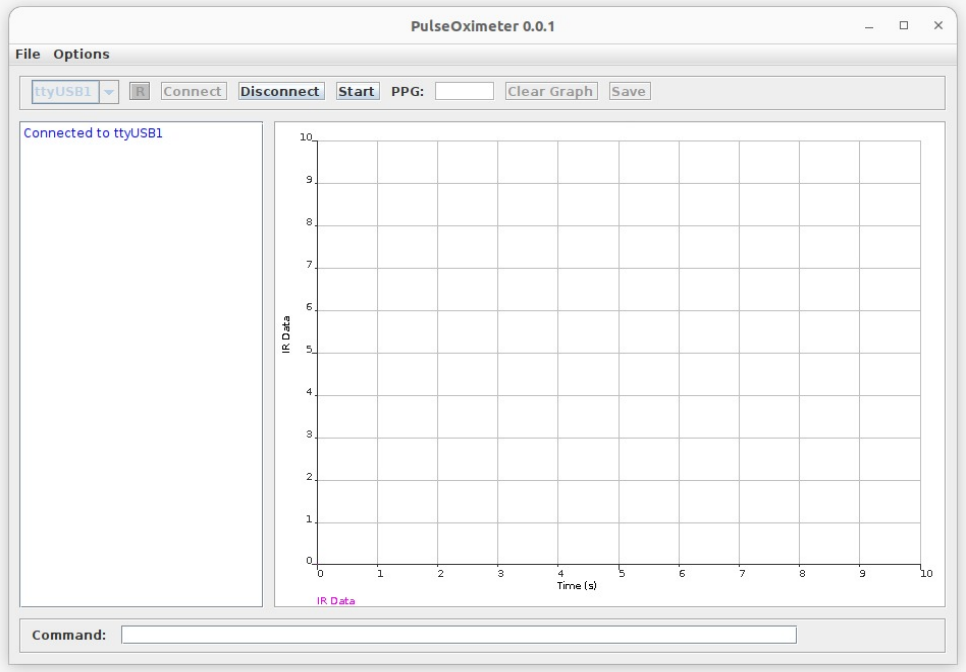


Figure 9.3: Port Connection Successful

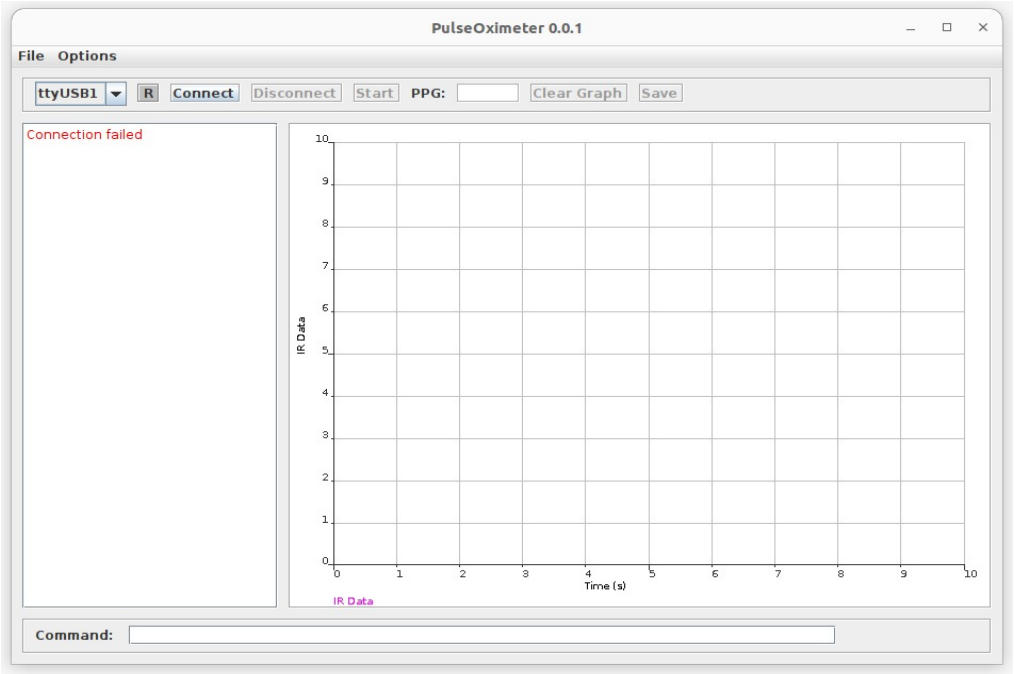


Figure 9.4: Port Connection Unsuccessful

9.4.2 Start Button

When "Start" button is clicked, if device is ready, application will receive data from device which will be displayed on the display box (PPG). Also, real time graph for IR data will be plotted in the Graph Panel. "Start" button will be disabled and "Clear Graph" & "Save" buttons will be enabled.

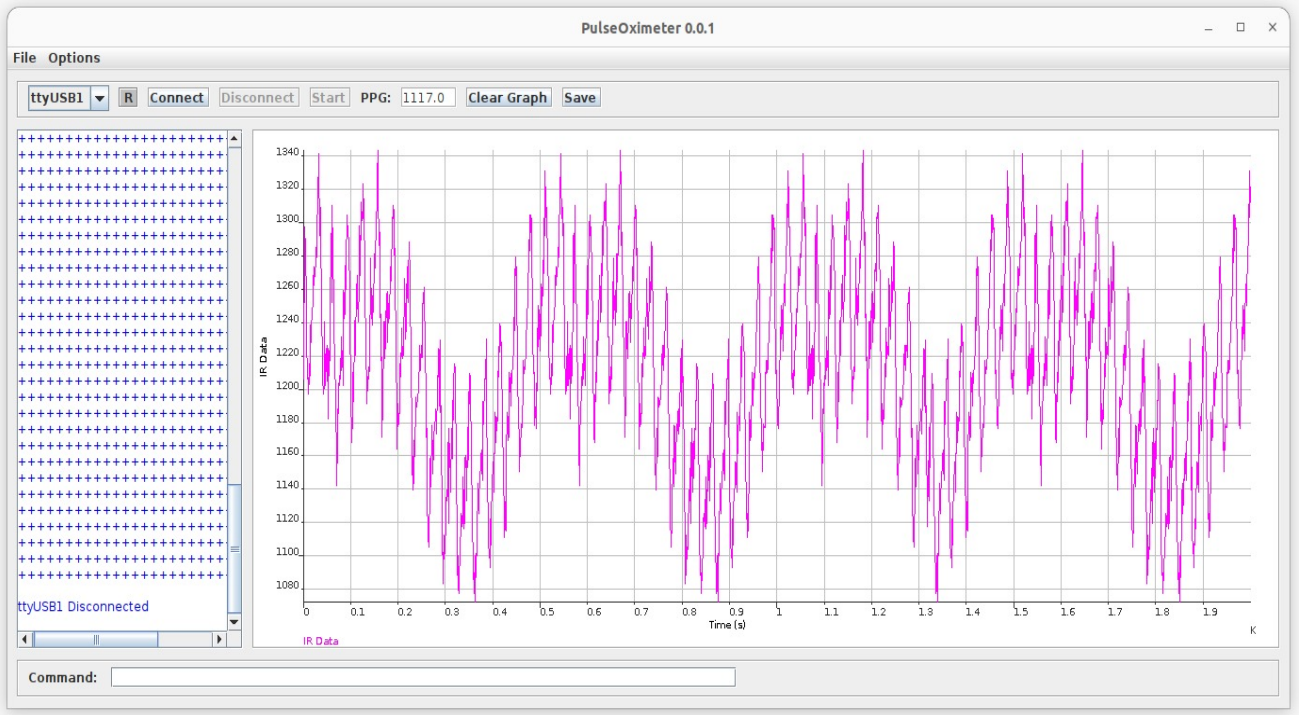


Figure 9.5: After Start Command

9.4.3 Save Button

On clicking "Save" button, the displayed graph will be saved as a "Graph.JPG" image and all the data packets of readings will be saved in a Text File named as "OutputFile" in the source folder where the application is running. A pop-up message and message on Scroll panel will be displayed if saving is successful. If a file of same name is already present in the source folder, it will overwrite it.

9.4.4 Clear Graph Button

On clicking "Clear Graph" button, the graph will be cleared and "Save Graph" button will be disabled.

9.4.5 Disconnect Button

On clicking "Disconnect" button, communication will be disconnected from the device and message will be displayed on Scroll panel. "Disconnect" button will be disabled and "Comm ports drop down, Refresh, Start" buttons will be enabled. The latest reading in Readings panel and graph will remain displayed on the GUI.

9.5 Working of GUI with Commands

GUI can also be operated by giving commands in the Command Text Field. On giving "help" command, application displays the list of commands available in the Scroll panel.

Command List:

1. "clc" – Clears the Scroll panel
2. "connect" – Connects the device selected on drop down menu, all actions according to "Connect" Button
3. "disconnect" – Disconnects the device, all actions according to "Disconnect" Button
4. "start" – Starts communication with device, all actions according to "Start" Button
5. "plot" – Plots the graph, all actions according to "Plot Graph" Button
6. "saveGr" – Saves the graph plotted on Graphs panel, all actions according to "Save" Button.
7. "saveFile" – Saves the readings received in a text file, all actions according to "Save" Button.
8. "clearGr" – Clears the plotted graph, all actions according to "Clear Graph"
9. "exit" – Disconnects the device and Exits the application

If any command other than the above-mentioned list is given to application, it will display error message in Scroll panel. If "start" command is given without successful connection, error message is displayed in scroll panel. If "plot" or "save" command is given before all values are received, error message is displayed in scroll panel. If "saveGr" or "saveFile" command is given before the graph is plotted, error message is displayed in scroll panel.

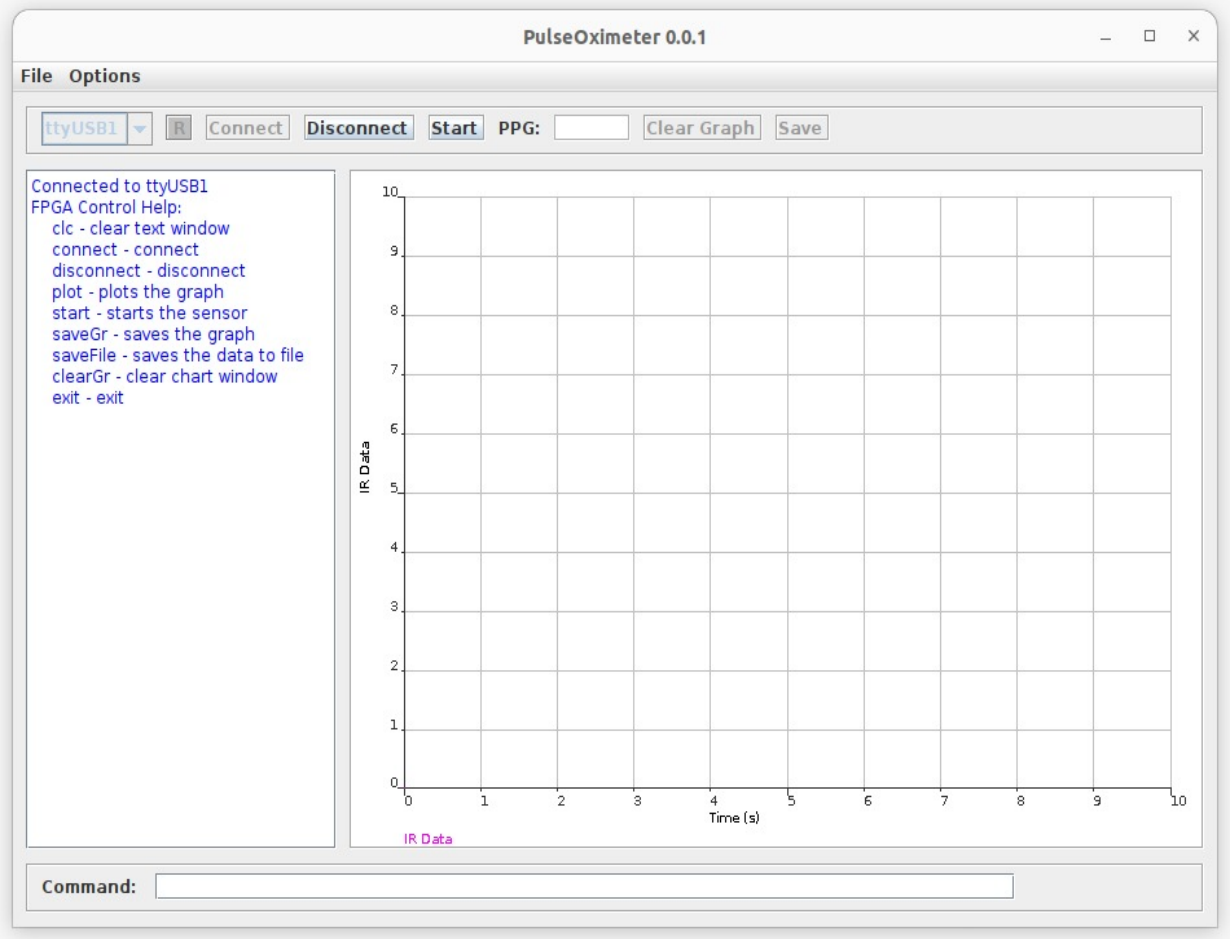


Figure 9.6: GUI Commands

9.6 Working of GUI with Menu Bar Options

There are two options on the Menu Bar: File => Save and Options => Exit. The Save option performs action similar to "Save" Button and Exit option performs action similar to "exit" command.

9.7 Disrupt Action

If the device is disconnected physically due to any reason, when the application is running connection will be broken and communication will be stopped. If disconnected while sending Data packets, readings displayed in Readings panel will stop.
The device needs to be reconnected physically and the application needs to be processed again from Connect.

9.8 Output Result

The final output with IR filtered data plotted is as follows:

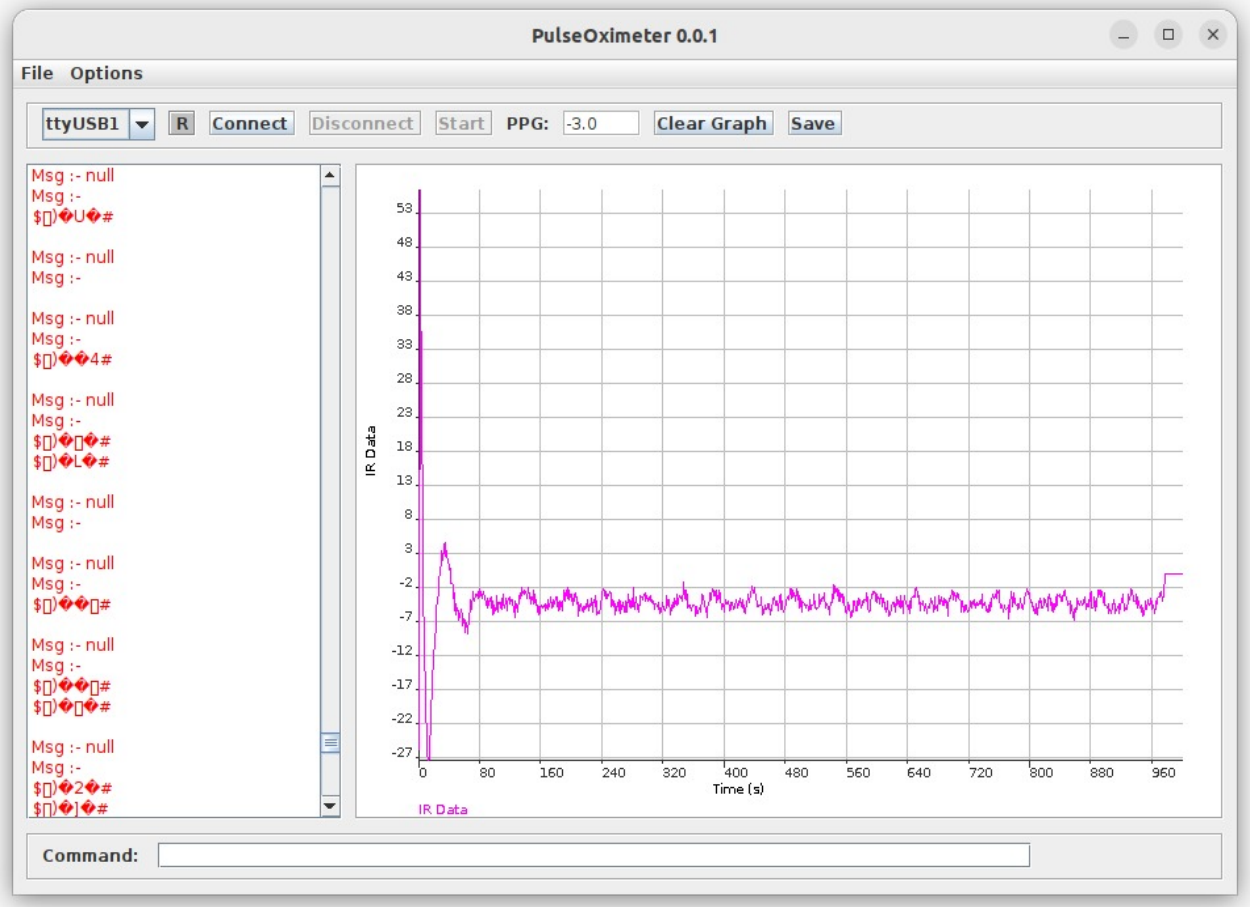


Figure 9.7: Final Result

10 Displaying Console Output in scroll panel in a Java GUI

You may use a JTextArea component to capture console output and add it to a JScrollPane component to show console text in a scroll panel in a Java GUI. This allows the user to scroll over the content.

The JTextArea component is commonly used to display multi-line text in a Java GUI. By default, the JTextArea does not provide any scrolling capabilities, so a JScrollPane is typically used to provide scrolling functionality.

10.1 Understanding Console Text and its Use.

Text that is output to a program's console or command line interface is referred to as console text, also known as console output.

Console text is frequently employed in programming for debugging purposes. The error message, for instance, may be printed to the console if a program finds a problem so the programmer may figure out what went wrong. To keep the user aware of what the program is doing, programs may also output status messages, progress updates, or other information to the console.
Using print(), output to the console can be produced. These techniques send text to the system's console or terminal window.

Here is an example of how to output console text:

```
1             printf ("FPGA_Start_Data Success \n")
```

This code will output the text `"FPGA_Start_Data Success"` to the console.

In a Java GUI application, you can capture console output and display it in a scroll panel

10.2 Displaying Console Output in a Java GUI

There are several situations when showing console output on a Java GUI can be helpful.

10.2.1 Debugging:

Console output is frequently employed during Java application development for debugging purposes. Developers may simply check the output and see any potential faults or problems by recording and presenting console output in a scroll panel within the GUI.

10.2.2 User opinions:

Some Java applications might carry out operations or processes that take a while to finish. Users can see real-time feedback about what the program is doing and how far along it is in the process by presenting console output in a scroll panel within the GUI. Insuring the user that the program is still active and hasn't frozen can assist.

10.2.3 Real-time observation:

In some circumstances, a Java program might be gathering information or keeping track of a system in real-time. Users can view real-time updates of the data being gathered or the condition of the system by showing console output in a scroll panel within the GUI.

In general, developers may find it simpler to debug their code, give users real-time feedback, and keep track of system activity by presenting console output in a scroll panel within a Java GUI.

10.3 Implementing Console Output in a Java GUI from an External Controller

You must set up a separate function that monitors the console output and updates the GUI as required if you want to show console output from an external controller in a Java GUI.

The general steps to apply this approach are as follows:

1. Create a new string function because text should be in string.
2. Use the **mPort.readBytes()** method to read the byte that represents the standard output of the external controller and store into a array.
3. Now covert this byte into string and updates the GUI as necessary. This can be done using a new **String ()** and store this string in a string variable.
4. Then print this string on scroll panel using **printTextWin** function.

Here is an example of how to implement this solution:

```
static String ReadString() {
    String recv_str;
    byte[] c_arr;
    int txsize;
    recv_str = null;
        try {
            if ((txsize = mPort.bytesAvailable())>0) {
                c_arr = new byte[txsize];
                mPort.readBytes(c_arr, txsize);
                    if(c_arr[0] != startByte) {
                        recv_str = new String(c_arr);
                        System.out.printf("\nReadString: %s",recv_str);
                    }
            }
        }
        catch(Exception ex){
            System.out.println(ex.toString());
        }
    return recv_str;
}
```

10.4 Console Output Printed on Scroll Panel

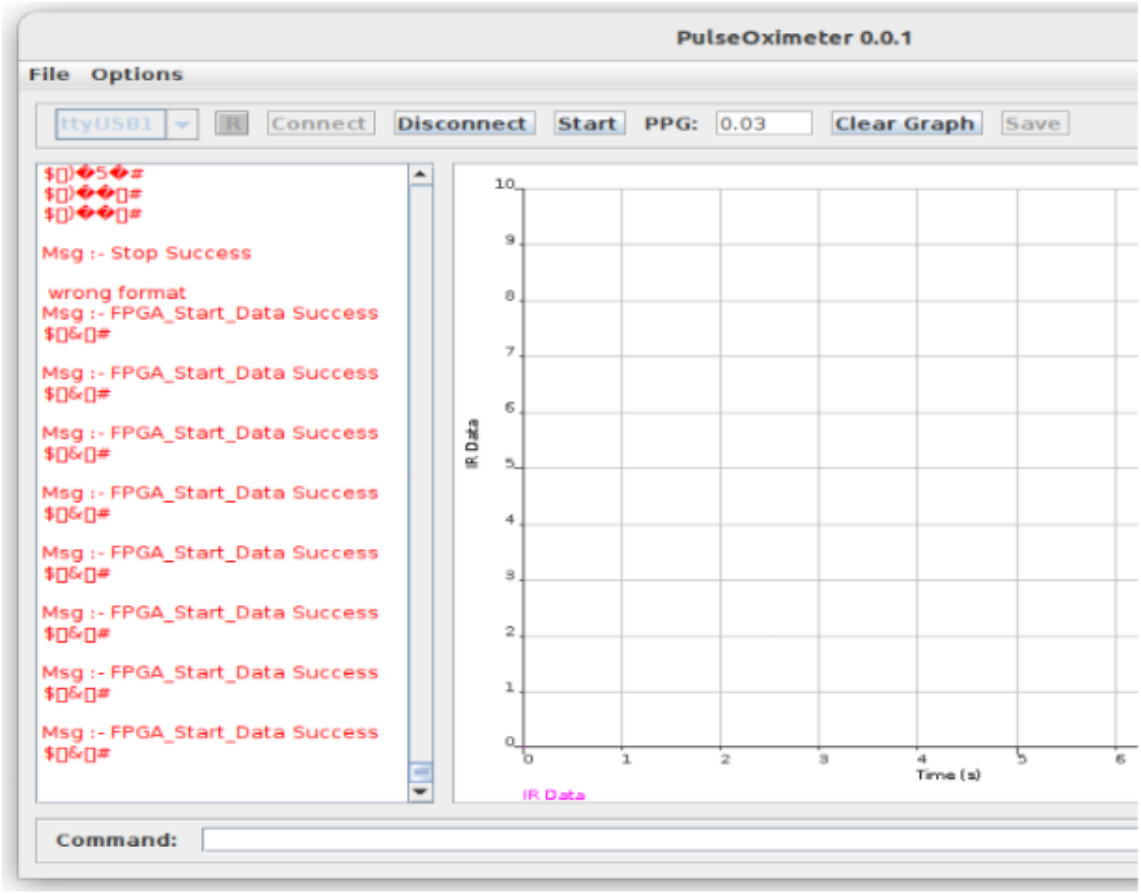


Figure 10.1: Console text on scroll panel

10.4.1 Challenges of Printing Console Text in Java GUI.

Printing a console text is a difficult process since it requires sending a send-packet to a controller each time to read the text, which is always impossible. Additionally, it frequently misinterprets the text data and sensor data and displays the sensor data as well. Additionally, when a controller receives data, another function—reading data from a sensor—prevents the controller from operating.

The sensor data, which is framed with `$` and `#` in the figure, is also visible, which is unacceptable.

10.4.2 Future Work / solution for Console Output Filtering in Java GUI Application

One solution to this problem is to implement a separate thread that listens for the console output from the external controller and filters out the sensor data. The thread can then update the GUI with the filtered console output.

Here are the general steps to implement this solution:

1. Create a separate thread that listens to the object.
2. Filters out the sensor data using regular expressions or other suitable methods. Most commanly read start and end byte and eliminate this data.
3. Within the thread, append each line of filtered console output to a JTextArea or another appropriate GUI component.

By filtering out the sensor data and displaying only the relevant console output in the GUI, the user can easily view and interpret the data without being distracted by irrelevant information.

11 Introduction to Charts in Java GUI

A chart is a graphical representation of data in Java GUI that enables users to quickly visualize and understand data. To make various charts, including line, bar, and pie charts, Java offers a number of frameworks and utilities.

11.1 Importance of Repainting Graphical Components in Java GUI Applications

The graphical elements in a Java GUI application, such charts, are typically implemented as instances of a subclass of the Component class, like JPanel or JChartPanel. The paint Component method on these components is in charge of painting the component on the screen.

A chart or any other graphical component has to be repainted for this reason: a number of variables could cause the component's look to alter. For instance, if the information on a chart changes

In conclusion, repeating of the chart (repainting/redrawing) is required in a Java GUI to ensure that the component's appearance is updated correctly upon initial display or when its state changes as a result of user input or other circumstances.

11.2 Repainting Graphical Components in Java GUI and output Pics

A graphical component must be repainted whenever it is first displayed on the screen, is resized, or is covered by another component. Calling the component's repaint method accomplishes this. The component will be scheduled for repainting using this method,

but it might not happen right away. We must immediately set it to **"0"** and leave the rest of the chart alone

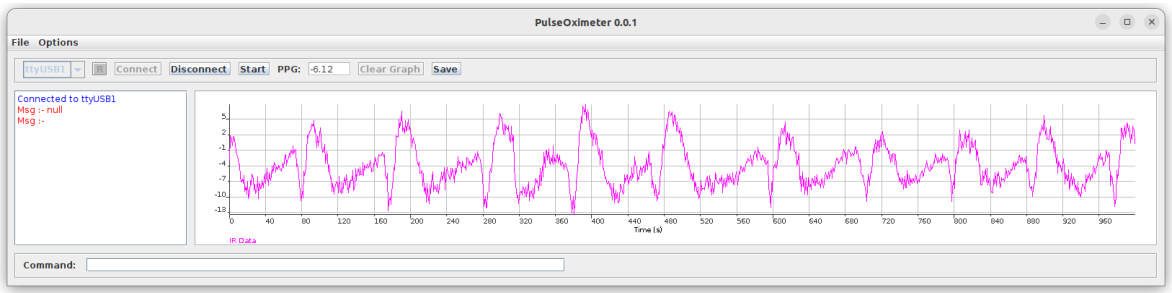


Figure 11.1: Graph on GUI-1

The illustration makes it clear that the chart should restart without eliminating the previous graphics when it reaches the end.

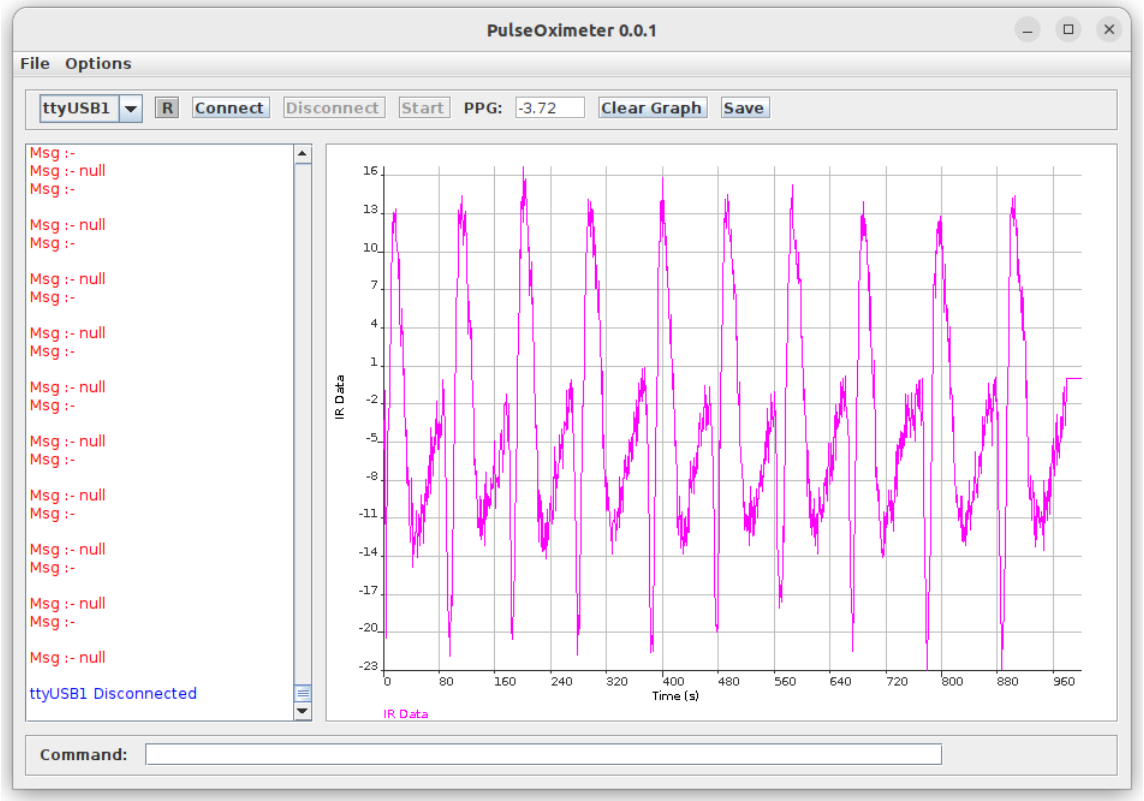


Figure 11.2: Graph on GUI-2

12 Stop Packet in FPGA-Java GUI Data Transmission

Introduction

Data from sensors is collected and processed using FPGA. The data is then transmitted to a Java GUI for analysis and display.

The two systems develop a communication mechanism to convey the sensor data from the FPGA to the Java GUI. The data format, data transmission rate, and other communication parameters are specified by this protocol.

A stop packet is delivered from the Java application to the FPGA to halt data transmission from the FPGA to the Java GUI. This stop packet is a message that is transmitted across a communication channel between the two systems and is prepared in accordance with the communication protocol. The FPGA stops delivering sensor data to the Java GUI when it gets the stop packet.

The stop packet may be initiated by a user action, such as pressing a button on the Java GUI. The Java program can then process and show the incoming sensor data after sending and receiving the stop packet. In order to terminate the transmission of sensor data to the GUI, the Java program sends a stop packet to the FPGA. It is a crucial part of the protocol used for communication between the Java program and the FPGA.

12.1 Sending Minimal Stop Packet for Efficient Communication.

Below is a comparison of actual send packet and stop-send packet

Sending a specific byte or series of bytes as a stop packet in some communication protocols is typical to signal that the data transmission should be stopped. To termi-

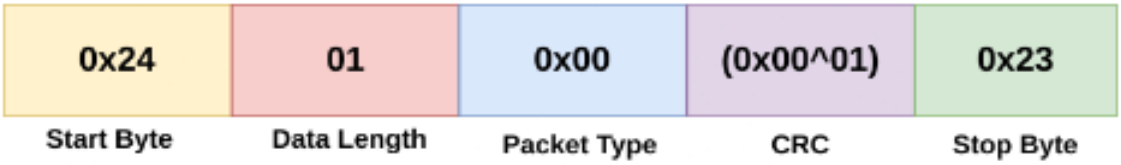


Figure 12.1: Send Packet

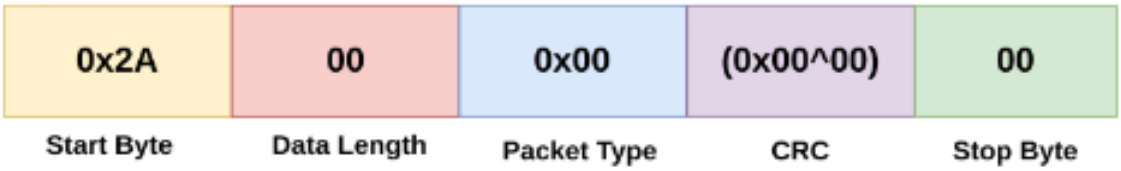


Figure 12.2: Send Packet to stop communiction

nate the transmission of sensor data, it might not be necessary to send a complete stop packet in the case of connection between the FPGA and Java GUI.

We can merely send a single byte, such as an asterisk `"*"`, to signal that the transmission should be halted in place of sending a complete stop packet. We can just send a string of zeros following the stop byte as the FPGA is not required to provide a confirmation message after receiving the stop packet. This can save bandwidth and simplify the implementation of the communication protocol.

13 References

1. Photoplethysmography signal processing and synthesis - Elisa Mejía-Mejía, John Allen, Karthik Budidha, Chadi El-Hajj, Panicos A. Kyriacou and Peter H. Charlton
2. Choosing FIR or IIR ? Understand design perspective – February 17, 2017 by Mathuranathan
3. Optimal filter bandwidth for pulse oximetry - October 2012, Norbert Stuban, Masatsugu Niwayama
4. Equalizing IIR filters for a constant group delay, July 2021, P.Trujillo
5. https://community.sw.siemens.com/s/article/introduction-to-filters-fir-versus-iir
6. Pulse oximetry, July 16, Amal Jubran
7. Design and development of pulse oximeter, August 2002, R.C. Gupta, S.S. Ahluwalia, S.S. Randhawa
8. Compact Pulse Oximeter Designed for Blood Oxygen Saturation and Heart Rate Monitoring, March 2022, Mohammod Abdul Motin, Partha Pratim Das, Chandan Kumar Karmakar, Marimuthu Palaniswami
9. Pulse oximetry: Its origin and development, October 1992, Takuo Aoyagi
10. Pulse oximetry: Understanding its basic principles facilitates appreciation of its limitations, March 2013, Edward D. Chan, Michael M. Chan, Mallory M. Chan
11. Analog vs. Digital Filtering of Data, Sabrina Miller
12. https://www.techopedia.com/definition/5435/graphical-user-interface-gui
13. https://fazecast.github.io/jSerialComm/
14. https://sourceforge.net/projects/jchart2d

15. EDK Concepts, Tools,and Techniques : Xilinx

16. Arty Z7 Reference Manual

17. Xilinx Zynq Datasheet

18. Xilinx Zynq Technical Reference Manual

19. Arty Z7 Revision D.0 Schematic

20. ARM Developer Suite Developer Guide

21. Cortex-A9 Technical Reference Manual

22. Design of Pulse Oximeters, J G Webster

23. Cheung P W, GauglitzK,Mason

24. Art of Designing Embedded Systems, Jack Gansle

25. Programming embedded systems in C and C++, Barr

26. Better Embedded System Software, Koopman

27. VHDL Script , Prof. Dr.-Ing. Kai Müller HS Brermerhaven

28. SoC Script , Prof. Dr.-Ing. Kai Müller HS Brermerhaven

29. Medical systems script , Prof. Dr.-Ing. Kai Müller HS Brermerhaven

30. The Designers Guide to VHDL, Peter J. Ashenden

31. Effective Coding with VHDL, Ricardo Jasinski

32. Circuit Design with VHDL, Volnei A. Pedroni

33. The Zynq Book: Embedded Processing Withe ARM® Cortex®-A9 on the Xilinx®

34. Zynq®-7000 All Programmable SoCMartin A. Enderwitz, Ross A. Elliot, Louise H Crockett, Robert W Stewart