University of Applied Sciences Bremerhaven
Department of Embedded System Design

Embedded System Master Project

submitted for the degree of

Master of Science

Pulse Oximeter

by

Dharmesh Patel Matriculation Nr.: 38073

Guided By : Prof. Dr.-Ing. Kai Mueller
Submission Date: June 16, 2021

Contents

List of Figures

Abstract

Measuring Oxygenation of blood (SaO2) plays a vital role in patient's health monitoring. This is often measured by pulse oximeter, which is standard measure during anesthesia, asthma, operative and post-operative recoveries. Despite all, monitoring Oxygen level is necessary for infants with respiratory problems, old people, and pregnant women and in other critical situations.

This project discusses the process of calculating the level of oxygen in blood and heart-rate detection using a non-invasive photo plethysmography also called as pulse oximeter designed using the Xilinx Zynq XC7Z020 FPGA and optical finger probe. The Serial Peripheral Interface is established between the ARM and the Digilent Pmod A/D & D/A converter. The probe uses infrared and red LED lights to measure and oxygen saturation in our body. The percentage of oxygen in the body is worked by measuring the intensity from each frequency of light after it transmits through the body and then calculating the ratio between these two intensities.

Chapter 1

Introduction

1.1 Background

In 1935, German physician Karl Matthes (1905–1962) developed the first two-wavelength ear O2 saturation meter with red and green filters (later red and infrared filters). His meter was the first device to measure O2 saturation [1]. Pulse oximetry was developed in 1972, by Takuo Aoyagi and Michio Kishi, bioengineers, at Nihon Kohden using the ratio of red to infrared light absorption of pulsating components at the measuring site. Susumu Nakajima, a surgeon, and his associates first tested the device in patients, reporting it in 1975 [2]. It was commercialized by Biox in 1980.

The human body requires and regulates with balance of oxygen in our blood. Oxygenation of Blood is often referred to Oxygen Saturation, which is the ratio of saturated hemoglobin to the total hemoglobin in the blood. Oxygenation occurs when an oxygen molecule enters the tissues of our body. Normal oxygen level in our body is 95% to 100%.If the oxygen level is measured to be below 90%, it creates a condition called Hypoxemia [3] i.e., low concentration of oxygen in blood. Oxygen levels below 80% will affect primary organ functions. Pulse oximetry is a non-invasive method to measure and monitor oxygen saturation.

1.2 Blood Oxygenation

Respiration is a vital process in human body. Body tissues require oxygen for aerobic respiration. Tissues gain all of their energy through respiration. The energy released through this process is used to store the synthesized adenosine triphosphate (ATP). This energy stored is responsible for distribution of oxygen to all the major parts of the body through our blood circulation flow.

Blood passes through pulmonary alveoli where carbon dioxide is released and the blood is oxygenated. This is called oxygenated process. The deoxygenated blood enters into the heart, it will be pumped out to lungs.

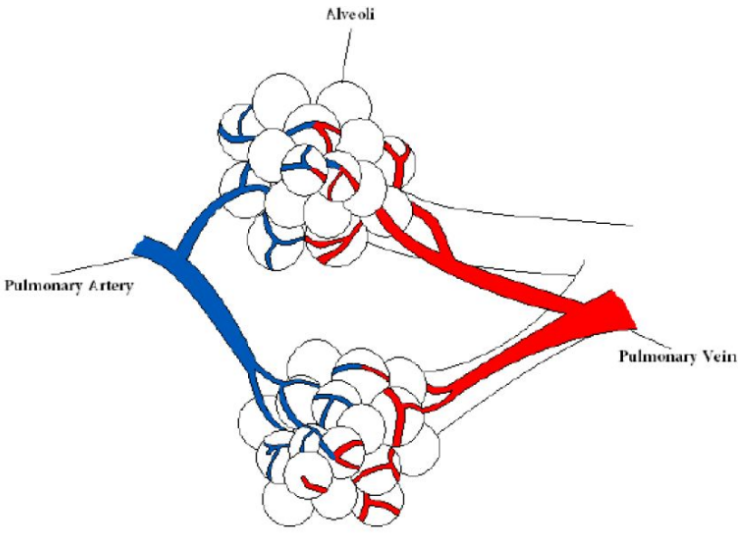


Figure 1.1: Pulmonary alveoli[(1)]

Hemoglobin, a protein in red cells gets attached to oxygen and forms oxyhemoglobin (HbO_2). When these red cells reach body tissues, it absorbs all the oxygen within and the red cells becomes deoxyhemoglobin (Hb). The block diagram below explains the process clearly. Deoxyhemoglobin once again reaches heart and the entire process is repeated and continued the same.

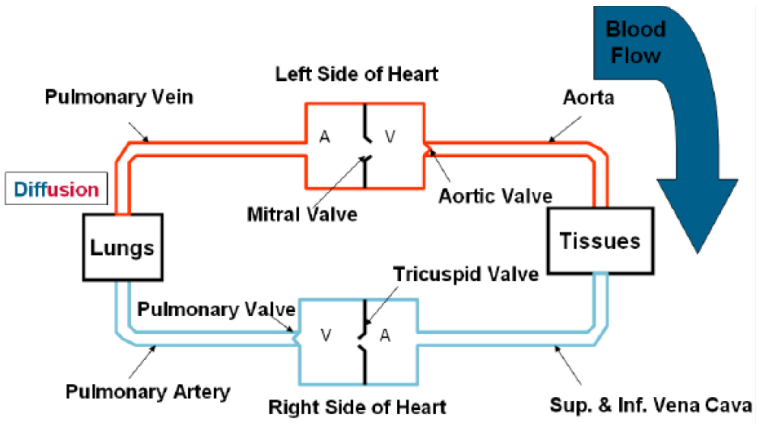


Figure 1.2: Blood Circulation[(1)]

1.3 Project Objective

The objective of the project is to design an electronic oxygen monitor using an embedded system. Xilinx Zynq XC7Z020 medium range FPGA board has been used along with PMOD connector which establish the interface between optical finger probe and FPGA. GUI has been design to show the SpO_2, Heart beat rate and plethysmograph.

1.4 Theory of Operation

Oxygen saturation is defined as the measurement of the amount of oxygen dissolved in blood, based on the detection of Hemoglobin and Deoxyhemoglobin. Two different light wavelengths are used to measure the actual difference in the absorption spectra of HbO_2 and Hb. The bloodstream is affected by the concentration of HbO_2 and Hb, and their absorption coefficients are measured using two wavelengths 660 nm (red light spectra) and 940 nm (infrared light spectra). Deoxygenated and oxygenated hemoglobin absorb different wavelengths. Deoxygenated hemoglobin (Hb) has a higher absorption at 660 nm and oxygenated hemoglobin (HbO_2) has a higher absorption at 940 nm.

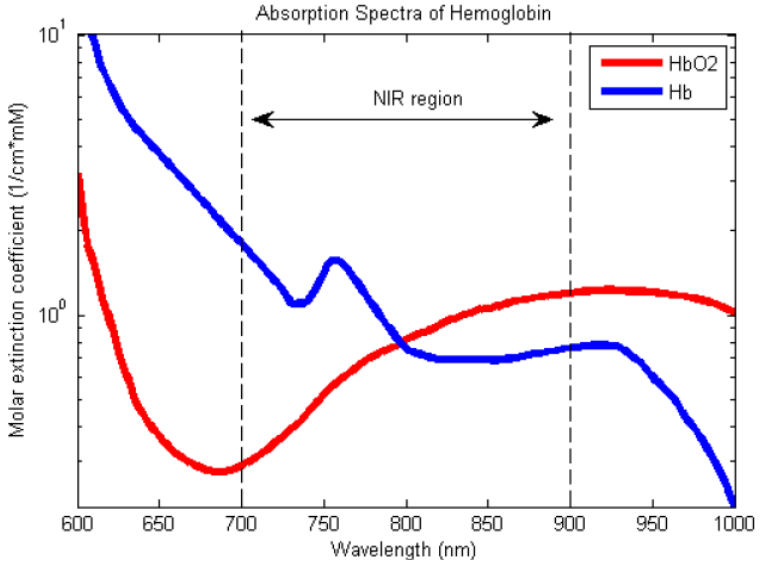


Figure 1.3: Light Absorption Spectra of Himoglobin[(2)]

A photodetector in the sensor perceives the non-absorbed light from the LEDs. This signal is inverted using an inverting operational amplifier (OpAmp) and the result is a signal like the one in Figure 4 [4]. This signal represents the light that has been absorbed by the finger and is divided in a DC component and an AC component. The DC component represents the light absorption of the tissue, venous blood, and non-pulsatile arterial blood. The AC component represents the pulsatile arterial blood.

The pulse oximeter analyzes the light absorption of two wavelengths from the pulsatile-added volume of oxygenated arterial blood (AC/DC) and calculates the absorption ratio using the following equation.

$$\frac{AC_{660}/DC_{660}}{AC_{940}/DC_{940}}$$

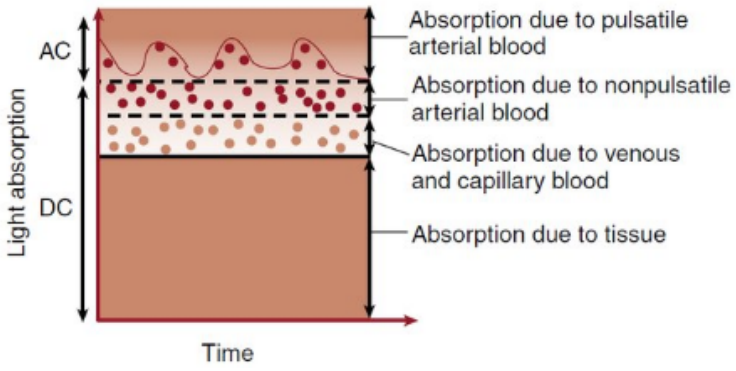


Figure 1.4: Light Absorption Diagram[3]

SpO_2 is taken out from a table stored on the memory calculated with empirical formulas. A ratio of 1 represents a SpO_2 of 85%, a ratio of 0.4 represents SpO2 of 100%, and a ratio of 3.4 represents SpO_2 of 0%. For more reliability, the table must be based on experimental measurements of healthy patients. R curve shows in figure 5 [4].

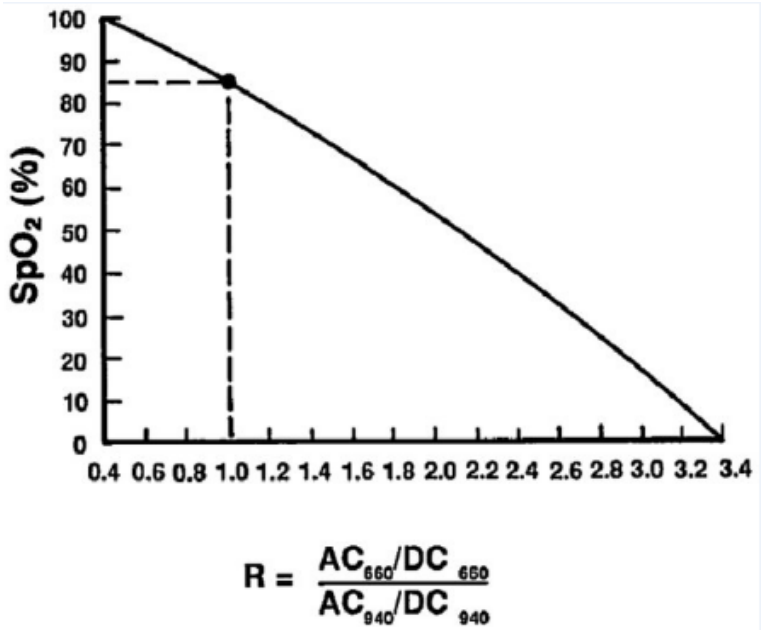


Figure 1.5: R-Curve[3]

1.5 Photoplethysomography

Pulse oximeters often show the pulsatile change in absorbance in a graphical form. This is called the “plethysmographic trace ” or more conveniently, as “pleth”. The pleth is an extremely important graph to see. It tells you how good the pulsatile signal is. If the quality of the pulsatile signal is poor, then the calculation of the oxygen saturation may be wrong. The

pulse oximeter uses very complicated calculations to work out oxygen saturation. A poor pleth tracing can easily fool the computer into wrongly calculating the oxygen saturation. As human beings, we like to believe what is good, so when we see a nice saturation like 99% , we tend to believe it, when actually the patients actual saturation may be much lower. So always look at pleth first, before looking at oxygen saturation.

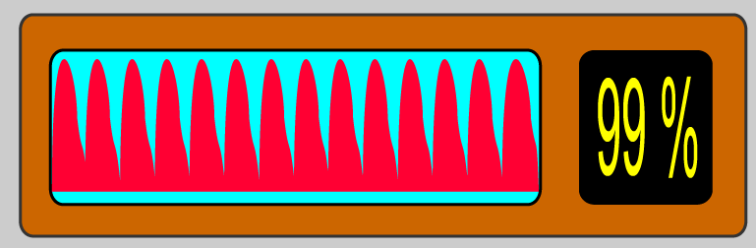


Figure 1.6: Plethysmographic trace[4]

1.6 Limitations

Pulse oximeters rarely cause any harm directly, though apparently some older models could cause burns and there are reports of the probes causing pressure ulcers. However if their limitations are not borne in mind harm could be caused by someone having the wrong (or no) treatment. So it's important to know what the limitations are.

One limitation is that pulse oximeters cannot operate reliably with a poor signal. This has been referred to as a 'safe' limitation, in that the pulse oximeter is not able to give an accurate reading but in some way indicates this fact. Obviously, pulse oximeters need to be technically capable of indicating this and the person using it must be aware of this point.

The other sort of limitation is more dangerous in that the pulse oximeter may appear to have a good signal and be displaying a saturation figure, but either the figure is inaccurate or gives a false sense of security.

1.6.1 Poor signal

Pulse oximeters need a strong regular pulse in the finger (or ear etc) that the probe is on.

A common problem is that people can have cold hands and feet, and have only a very weak pulse. In this case a pulse oximeter may display a reading but it might not be accurate. Some pulse oximeters have a means of indicating how strong the signal is they are receiving and it is important to check this. A still weaker signal may mean the pulse oximeter is not able to work at all.

An irregular signal can also cause problems for a pulse oximeter trying to determine oxygen saturation. This can be caused by an irregular heartbeat or by the patient moving, shivering or fitting.
Poor positioning of the probe can cause inaccurate readings due to various problems. This can be a particular problem with very small fingers and very large ones. Make sure the probe is well on the finger.

1.6.2 Carbon dioxide

A pulse oximeter can cause a false sense of security by giving a good saturation figure when someone's breathing is completely inadequate. This is especially true when a patient is getting supplementary oxygen.
There are two main functions of breathing, one is getting oxygen out of the air and into the body, the other is getting carbon dioxide out of the body and into the air. It possible for someone to be getting enough oxygen into their body but not be getting rid of enough carbon dioxide.
Oxygen saturation by itself does not tell the whole story about breathing - this is especially true if someone is being given oxygen. As a minimum it is also necessary to record the respiratory rate, and if they are having oxygen, how much they are having.

1.6.3 Delays

There will be a delay between an event such as a patient taking off an oxygen mask and the subsequently less oxygenated blood passing through the finger the probe is on. It has been reported that there will be a longer delay if the pulse oximeter probe is attached to a toe compared with a finger or ear.
Another point is that pulse oximeters average signals over a period of time, this will cause a delay in giving a new (real) oxygen saturation.

1.6.4 Carbon monoxide

Carbon monoxide is a colourless, odorless gas that is produced in most fires. Breathing in carbon monoxide will lead to it becoming attached to hemoglobin in preference to oxygen, so it is only necessary to breathe in a small amount of carbon monoxide to have a large amount of hemoglobin taken up by it and therefore not available to carry oxygen. For instance, 25%

of someone's hemoglobin is taken up by carbon monoxide then only 75% is available to carry oxygen, and so their oxygen saturation could, at best, be only 75%.

Pulse oximeters will display an oxygen saturation which is approximately equal to the percentage of hemoglobin combined with oxygen plus the percentage of hemoglobin combined with carbon monoxide. So if someone has 25% of their hemoglobin saturated with carbon monoxide and a true oxygen saturation of 70% a pulse oximeter will display an oxygen saturation of about 95%. This is obviously extremely dangerous and for this reason, pulse oximeters should not be used with people who may have inhaled smoke, ie anyone who has been involved with any sort of fire, unless you are certain that they do not have any significant level of carbon monoxide in their blood.

1.6.5 Methemoglobin

Methaemoglobin is an abnormal type of hemoglobin that does not bind oxygen well. Normally 1-2% of people's hemoglobin is methemoglobin, a higher percentage than this can be genetic or caused by exposure to various chemicals and depending on the level can cause health problems. A higher level of methemoglobin will tend to cause a pulse oximeter to read closer to 85% regardless of the true level of oxygen saturation.

Chapter 2

Hardware Interface

2.1 Interface Design

2.1.1 Interface Block Diagram

A block diagram of the interface design shown in below figure[7].

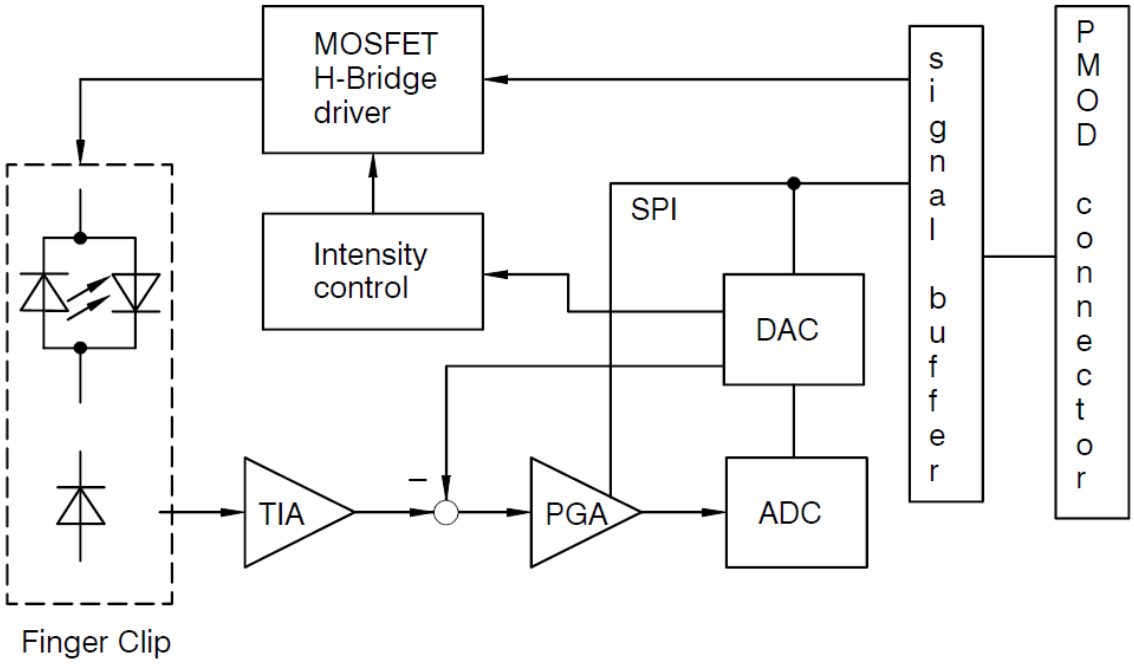


Figure 2.1: Interface Block Diagram[(5)]

2.1.2 Finger Probe

The finger clip consists of two LEDs and one pin diode. A red LED with a wavelength of 660 nm and an infra red LED with a wavelength of 895 nm has attached in parallel with opposite direction so that both LEDs will never turn on at the same time. A photo diode is used to capture light intensity from the LEDs.

Figure 2.2: Finger probe[5]

2.1.3 PMOD Connector

Pmod™ devices are Digilent's line of small I/O interface boards that offer an ideal way to extend the capabilities of programmable logic and embedded control boards. They allow sensitive signal conditioning circuits and high-power drive circuits to be placed where they are most effective - near sensors and actuators.

Pmod modules communicate with system boards using 6, 8, or 12-pin connectors that can carry multiple digital control signals, including SPI and other serial protocols. Pmod modules allow for more effective designs by routing analog signals and power supplies only where they are needed, and away from digital controller boards.

In this project 12-pin PMOD connector has been used which is carry SPI protocol.[7]

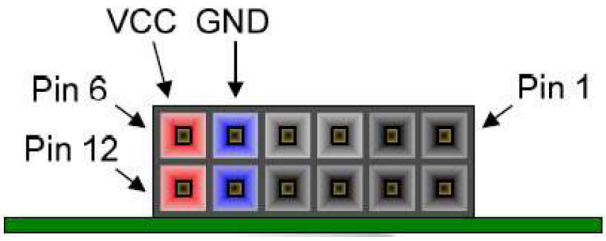


Figure 2.3: PMOD Connector (front view of FPGA board)[5]

PIN No.	PIN Name	Description
1	sclk (Input)	SPI system clock
2	adccs (Input)	Chip select/sync for ADC - Active High
3	redon (Input)	red led on
4	iron (Input)	infrared led on
5	GND (power)	Ground
6	Vss (power)	3.3 V power supply
7	pgacs (Input)	Chip select/sync for PGA - Active High
8	daccs (Input)	Chip select/sync for DAC - Active High
9	mosi (Input)	SPI sdi (sdo for FPGA)
10	miso (Input)	SPI sdo (sdi for FPGA)
11	GND (power)	Ground
12	Vss (power)	3.3 V power supply

Figure 2.4: PMOD PINs

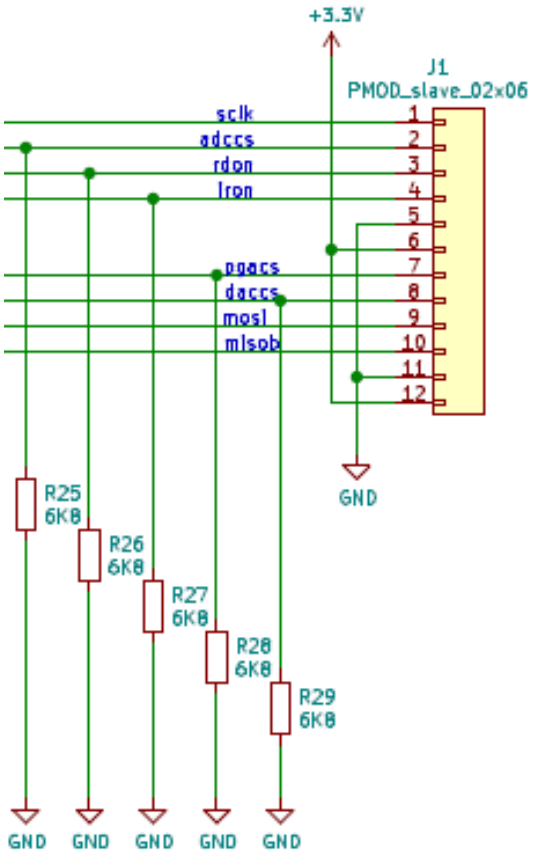


Figure 2.5: PMOD Connector[5]

2.1.4 MOSFET H-bridge with Intensity control

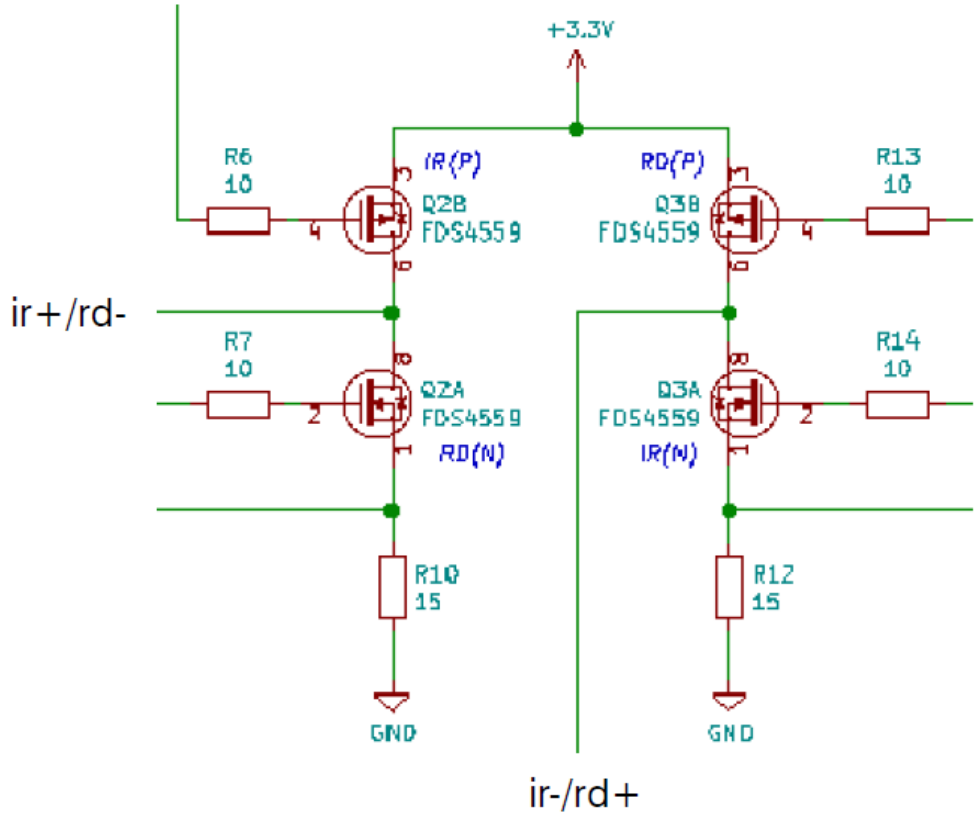


Figure 2.6: MOSFET H-bridge circuit[5]

The H-bridge provides the output for the RED and IR LEDs. As shown in figure leds have only two terminals that connected anti-parrallel. Q2B(p channel) and Q3A(n channel) are responsible for the IR led. Q3B(p channel) and Q2A(n channel) are responsible for the RED led.

2.1.5 DAC and anti saturation circuit

The DAC provided for outputs VOUTA-VOUTD:

1	VOUTA	Red light intensity
2	VOUTB	Infrared light intensity
3	VOUTC	Variable light intensity pulse indicator(LED)
4	VOUTD	Ambient light cancelling

All outputs of the DAC are intended to be changed only if necessary, i.e. if ambient light condition change. The LEDs should not be switched on and off by DAC. This is achieved by digital signals RED and IR in PMOD.

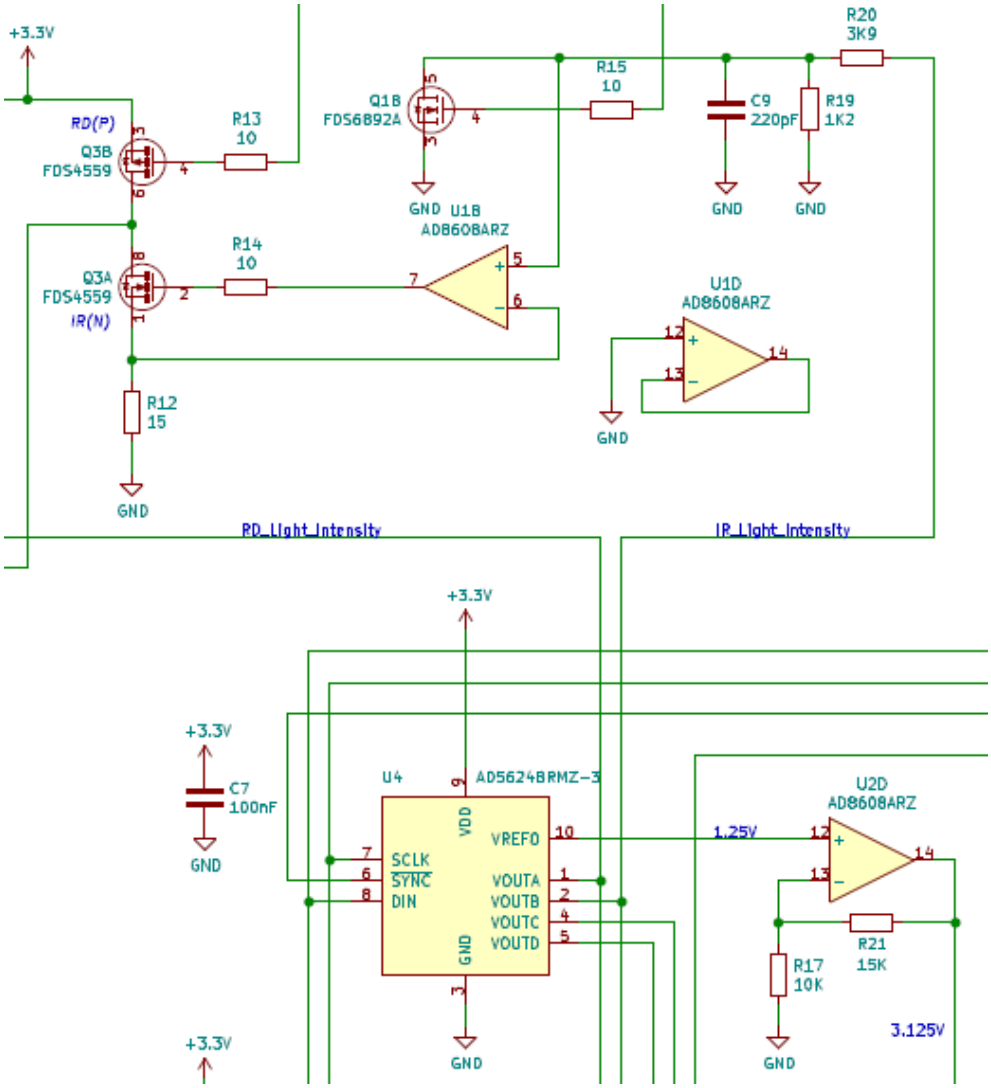


Figure 2.7: DAC and anti saturation circuit with voltage driver[5]

The DAC provides an output between 0 to 2.5 V. The voltage divider given by R19 and R20(see the figure) reduce the maximum output to

$$\frac{R_{19}}{R_{19}+R_{20}} \times 2.5V = 0.5882V \tag{2.1}$$

The MOSFET Q3NA sets the output to zero volts when IR led is off. This prevents saturation for the operational amplifier IC3B. The DAC output can remains constant due to resister R20.

2.1.6 Trans Impedance Amplifier(TIA)

Current to Voltage conversion is called TIA(trans impedance amplifier). Red and IR light causes a small current i_s in reverse direction of the pin diode of the finger clip. This current is converted into a voltage at the output of U2A.

The next OP U2B subtracts voltage VOUTD from the DAC to cancel ambient light. R9 could

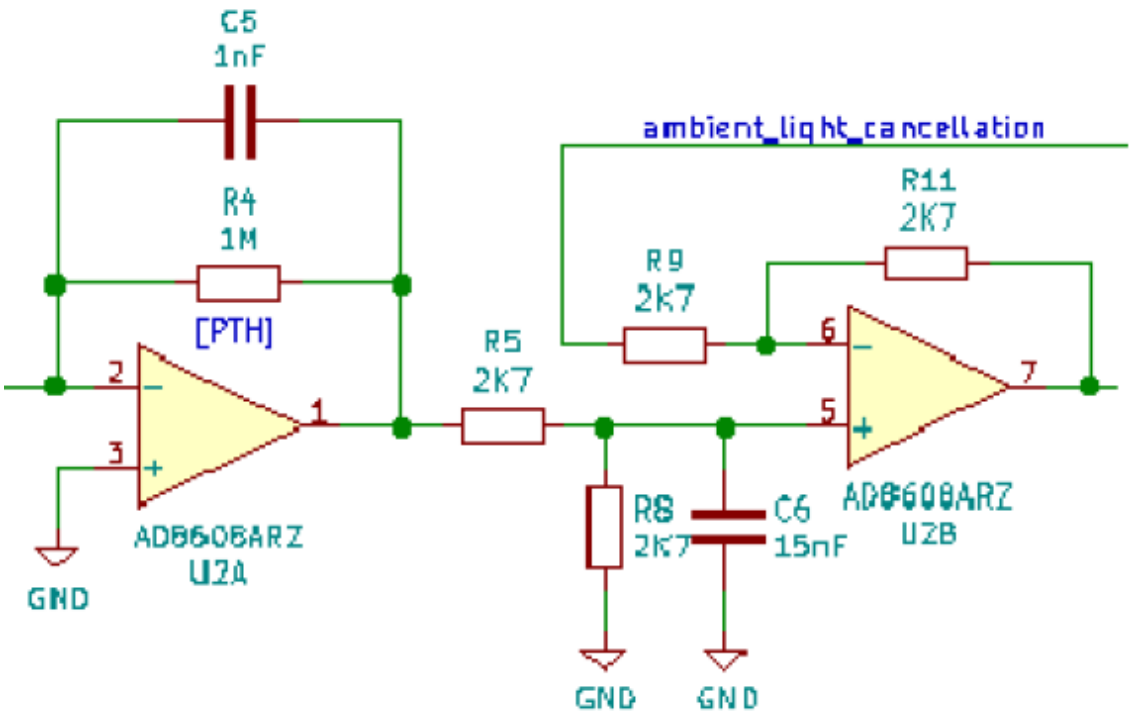


Figure 2.8: TIA (subtractor)[5]

be increased to decrease the gain for DAC voltage. Since all resistors R5,R8,R9 and R11 are equal the gain of the instrumentation amplifier is equal to one.
The components R5,R8 and C6 from a passive low pass according to

$$H(j\omega) = \frac{R_8}{R_5 + R_8} \times \frac{1}{1 + j\omega \frac{R_5 R_8}{R_5 + R_8} C_6} \tag{2.2}$$

this acts as an anti-aliasing filter for the DAC. The cut-off frequency can easily made smaller with greater values for C_6.

2.1.7 Analog Test PIN points

The purpose of providing analog test PINs on board is that to measure the outputs from the DAC and inputs of the PGA and ADC. As we are aware that the LEDs of the finger clip are very hard to observed with naked eyes. So, in order to check whether our DAC really produced output voltage accordance with the provided input intensity its very important to have such test pins available in board which you can measure using an oscilloscope or ammeter.

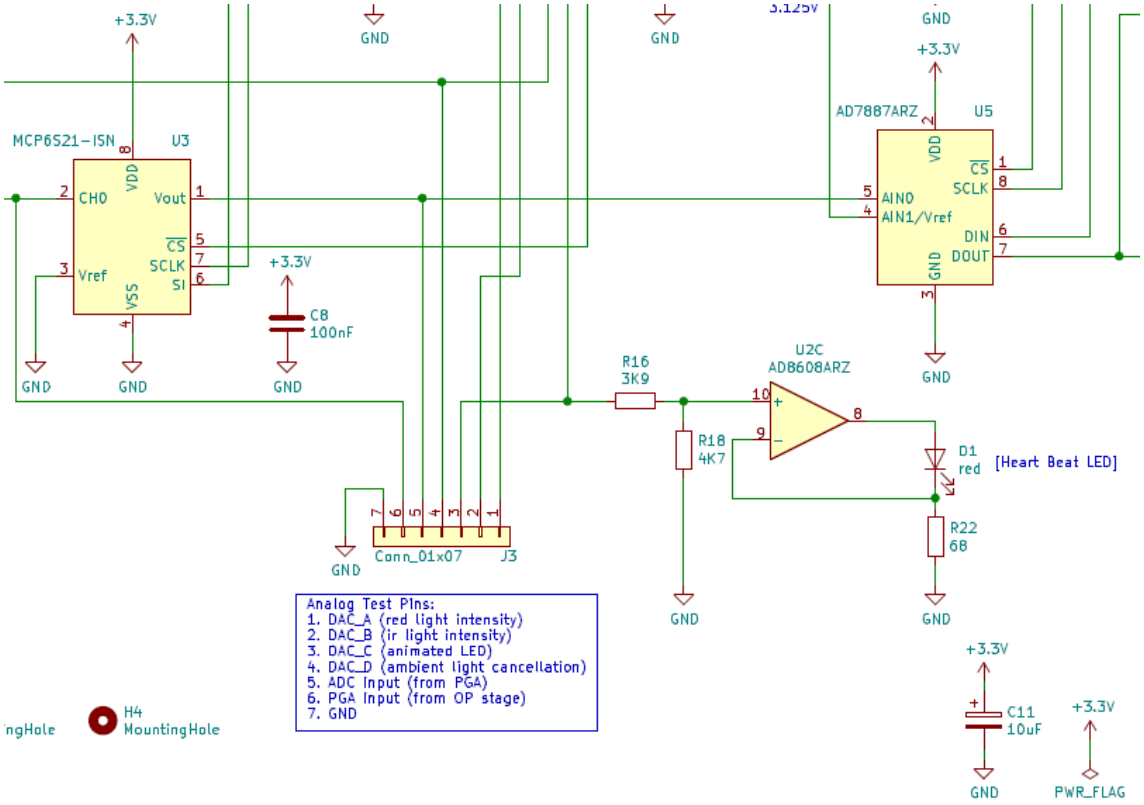


Figure 2.9: Analog test PINs[5]

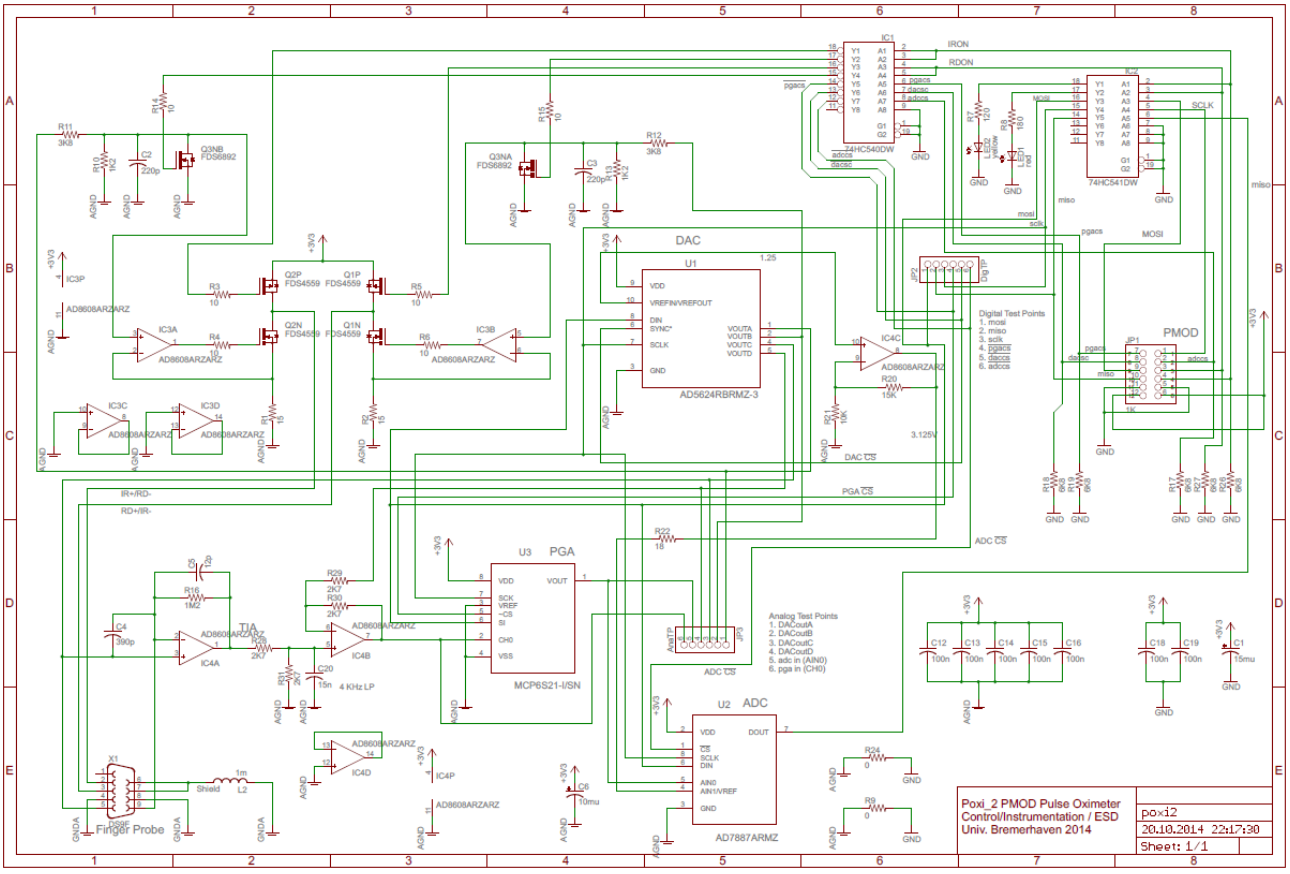


Figure 2.10: POXI Interface board Schematic[6]

2.1.8 Pulse Oximeter Interface board Diagram

2.1.9 Digital-to-Analog Convertor(DAC)

DAC AD5624R[8] has been used in this application. DAC mainly used here for two purposes, one is to increase light intensities using its four output channels and second is to provide internal reference voltage to ADC.
In order to work with DAC, proper data bits should be send to DAC by FPGA.

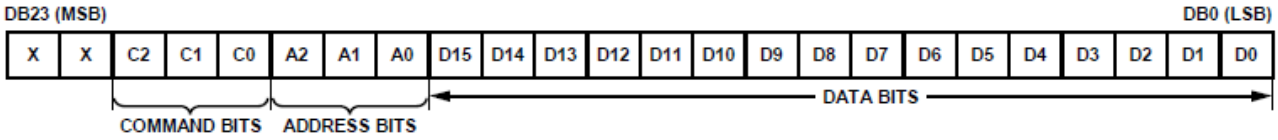


Figure 2.11: DAC Input Shift Register

The following table shows the list of operation and input shift register bits to be send. Detailed information can be found in the AD5624R data sheet(see the reference).

Sr.No.	Operation	Input Shift register Bits
1	Reset DAC	0x280000
2	Power-Up DAC	0x20000F
3	LDAC Register Mode	0x300000
4	Internal Reference	0x380001

Table 2.1: DAC Input bits

2.1.10 Analog-to-Digital Convertor(ADC)

ADC AD7887[9] has been used in this application. ADC will provide the measured output of the FPGA. In order to work with ADC following control bit has to be send to ADC. The control register on the AD7887 is an 8-bit, write-only register.

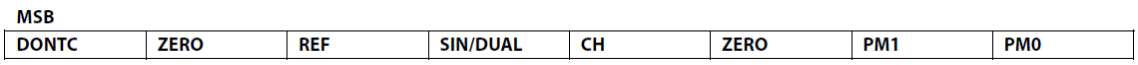


Figure 2.12: ADC Control Register

Operation	Control register Bits
ADC setup	0x0100

Table 2.2: ADC control bits

Detailed information can be found in AD7887 data sheet(see the reference).

2.1.11 Programmable Gain Amplifier(PGA)

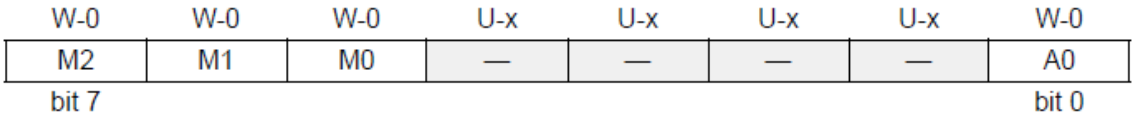


Figure 2.13: PGA Instruction Register

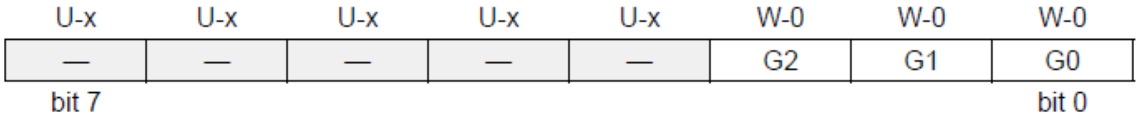


Figure 2.14: PGA Gain Register

Operation	**Register Bits**
PGA setup and Gain	0x4001

Table 2.3: PGA gain bits

Detailed information can be found in MCP6S21/2/6/8 data sheet(see the reference).

2.2 SPI Protocol

Communication between electronic devices is like communication between humans. Both sides need to speak the same language. In electronics, these languages are called communication protocols. SPI (serial peripheral interface) is quite a bit slower than protocols like USB, ethernet, Bluetooth, and WiFi, but they're a lot more simple and use less hardware and system resources. SPI is a common communication protocol used by many different devices. For example, SD card modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers. One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream.

Devices communicating via SPI are in a master-slave relationship. The master is the controlling device (usually a microcontroller), while the slave (usually a sensor, display, or memory chip) takes instruction from the master. The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave.[5]

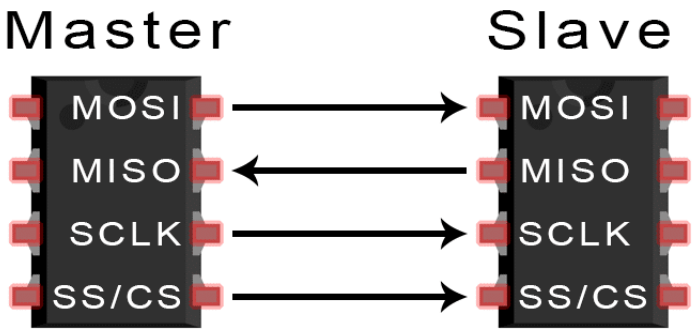


Figure 2.15: SPI master and slave interface(7)

MOSI (Master Output/Slave Input) - Line for the master to send data to the slave.
MISO (Master Input/Slave Output) - Line for the slave to send data to the master.
SCLK (Clock) – Line for the clock signal.
SS/CS (Slave Select/Chip Select) – Line for the master to select which slave to send data.

*In practice, the number of slaves is limited by the load capacitance of the system, which reduces the ability of the master to accurately switch between voltage levels.

2.2.1 HOW SPI WORKS

THE CLOCK

Wires Used	4
Maximum Speed	Up to 10 Mbps
Synchronous or Asynchronous?	Synchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	Theoretically unlimited*

Figure 2.16: SPI Specifications

The clock signal synchronizes the output of data bits from the master to the sampling of bits by the slave. One bit of data is transferred in each clock cycle, so the speed of data transfer is determined by the frequency of the clock signal. SPI communication is always initiated by the master since the master configures and generates the clock signal.Any communication protocol where devices share a clock signal is known as synchronous. SPI is a synchronous communication protocol.

The clock signal in SPI can be modified using the properties of clock polarity and clock phase. These two properties work together to define when the bits are output and when they are sampled. Clock polarity can be set by the master to allow for bits to be output and sampled on either the rising or falling edge of the clock cycle. Clock phase can be set for output and sampling to occur on either the first edge or second edge of the clock cycle, regardless of whether it is rising or falling.[5]

Clock polarity and phase

In addition to setting the clock frequency, the master must also configure the clock polarity and phase with respect to the data.The timing diagram is shown in below figure. The timing is further described below and applies to both the master and the slave device.

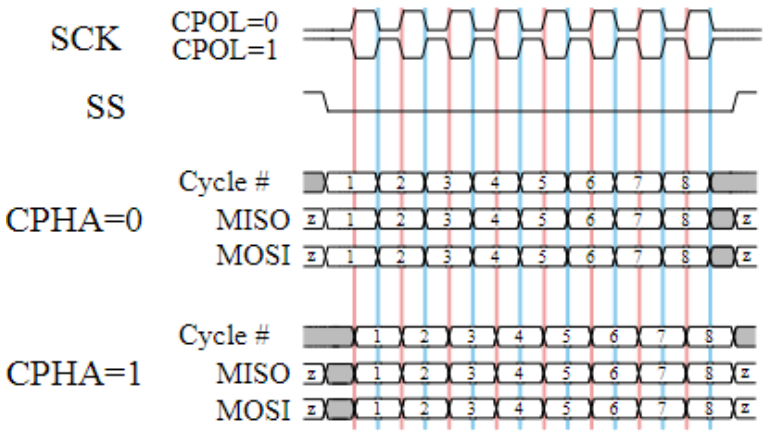


Figure 2.17: SPI Mode timing diagram[8]

There are four possible modes that can be used in an SPI protocol[6]:

SPI mode	Clock polarity (CPOL/CKP)	Clock phase (CPHA)
0	0	0
1	0	1
2	1	0
3	1	1

Figure 2.18: SPI Mode list

1. For CPOL=0,the base value of the clock is zero.For CPHA=0, data are captured on the clock's rising edge and data are propagated on a falling-edge.
2. For CPOL=0,the base value of the clock is zero.For CPHA=1,data are captured on the clock's falling edge and data are propagated on a rising edge.
3. For CPOL=1,the base value of the clock is one.For CPHA=0,data are captured on the clock's rising edge and data are propagated on a falling-edge.
4. For CPOL=1,the base value of the clock is one.For CPHA=1,data are captured on the clock's falling edge and data are propagated on a rising-edge.

SLAVE SELECT

The master can choose which slave it wants to talk to by setting the slave's CS/SS line to a low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple CS/SS pins may be available on the master, which allows for multiple slaves to be wired in parallel. If only one CS/SS pin is present, multiple slaves can be wired to the master by daisy-chaining.

MOSI AND MISO

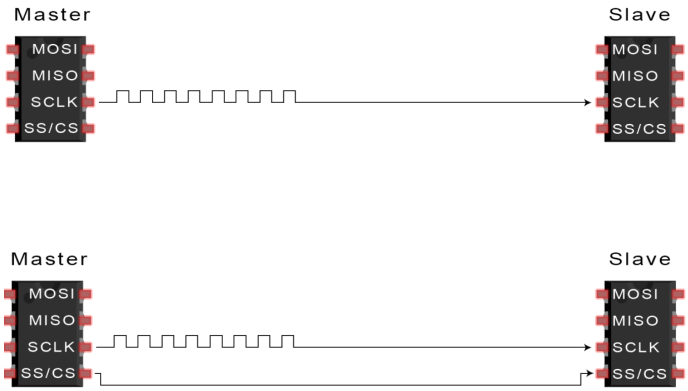
The master sends data to the slave bit by bit, in serial through the MOSI line. The slave receives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.

The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first[5].

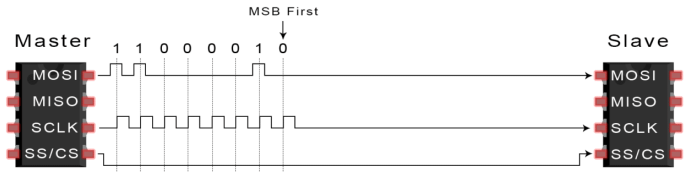
STEPS OF SPI DATA TRANSMISSION

1. The master outputs the clock signal:

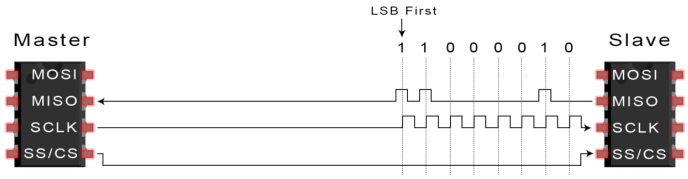
2. The master switches the SS/CS pin to a low voltage state, which activates the slave:



3. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:



4. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:



Now we will utilize SPI protocol for communication using FSM (finite state machine) approch. The VHDL code is written for this purpose in vivado tool (Refere Appendix A).
A state machine is a modeling design technique for sequential circuits. At any time, the machine sits in one of a finite number of possible states. For each state, both the output values and the transition conditions into other states are fully defined. The state is stored by the FSM, and the transition conditions are usually reevaluated at every (positive) clock edge, so the state-change procedure is always synchronous because the machine can only move to another state when the clock ticks.
The FSM model provides a systematic approach (a method) for designing sequential circuits, which can lead to optimal or near-optimal implementations. Moreover, the method does not require any prior knowledge or specifics on how the general circuit (solution) for the problem at hand should look like.

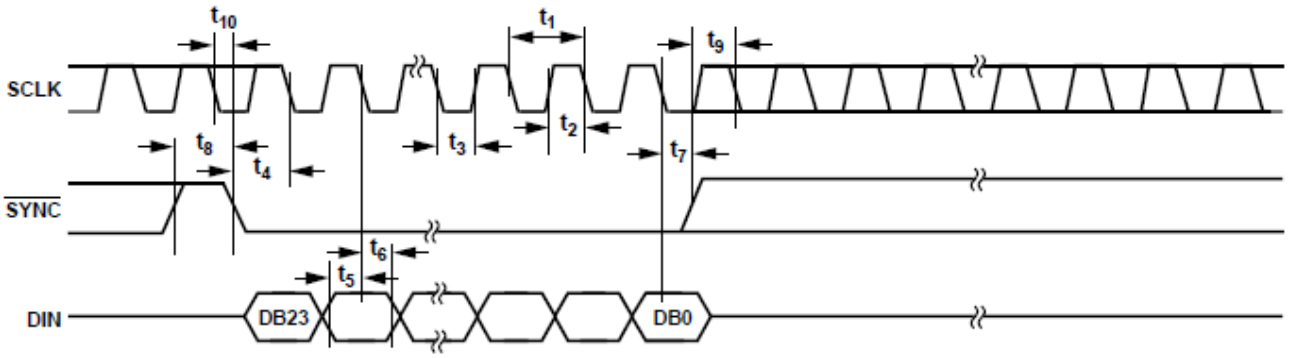


Figure 2.19: DAC SPI write cycle[9]

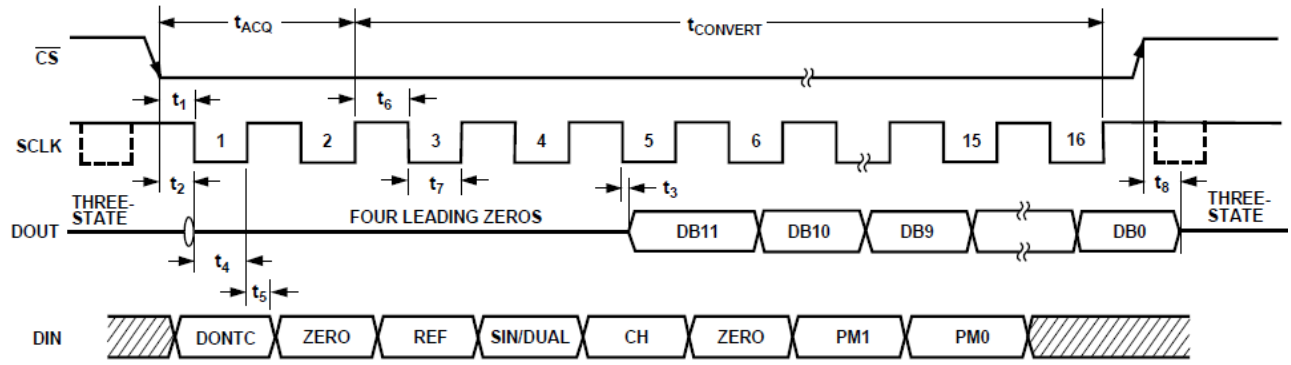


Figure 2.20: ADC SPI read/write cycle[10]

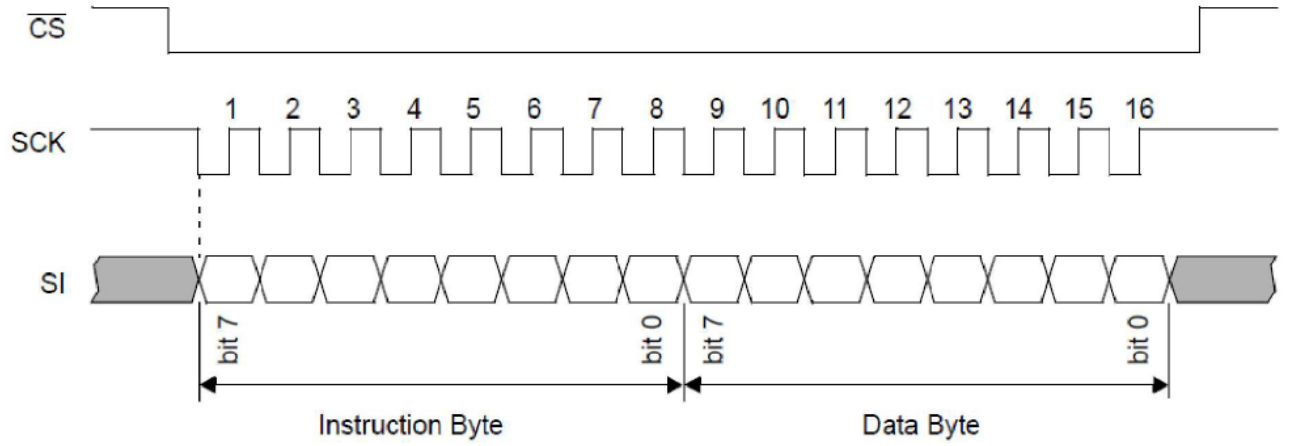


Figure 2.21: PGA SPI write cycle[11]

Device	Data Length(bits)	CPOL	CPHA
DAC	24	1	0
PGA	16	1	1
ADC	16	1	0

Table 2.4: SPI Parameters

Chapter 3

Pulse Oximeter SoC design

The Pulse Oximeter SoC design is basically built up by peripheral interface, an AXI bus, and a processing system.

The peripheral interfaces were implemented by creating custom IP. In the custom IPs contain an instance of the universal reconfigurable SPI, and additional logic for specific purposes.

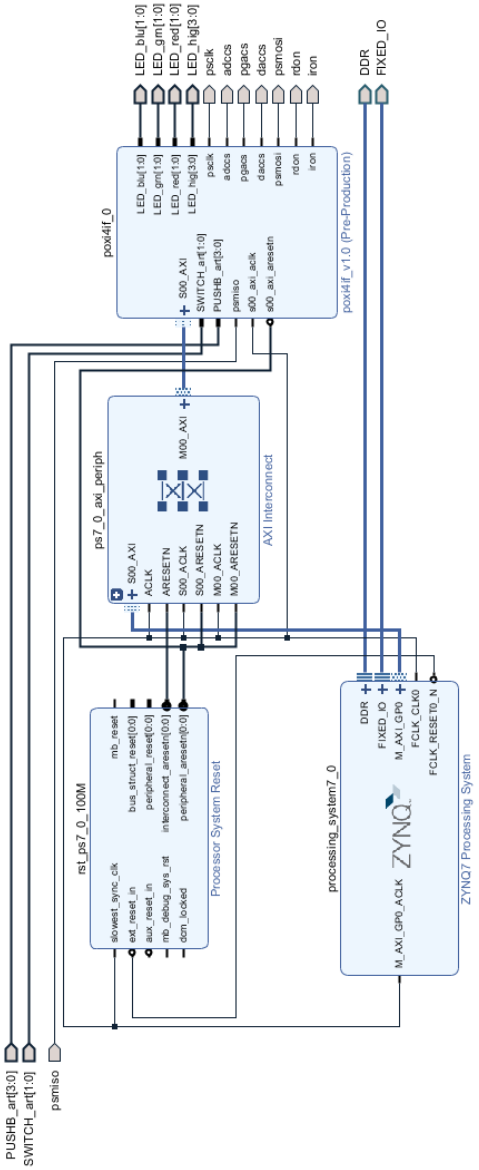


Figure 3.1: POXI SoC Design

3.1 POXI Custom IP Component

The poxi4if_0 (below figure) block is intended to establish communication and control to the base POXI interface board and Artix-7 FPGA[10]. Custom IP with target language VHDL, AXI peripheral 4 registers (32 bits each). For communication with POXI board, inside it is implemented a SPI module and its corresponding logic for chip select distribution, red and infrared LED drivers, Artix-7 board LEDs drivers, Artix-7 switches and pushbuttons.
Following figure below shows detailed information of the POXI ports and interfaces.

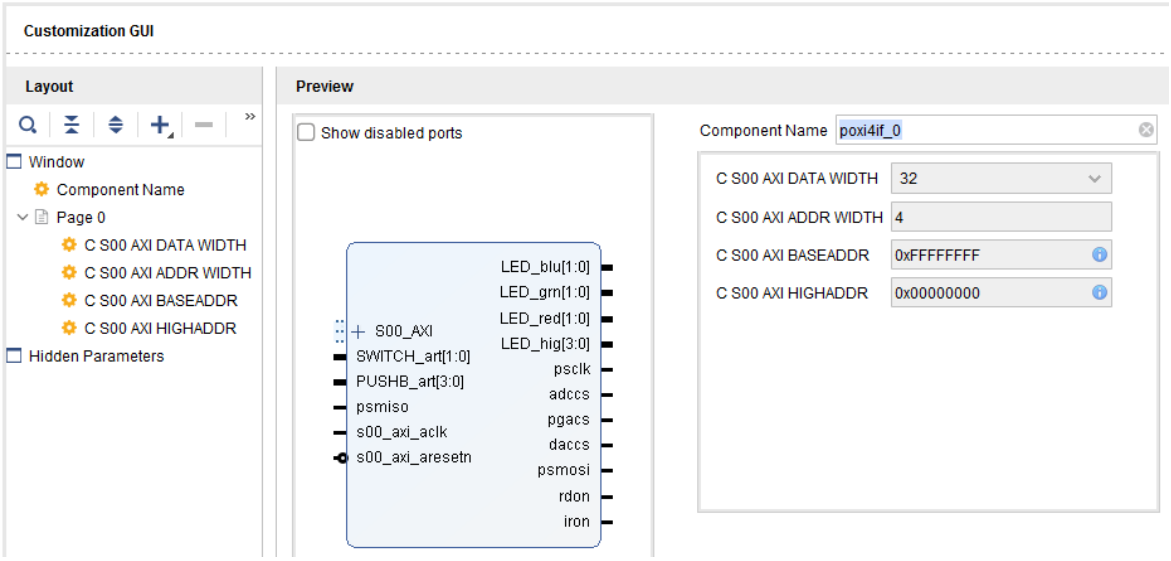


Figure 3.2: IP block diagram

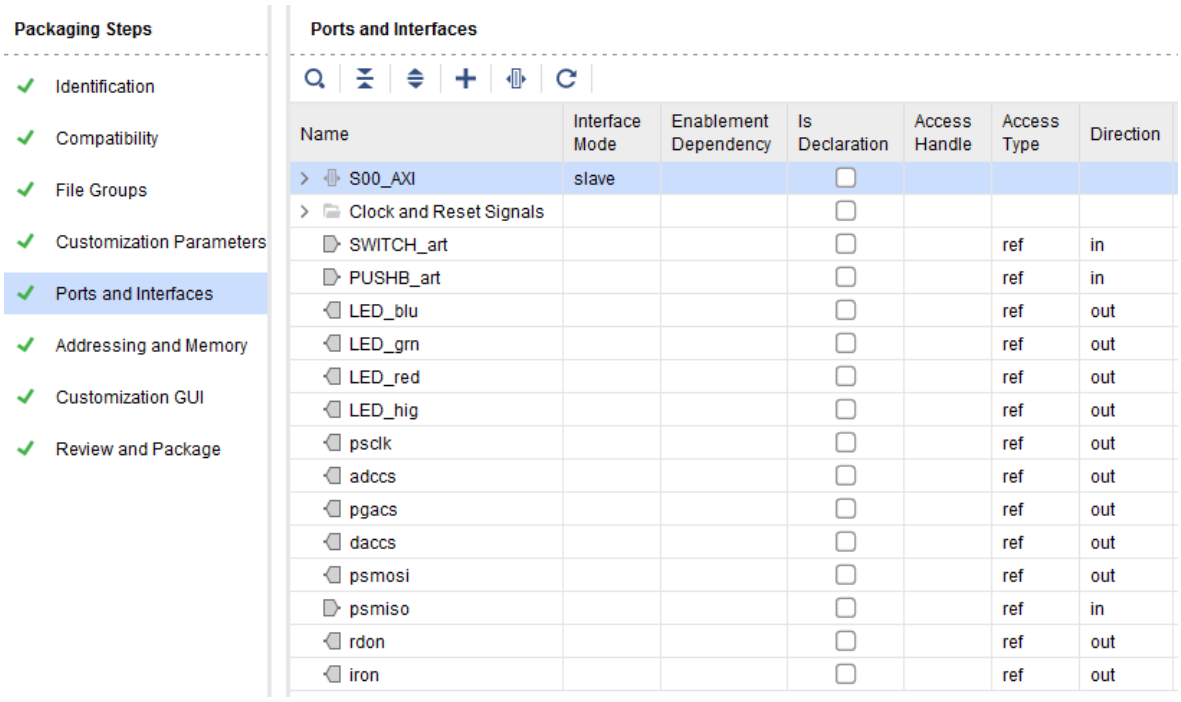


Figure 3.3: IP ports and interfaces

The figure below shows the file group contained in the IP. This IP contains three main VHDL files, two AXI interface description files, and one SPI description file. The "spifinal.vhd" contains the design for the SPI protocol, and "poxi4if_v1_0_S00_AXI.vhd" implements and instance for SPI communication and the rest of the logic, "poxi4if_v1_0.vhd" can be considered as wrapper-interface for the AXI peripheral.

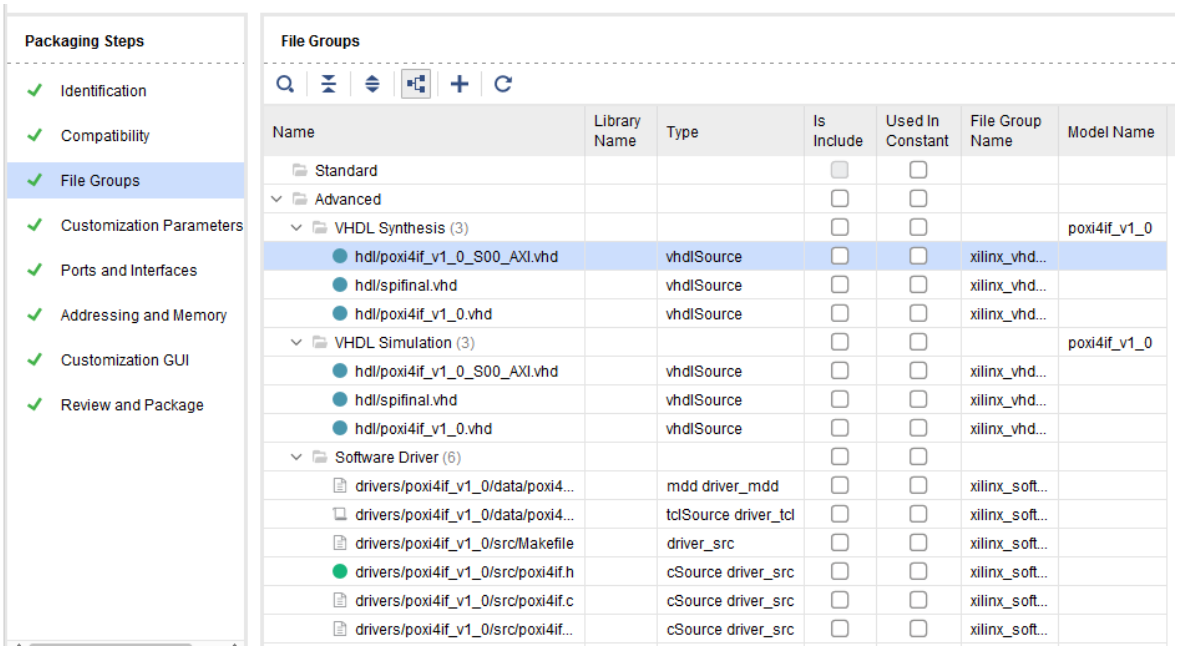


Figure 3.4: POXI file group

The main logic of the custom IP is contained in "poxi4if_v1_0_S00_AXI.vhd". Following changes were made in this file.

VHDL logic for SPI instantiation, Internal Signal Declaration:

```
--- SPI COMPONENT INSTANTIATION----
COMPONENT spifinal IS
GENERIC (DATA_LENGTH_BIT_SIZE   : INTEGER := 2;
         DATA_SIZE              : INTEGER := 32;
         BAUD_RATE_DIVIDER_SIZE : INTEGER := 8);
PORT ( sysclk       : IN STD_LOGIC;
reset               : IN STD_LOGIC;
data_length         : IN STD_LOGIC_VECTOR(DATA_LENGTH_BIT_SIZE-1 DOWNTO 0);
baud_rate_divider   : IN STD_LOGIC_VECTOR(BAUD_RATE_DIVIDER_SIZE-1 DOWNTO 0);
clock_polarity      : IN STD_LOGIC;
clock_phase         : IN STD_LOGIC;
start_transmission  : IN STD_LOGIC;
```

```
transmission_done  : OUT STD_LOGIC;
data_tx  : IN  STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0);
data_rx  : OUT STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0):=(others =>'0');
spi_clk  : OUT STD_LOGIC;
spi_MOSI : OUT STD_LOGIC;
spi_MISO : IN STD_LOGIC;
spi_cs   : OUT STD_LOGIC);
END COMPONENT spifinal;

--- INTERNAL SIGNAL DECLARATION FOR SPI ----
CONSTANT  SPI_DATA_LENGTH_BIT_SIZE   : INTEGER := 2;
CONSTANT  SPI_BAUD_RATE_DIVIDER_SIZE : INTEGER := 8;
CONSTANT  SPI_SLAVE_SELECT_SIZE      : INTEGER := 2;

CONSTANT  SPI_DAC_CS      : STD_LOGIC_VECTOR := "01";
CONSTANT  SPI_PGA_CS      : STD_LOGIC_VECTOR := "10";
CONSTANT  SPI_ADC_CS      : STD_LOGIC_VECTOR := "11";

SIGNAL spi_data_length       : STD_LOGIC_VECTOR
                       (SPI_DATA_LENGTH_BIT_SIZE-1 DOWNTO 0);
SIGNAL spi_baud_rate_divider  : STD_LOGIC_VECTOR
                       (SPI_BAUD_RATE_DIVIDER_SIZE-1 DOWNTO 0);
SIGNAL spi_clock_polarity    : STD_LOGIC;
SIGNAL spi_clock_phase       : STD_LOGIC;
SIGNAL spi_start_transmission: STD_LOGIC;
SIGNAL spi_transmission_done : STD_LOGIC;
SIGNAL spi_slave_select      : STD_LOGIC_VECTOR
                       (SPI_SLAVE_SELECT_SIZE-1 DOWNTO 0);
SIGNAL spi_data_rx           : STD_LOGIC_VECTOR
                       (C_S_AXI_DATA_WIDTH-1 DOWNTO 0);
SIGNAL spi_cs                : STD_LOGIC;
SIGNAL spi_reset_high        : STD_LOGIC;
```

VHDL process for reading registers.

```
process (slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN, slv_reg_rden,
```

```
upsmiso,spi_transmission_done,spi_data_rx)
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 <= x"0000" & spi_transmission_done &
                                        "000" & x"000";
                when b"01" =>
                reg_data_out <= x"0000000" & "000" & upsmiso;
                when b"10" =>
                reg_data_out <= spi_data_rx;
                when b"11" =>
                reg_data_out <= slv_reg3;
                when others =>
                reg_data_out  <= (others => '0');
        end case;
end process;
```

User Logic for LEDs ON drivers and Port Map for SPI module.

```
-- Add user logic here
spi_reset_high <= NOT S_AXI_ARESETN;
urdon <= slv_reg0(24);
uiron <= slv_reg0(25);
spi_baud_rate_divider  <= slv_reg1(07 downto 00);
spi_clock_phase        <= slv_reg1(8);
spi_clock_polarity     <= slv_reg1(9);
spi_data_length        <= slv_reg1(11 downto 10);
spi_slave_select       <= slv_reg1(17 downto 16);

udaccs <= NOT spi_cs WHEN (spi_slave_select = SPI_DAC_CS) ELSE '0';
upgacs <= NOT spi_cs WHEN (spi_slave_select = SPI_PGA_CS) ELSE '0';
uadccs <= NOT spi_cs WHEN (spi_slave_select = SPI_ADC_CS) ELSE '0';
```

```
spiif: spifinal
GENERIC MAP (DATA_LENGTH_BIT_SIZE   => SPI_DATA_LENGTH_BIT_SIZE,
        DATA_SIZE              => C_S_AXI_DATA_WIDTH,
        BAUD_RATE_DIVIDER_SIZE => SPI_BAUD_RATE_DIVIDER_SIZE)
       PORT MAP (
        sysclk             => S_AXI_ACLK,
        reset              => spi_reset_high ,
        data_length        => spi_data_length,
        baud_rate_divider  => spi_baud_rate_divider,
        clock_polarity     => spi_clock_polarity,
        clock_phase        => spi_clock_phase,
        start_transmission => spi_start_transmission,
        transmission_done  => spi_transmission_done,
        data_tx  => slv_reg2,
        data_rx  => spi_data_rx,
        spi_clk  => upsclk,
        spi_MOSI => upsmosi,
        spi_MISO => upsmiso,
        spi_cs   => spi_cs);

-- User logic ends
```

3.2 SPI VHDL Module

The communication between the Artix-7 FPGA device and the external devices is established by using SPI protocol. Inside the custom IPs (AXI peripherals) it is instantiated an final SPI which was designed with the following features.

- Data Transmission speed is configured to 500ns(2 MHz).
- Configurable data length (16 and 24 bits).
- Configurable clock polarity (CPOL) and clock phase (CPHA).
- Full duplex data transmission.

For details on SPI Protocol VHDL code, refer Appendix A.

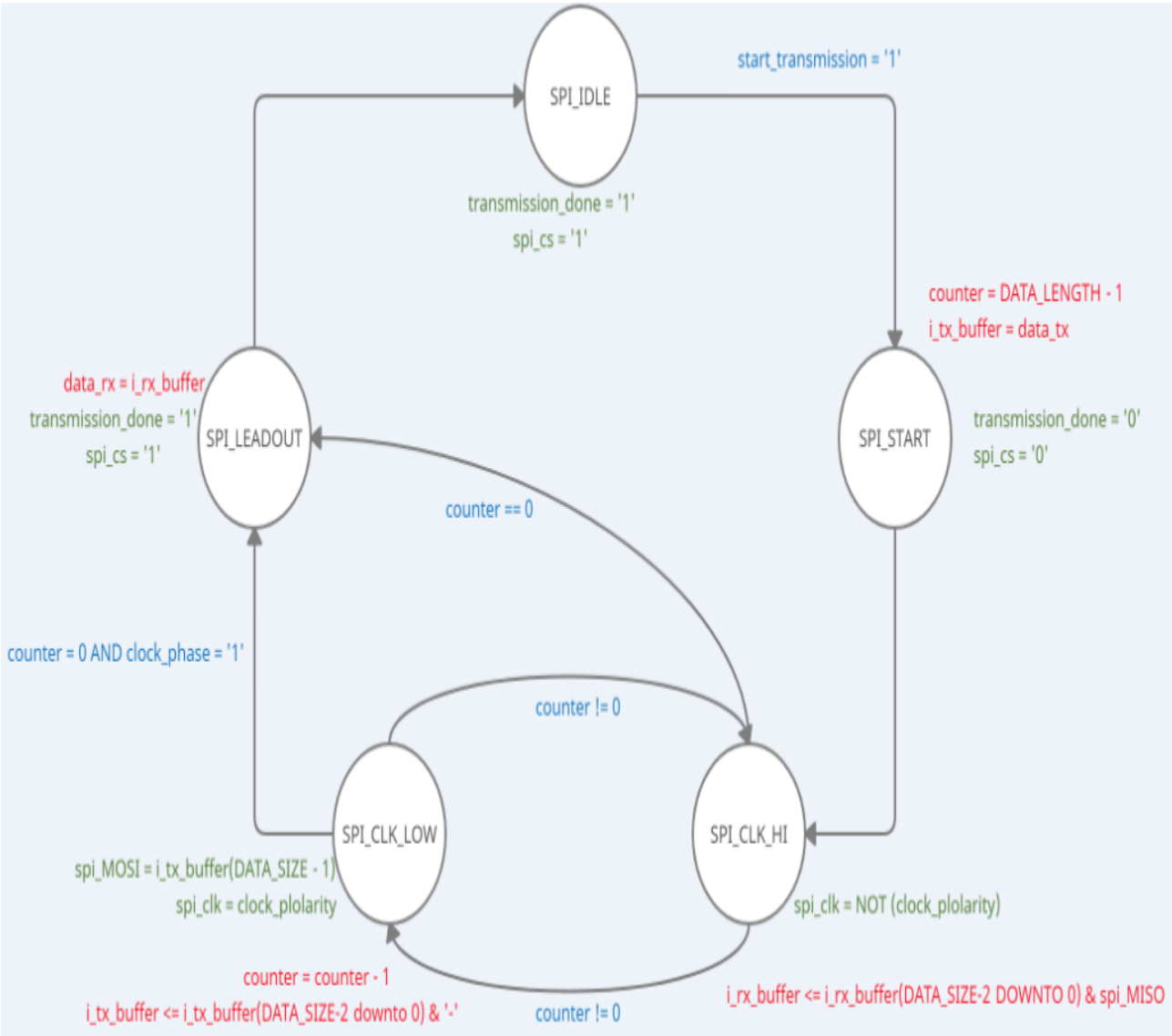
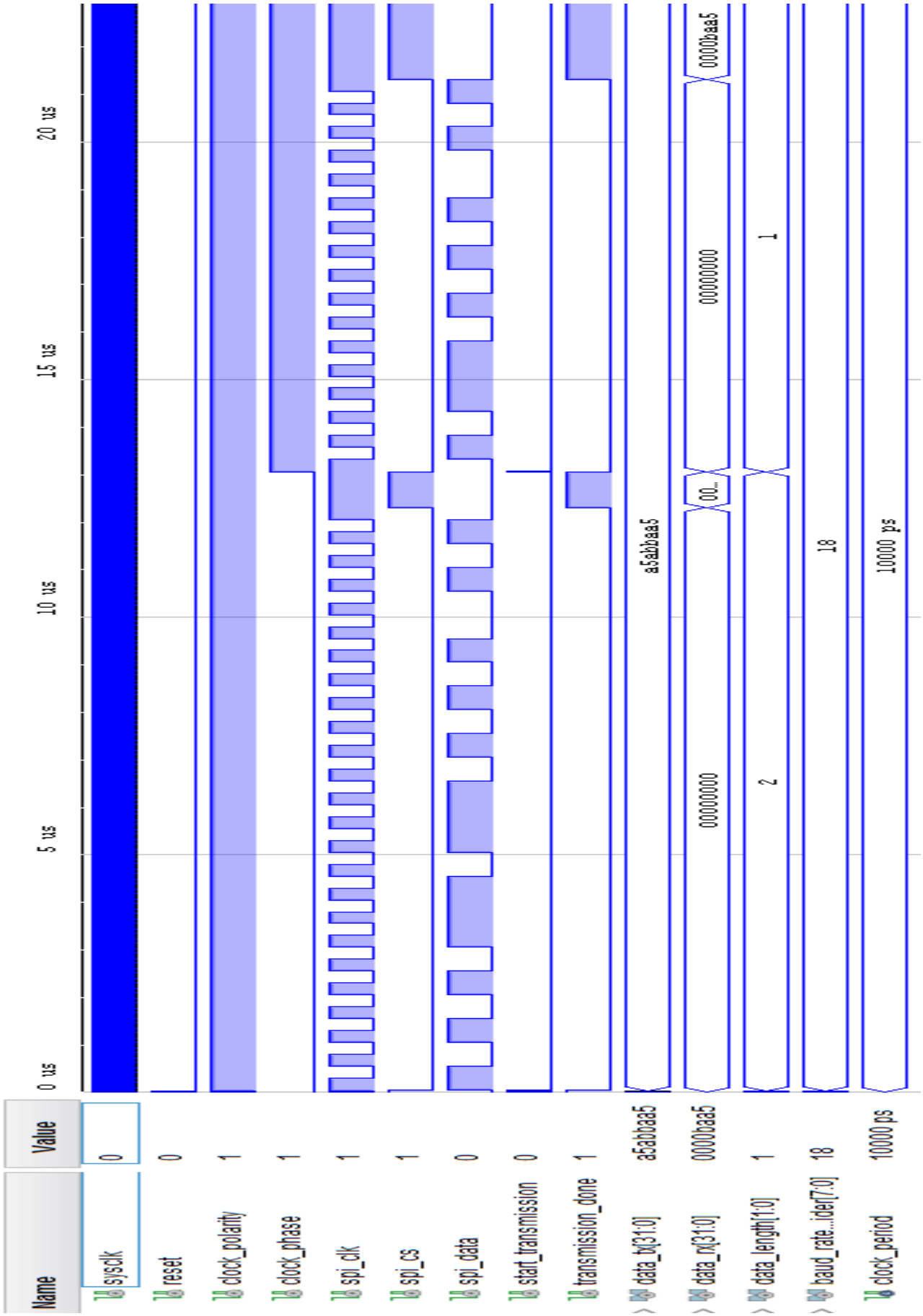


Figure 3.5: SPI state machine

3.2.1 SPI Simulation Output



Chapter 4

Digital Signal Progression (DSP)

A row data signal recored from Pulse oximeter application consist of unwanted noise which needs to be filtered before showing the result on screen. In this project 2^{nd} order Highpass and Lowpass filter has been used to rectify unwanted noise.

4.1 Using MATLAB

This application uses MATLAB mainly for following two purposes:

(a) To calculate filter coefficients

(b) Validate C code using Legacy function block in Simulink

2^{nd} Order Filter coefficient

```
>> poxi_raws
High pass filter coeffcients:
    IIR_Filt.num[0] = 0.997337820133346;
    IIR_Filt.num[1] = -1.99467564026669;
    IIR_Filt.num[2] = 0.997337820133346;

    IIR_Filt.den[0] = 1;
    IIR_Filt.den[1] = -1.99466855305249;
    IIR_Filt.den[2] = 0.994682727480893;

Low pass filter coeffcients:
    IIR_Filt.num[0] = 0.00134871194835634;
    IIR_Filt.num[1] = 0.00269742389671268;
    IIR_Filt.num[2] = 0.00134871194835634;

    IIR_Filt.den[0] = 1;
    IIR_Filt.den[1] = -1.89346414636183;
    IIR_Filt.den[2] = 0.898858994155252;

That`s all, folks.
```

Pulse Oximeter Raw Data

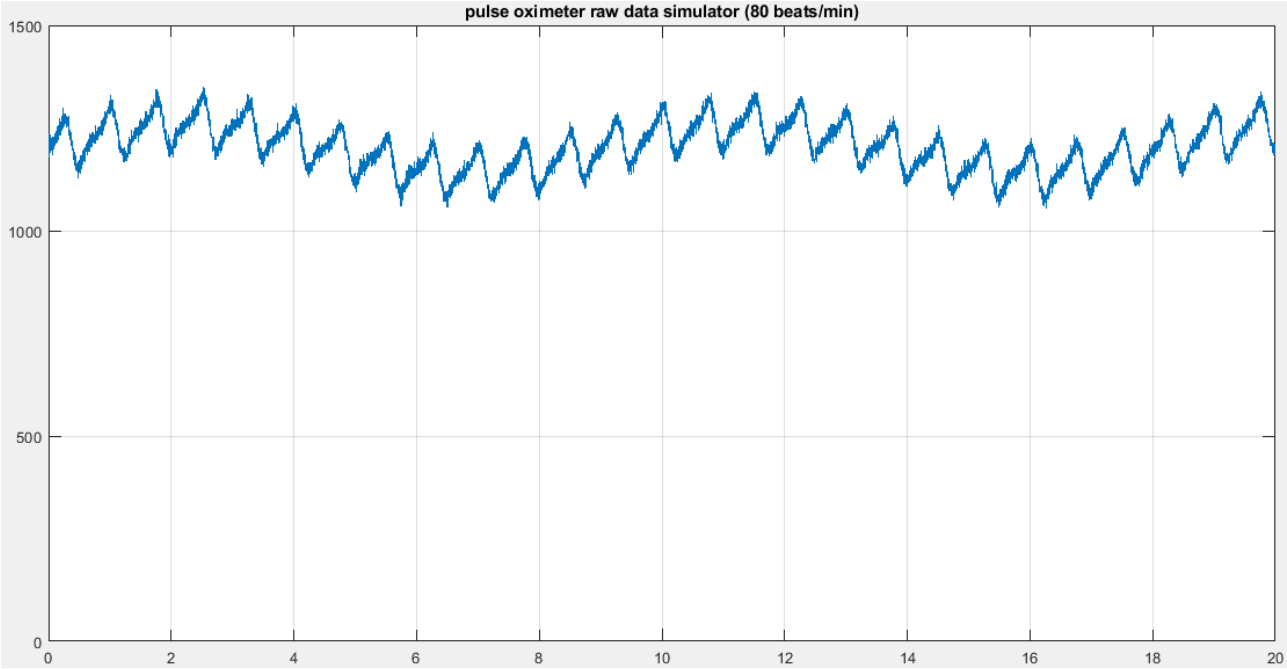


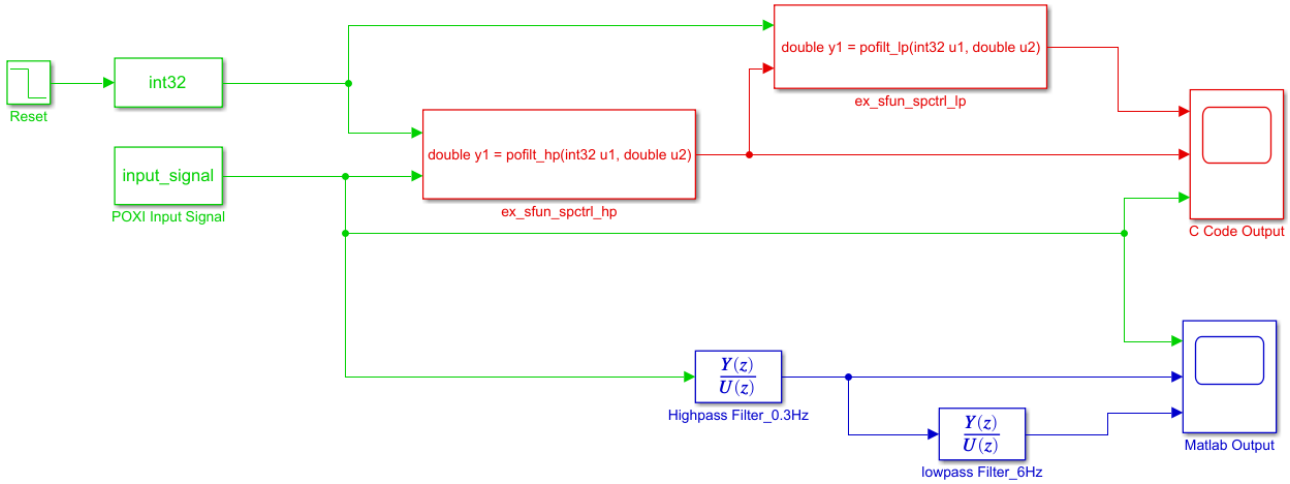
Figure 4.1: POXI raw data

MATLAB Legacy Code

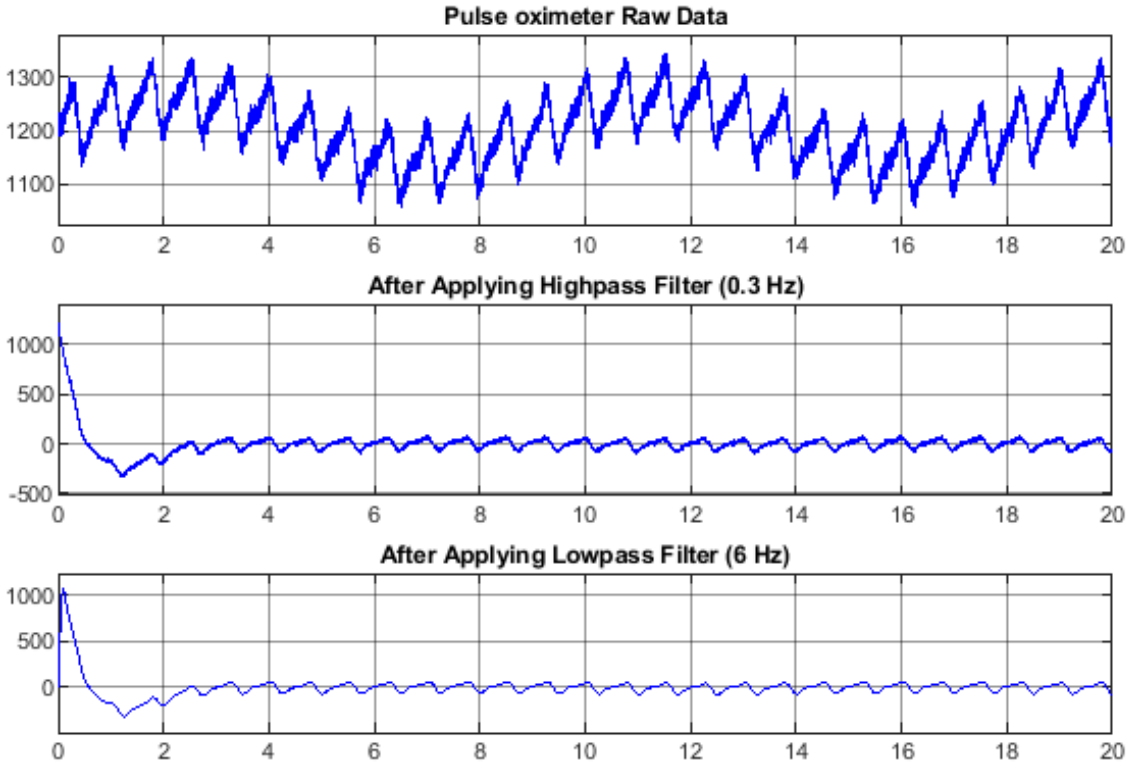
```
% create legacy C code mex function for Highpass filter
def = legacy_code('initialize');
def.SourceFiles = {'pofilt_hp.c'};
def.HeaderFiles = {'pofilt_hp.h'};
def.SFunctionName = 'ex_sfun_spctrl_hp';
% double pofilt(int reset, double u);
def.OutputFcnSpec = 'double y1 = pofilt_hp(int32 u1, double u2)';
legacy_code('sfcn_cmex_generate', def)
legacy_code('compile', def)
legacy_code('slblock_generate', def)

% create legacy C code mex function for Lowpass filter
def = legacy_code('initialize');
def.SourceFiles = {'pofilt_lp.c'};
def.HeaderFiles = {'pofilt_lp.h'};
def.SFunctionName = 'ex_sfun_spctrl_lp';
% double pofilt(int reset, double u);
```

```
def.OutputFcnSpec = 'double y1 = pofilt_lp(int32 u1, double u2)';
legacy_code('sfcn_cmex_generate', def)
legacy_code('compile', def)
legacy_code('slblock_generate', def)
disp('That's all folks.')
```

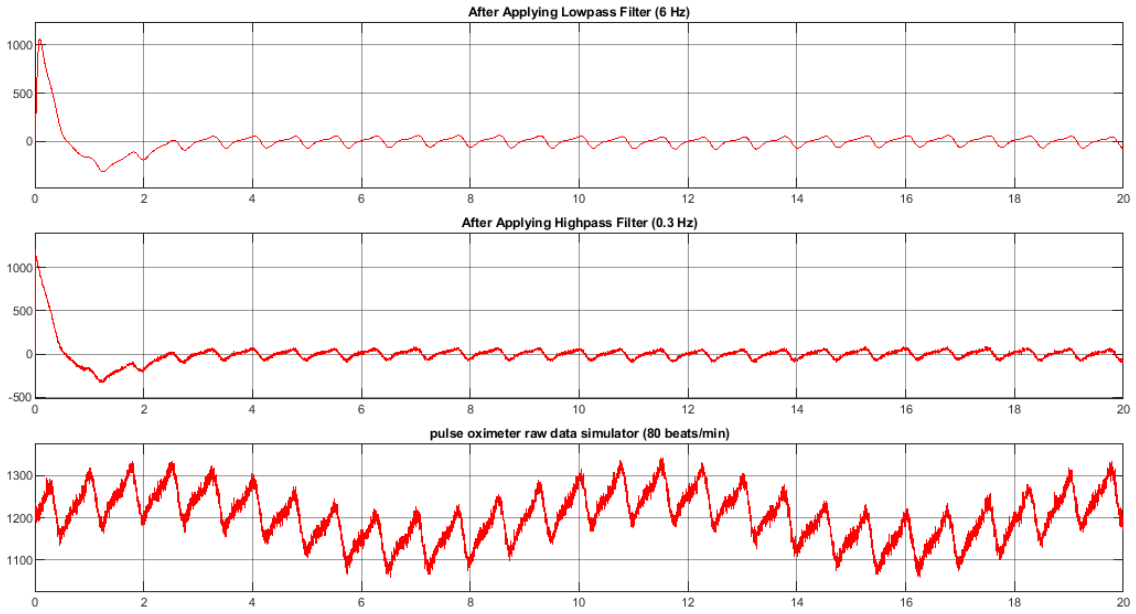
Simulink Model for Verification



MATLAB Filter Result



C code Filter Result



Above figures illustrate that C code works perfectly as expected for the filter design. This checks is not mandatory but it gives clear idea if written c code working as expected or not before run it on hardware.

4.2 Using C code

Transpose direct form II IIR filter is implemented in this project and their coefficients are computed for 2^{nd} order Butter worth Highpass and Lowpass filter with pass bank ranging from 0.3Hz - 6Hz.

TDF II block diagram

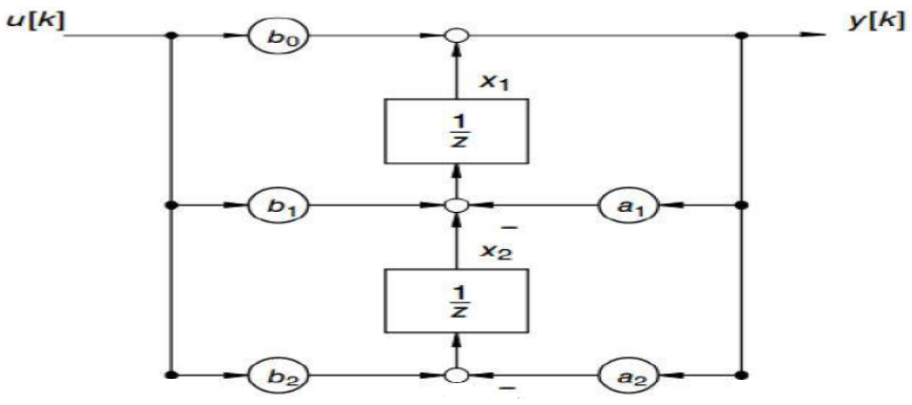


Figure 4.2: TDF-II block diagram

Highpass filter C code

```
/*
============================================================
Name        : pofilt_hp.c
Author      : Dharmesh Patel
Version     : V0.a 15.01.2020
Copyright   : (c) 2020 Univ. Bremerhaven , Kai Mueller
Description : 2nd order High pass filter - 0.3Hz  (discrete)
============================================================
*/

#include "pofilt_hp.h"

#define BH_0 0.997337820133346
#define BH_1 -1.99467564026669
#define BH_2 0.997337820133346
#define AH_1 -1.99466855305249
#define AH_2 0.994682727480893

typedef struct {
                double b0, b1, b2, a1, a2;
                double u_k1, u_k2, y_k1, y_k2;
        }Filterobj;

static Filterobj HP_f;

//Filter for poxi data (High pass 0.3 Hz)
double pofilt_hp(int reset, double u)
{
                double y_out;
                if (reset>0) {
                HP_f.b0 = BH_0;
                HP_f.b1 = BH_1;
                HP_f.b2 = BH_2;
                HP_f.a1 = AH_1;
```

```
                HP_f.a2 = AH_2;
                HP_f.u_k1 = 0.0;
                HP_f.u_k2 = 0.0;
                HP_f.y_k1 = 0.0;
                HP_f.y_k2 = 0.0;
                y_out = 0;
} else {
                y_out = HP_f.b0*u + HP_f.b1*HP_f.u_k1 +
                HP_f.b2*HP_f.u_k2 - HP_f.a1*HP_f.y_k1 -
                HP_f.a2* HP_f.y_k2;
                HP_f.u_k2 = HP_f.u_k1;
                HP_f.u_k1 = u;
                HP_f.y_k2 = HP_f.y_k1;
                HP_f.y_k1 = y_out;
}
return y_out;
}
```

Lowpass filter C code

```
/*
============================================================
Name        : pofilt_lp.c
Author      : Dharmesh Patel
Version     : V0.a 15.01.2020
Copyright   : (c) 2020 Univ. Bremerhaven , Kai Mueller
Description : 2nd order Low pass filter - 6Hz  (discrete)
============================================================
*/

#include "pofilt_lp.h"

#define BL_0 0.00134871194835634
#define BL_1 0.00269742389671268
#define BL_2 0.00134871194835634
#define AL_1 -1.89346414636183
```

```
#define AL_2 0.898858994155252

typedef struct {
                double b0, b1, b2, a1, a2;
                double u_k1, u_k2, y_k1, y_k2;
                }Filterobj;

static Filterobj LP_f;

//Filter for poxi data (Low pass 6 Hz)
double pofilt_lp(int reset, double u)
{
                double y_out;
                if (reset>0) {
                LP_f.b0 = BL_0;
                LP_f.b1 = BL_1;
                LP_f.b2 = BL_2;
                LP_f.a1 = AL_1;
                LP_f.a2 = AL_2;
                LP_f.u_k1 = 0.0;
                LP_f.u_k2 = 0.0;
                LP_f.y_k1 = 0.0;
                LP_f.y_k2 = 0.0;
                y_out = 0.0;
} else {
                y_out = LP_f.b0*u + LP_f.b1*LP_f.u_k1 +
                LP_f.b2*LP_f.u_k2 - LP_f.a1*LP_f.y_k1 -
                LP_f.a2* LP_f.y_k2;
                LP_f.u_k2 = LP_f.u_k1;
                LP_f.u_k1 = u;
                LP_f.y_k2 = LP_f.y_k1;
                LP_f.y_k1 = y_out;
}
return y_out;
}
```

Chapter 5

Software Solution (SDK)

Pulse oximeter project consists software part that can provide user manual options to do the operation also the output filtered data is transfred to GUI. The software code is implemented in C language. The desktop application is intended to have a Graphical User Interface (GUI) to present processed signals and information sent from the POXI device. This desktop application is written in Java language.

The following figure shows the software architecture diagram.

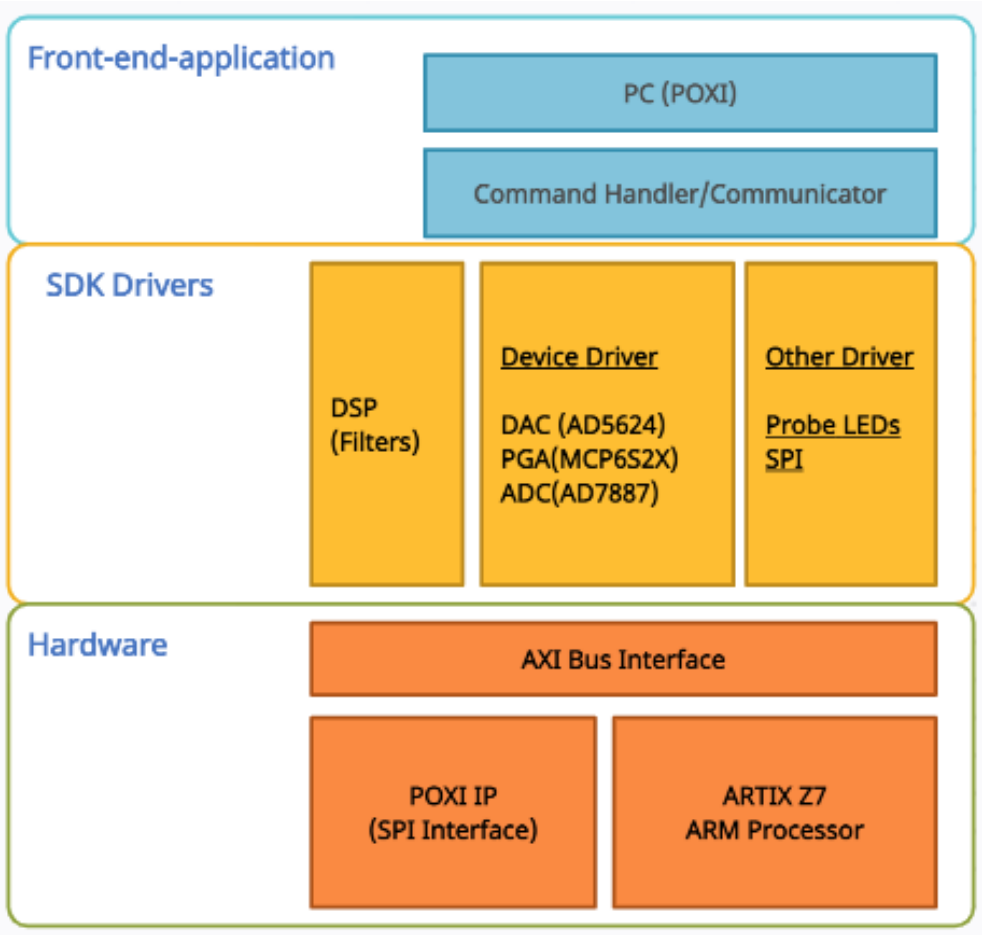


Figure 5.1: Software Architecture diagram

5.1 Device drivers

The POXI board (base hardware) consist basically of a data acquisition system, it contains an analogue to digital converter (ADC) to digitalize signals of red and infrared light absorbance (signals from finger clip), also uses a digital to analogue converter (DAC) to define and provide analogue signals for red and infrared light intensities, finger clip sensor bias voltage and data acquisition offset voltage subtraction (DC component of light absorbance signal, or ambient light). For signal conditioning, is also used a programmable gain amplifier (PGA). These devices (DAC, ADC, PGA) establish communication with the AXI peripherals by using SPI protocol communication.

In order to have a proper software device driver for POXI board, it was implemented the device driver file for these SPI devices, and red and infrared LEDs flinger clip drivers. Following Figure shows list of Driver functions used to achieve the proper sequence of operation for the application.

```
/*
 * ------------------------------------------------------------
 * Interrupt handler (ZYNQ private timer)
 * ------------------------------------------------------------
 *
 * ------------------------------------------------------------
 * Step 1: Turn ON Red LED
 * Step 2: Read ADC value,Filter it and update it to internal buffer and Turn OFF LEDs
 * Step 3: Turn ON Infrared LED
 * Step 4: Read ADC value,Filter it and update it to internal buffer and Turn OFF LEDs
 * ------------------------------------------------------------
 */
static void TimerIntrHandler(void *CallBackRef)
{
// Initialize filter input and output object variables
static void pofilt_init()
{
//Filter for POXI data (High pass 0.3 Hz and Low pass 6 Hz)
static double pofilt(double u)
{
// Function for DAC configuration
static void DAC_config()
{
// DAC write Function based on input Channel and Value
static void DAC_Write(int chan, int val)
// PGA write Function to Set Gain
static void PGA_Write()
{
// ADC Read Function, Conversion value will be updated to ADC_VAL resister after transmission is done
static short int ADC_Read()
{
// clear outputs and turn off Interface devices on exit
static void ClearAll()
{
// To initialize Interrupt handler function
static void Poxi_initInterruptHandlers(void)
{
// To transfer sensor recorded data to GUI for Plotting the output
static void TransferTo_GUI(){
// Main function (starting point)
int main()
{
```

Figure 5.2: Functions list of the Software

5.1.1 DAC,PGA and ADC Drivers

As per the DAC(AD5624R) data sheet device needs to be configure before operating it. Set of command needs to send in proper sequence to the device which makes it ready to accept actual data set and increase the intensity of the sensor light. This driver implements the SPI communication with the hardware.

Following function will initialize SPI protocol for the DAC.

```
// Function for DAC configuration
static void DAC_config()
{
while(!SPI_DONE);
SPI_CONFIG = 0x00010a10;// (Data Length = 24("10"),CPOL = 1, CPHA = 0)
SPI_TX_BIT = 0x00280000;// Reset DAC internal registers and channels
while(!SPI_DONE);
SPI_CONFIG = 0x00010a10;
SPI_TX_BIT = 0x0020000F; // Power-up DAC channels
while(!SPI_DONE);
SPI_CONFIG = 0x00010a10;
SPI_TX_BIT = 0x00300000; // setting-up LDAC register
while(!SPI_DONE);
SPI_CONFIG = 0x00010a10;
SPI_TX_BIT = 0x00380001; // setting-up DAC internal reference
}
```

For DAC write function and its operation please refere appendix B.

Following function will set PGA Gain to 8.

```
// PGA write Function to Set Gain
static void PGA_Write()
{
while(!SPI_DONE);
SPI_CONFIG = 0x00020710;// (Data Length = 16("01"),CPOL = 1, CPHA = 1)
SPI_TX_BIT = 0x4004; // Set PGA gain to 8 (gain register:="100")
}
```

Following function will get ADC value.

```
// ADC Read Function, Conversion value will be updated to ADC_VAL
// resister after transmission is done
static short int ADC_Read()
{
while(!SPI_DONE);
SPI_CONFIG = 0x00030680;// (Data Length = 16("01"),CPOL = 1, CPHA = 0)
SPI_TX_BIT = 0b0000000100000000;
while(!SPI_DONE);
return 0x0fff & ADC_VAL;
}
```

5.1.2 Driver for Serial Communicator

"SerialNetw" JAVA class executes and ensures a proper communication between the POXI integrated device and the desktop systems. This class implements a protocol communication over the UART serial communication layer. Following function send acquired sensor data to GUI for ploatting and show results on Graphs.

```
// To transfer sensor recorded data to GUI for Plotting the output
static void TransferTo_GUI(){
if (Logger_flag)
 {
     // For logging data start string with "~"
        if(sample<Buffer_L)
        {
        printf("~ %f %f %f %f\n",Rbuffer[sample],IRbuffer[sample],
                    Rbuffer_filt[sample],IRbuffer_filt[sample]);
        sample++;
        }
        else
        {
                printf("@ %f\n",13.000); // Exit the application
        }
 }
}
```

5.1.3 Timer Interrupt Handler

Timer interrupts allow you to perform a task at very specifically timed intervals regardless of what else is going on in your code. Following sequence of task has been performed every 2ms using ARTY Z7 provided timer interrupt.

```
/*
* ----------------------------------------------------------
* Interrupt handler (ZYNQ private timer)
* ----------------------------------------------------------
*
* ----------------------------------------------------------
* Step 1: Turn ON Red LED
* Step 2: Read ADC value,Filter it and update it to internal buffer
          and Turn OFF LEDs
* Step 3: Turn ON Infrared LED
* Step 4: Read ADC value,Filter it and update it to internal buffer
          and Turn OFF LEDs
* ----------------------------------------------------------
*/
static void TimerIntrHandler(void *CallBackRef)
{
XScuTimer *TimerInstance = (XScuTimer *)CallBackRef;

XScuTimer_ClearInterruptStatus(TimerInstance);

switch(ISR_Count)
{
        case 0:
                RedIred = 0b01; // 24th bit is used for red LED in HW
                ISR_Count++;
                break;
        case 1:
                adc_reading = ADC_Read();
                filt_out = pofilt(adc_reading);
                RedIred = 0;
                if (R_size < Buffer_L)
```

```
                {
                Rbuffer_filt[R_size] = filt_out;
                Rbuffer[R_size] = adc_reading;
                R_size++;
                }
                ISR_Count++;
                break;
        case 2:
                RedIred = 0b10;//25th bit is used for Infrared LED
                ISR_Count++;
                break;
        case 3:
                adc_reading = ADC_Read();
                filt_out = pofilt(adc_reading);
                RedIred = 0;
                if (IR_size < Buffer_L)
                {
                IRbuffer_filt[IR_size] = filt_out;
                IRbuffer[IR_size] = adc_reading;
                IR_size++;
                }
                ISR_Count = 0;
                break;
        default:;
}

}
```

Timer load value calculation:

$Sampling\_rate = \frac{333.333\ \ MHz}{Counter\_Load}$

So, for 2 ms Timer load value is:

$500\ \ Hz = \frac{333.333e6\ \ Hz}{666666}$

Timer load value is devided by 4 since we have 4 switch cases in in the ISR.

Note: Appendix B can be refered in order to get full software solution code for pulse oximeter application.

5.2 Flow chart of the Code execution

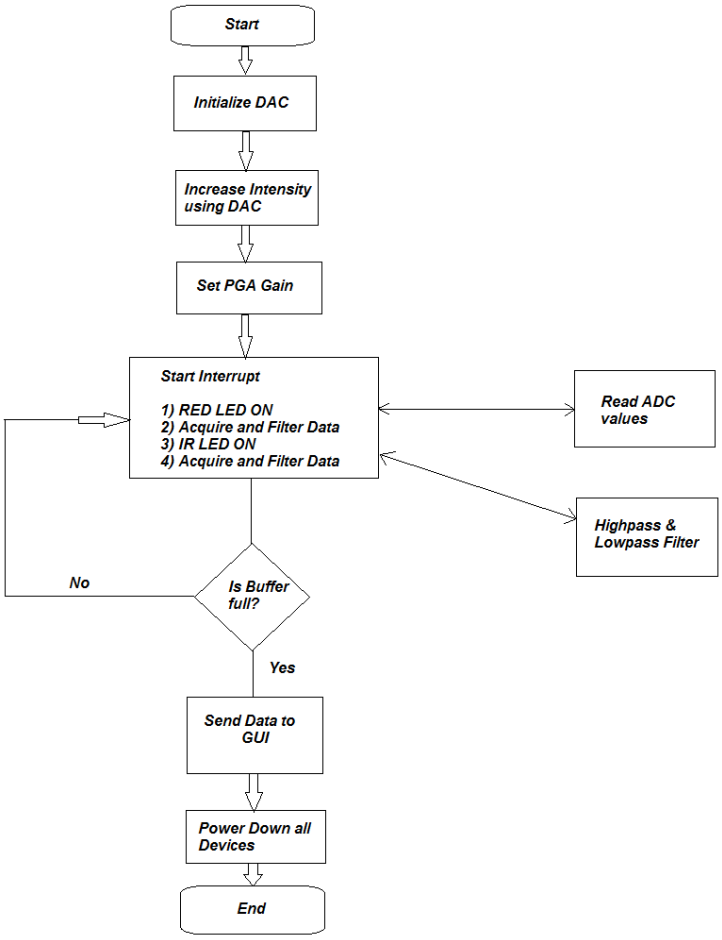


Figure 5.3: SW Code execution Flow chart

Chapter 6

Graphical user Interface (GUI)

6.1 Introduction

The **graphical user interface (GUI)** is a form of user interface that allows users to interact with electronic devices through graphical icons and an audio indicator such as primary notation, instead of text-based user interfaces, typed command labels or text navigation. GUIs were introduced in reaction to the perceived steep learning curve of command-line interfaces (CLIs), which require commands to be typed on a computer keyboard. The actions in a GUI are usually performed through direct manipulation of the graphical elements.
Based on the requirement of continuous plotting of the output graph, the GUI continuously plots the output graph (for red and infrared light). The GUI enables the host computer to connect with the UART port of FPGA via a J-tag loader and shows the results on Graphical User Interface by pressing buttons or typing commands for specific tasks.

6.2 GUI Implementation

Java uses the Composite Design Pattern to create GUI components that can also serve as containers to hold more GUI components. Here is a UML diagram showing the class relations between a few of the more commonly used GUI components:
The commonly used container components are in orange while the non-container components are in blue. Note that technically, all javax.swing components are capable of holding other Components, but in practice, only JFrame, JApplet, and JPanel, plus a few not shown above (JSplitPane, JTabbedPane and JScrollPane) are commonly used for that purpose.

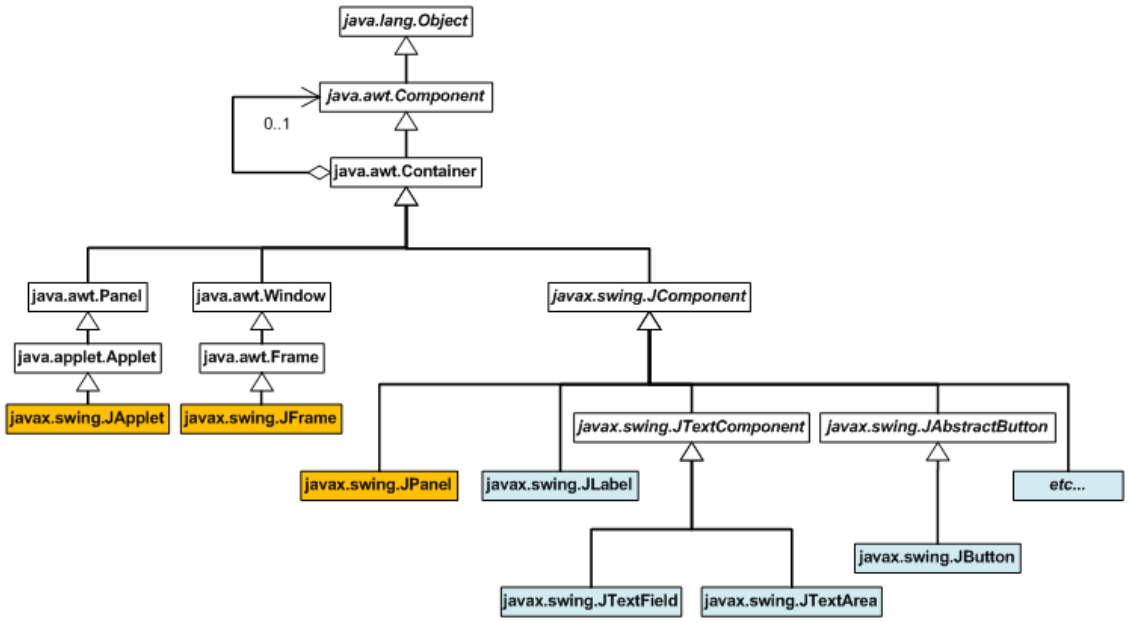


Figure 6.1: UML diagram[12]

In order to implement GUI, we need to establish Serial Communication between the Hardware and the PC in order to send or receive data from Hardware, a dedicated window to incorporate user interfaces.

In Pulse Oximeter project we require different JAR files namely RXTXComm.jar and JChart2D.jar. In order to add the above mentioned External Dependency JAR files to the Project we have to configure build path and select Add External JARS.

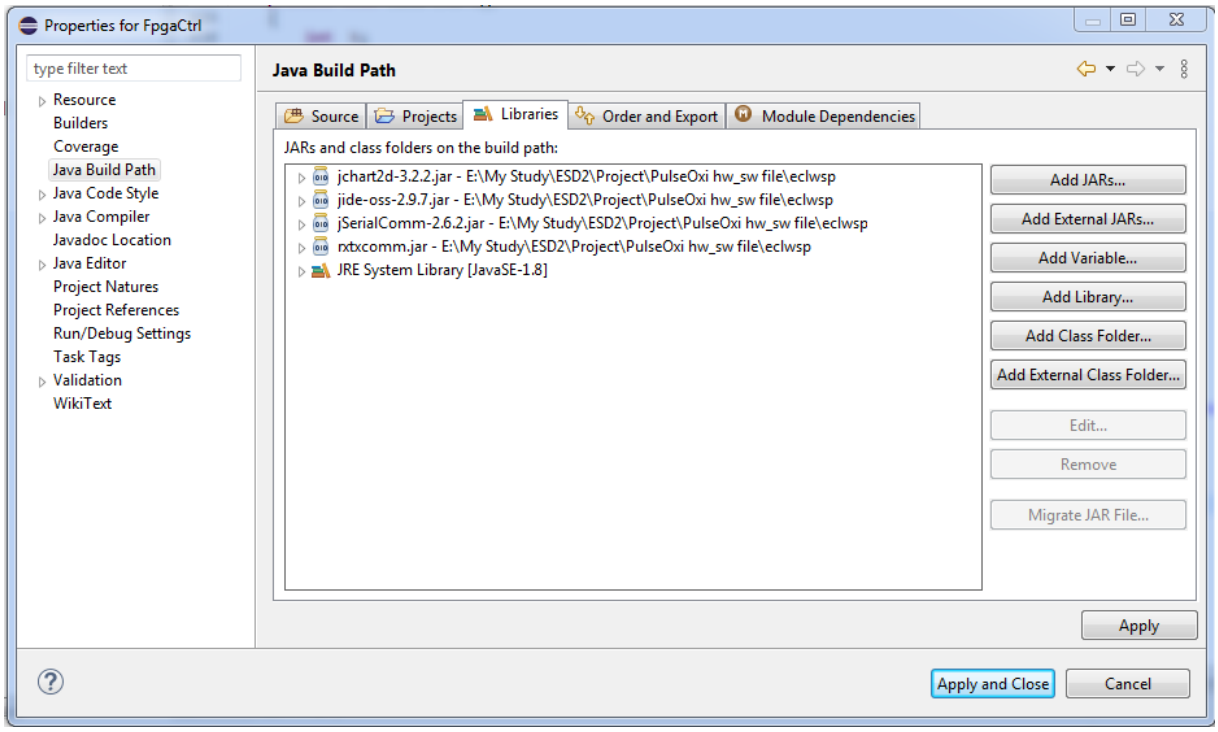


Figure 6.2: additional .jar files

RXTXComm file provides all the Methods, Classes and Interfaces to establish Serial Communication and JChart2D is use for displaying two-dimensional traces in a coordinate system written in Java. It supports real-time (animated) charting, custom trace rendering, Multithreading, viewports, automatic scaling and labels.
Befor working with design view, windows builder should be installed for JAVA application. You can find this installation Help⇒Install new Software⇒select JAVA app version⇒General purpose tools⇒select and Install.
This GUI contains three java files named FpgaCtrl.java, SerialNetw.java and CreateChart.java. FpgaCtrl file is to design Front End, CreateChart file is use to Plot Heart Pulse and SerialNetw.java file is to Establish Serial Communication between hardware and PC.

6.2.1 Serial network class

As the "FpgaCtrl" application is launched it initializes the "SerialNetw" in the main class. Normally, communication with serial ports involves these steps:

- Searching for serial ports.
- Connecting to the serial port and initializing the input output streams.
- Adding an event listener to listen for incoming data.
- Receiving Data.
- Disconnecting from the serial port.

6.3 GUI Contents

The following are the main contents of Graphical User Interface.

6.3.1 Main Frame

The interface contains main frame which is called as "Pulse Oximeter GUI" the background color, dimension (size) and other settings can be adjusted using corresponding java code directly or from design view also. "JFrame" class is used for main frame and JFrame is displayed in the upper-left corner of the screen. Method of setLocation(x, y) in the JFrame class may be used. This method places the upper-left corner of a frame at location (x, y).

Inside the main frame there are three main components.

(1) Command Panel

(2) Scroll/Text Panel

(3) Graphical Panel

JFrame objects store several objects including a container object known as the content pane. We have added several components to a JFrame by adding it to content pane. Following figure shows included components. These are added using getContentPane method of main frame.

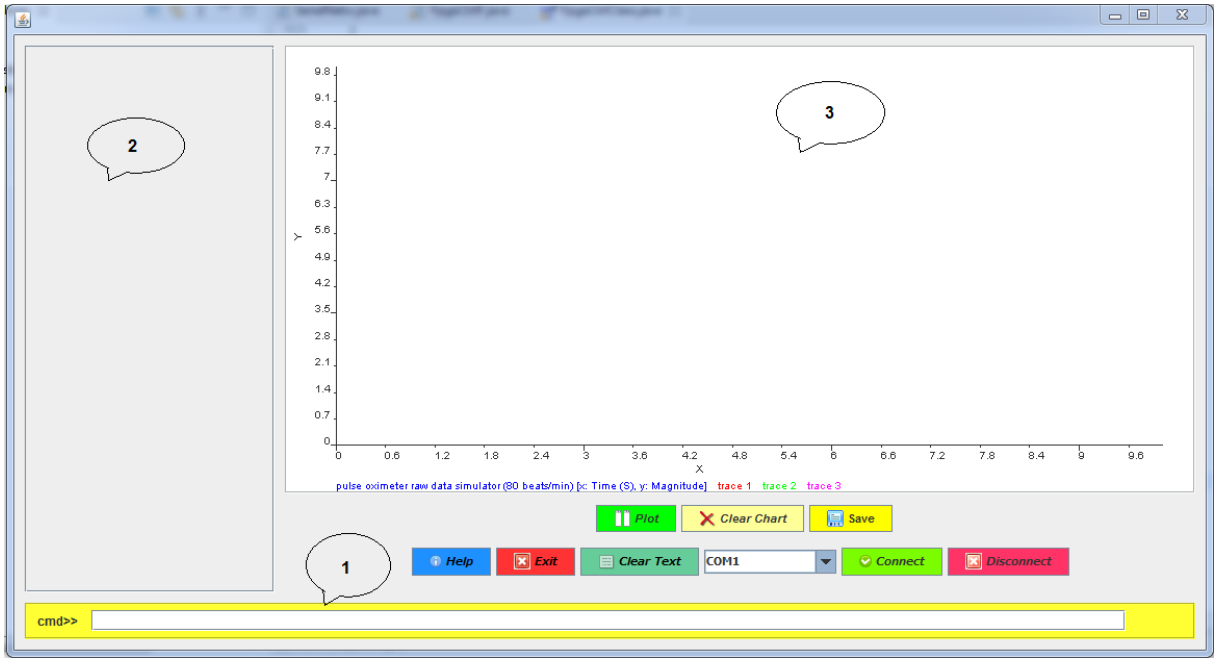


Figure 6.3: GUI Component Overview

6.3.2 Command panel

Command Panel is added at the bottom of the main frame using JPanel class. Setting the background color, dimensions and borders can be done either from design view or by directly modifying java code. Command panel contains the following:

Text field

Using JTextField class a text field is made so that user can type commands. This text field is added in command panel also it is labeled as **"cmd**>>**"** using JLabel. Whenever user type some commands in the text field, then action performed by the action listener.

Buttons

Few buttons are added in command panel using Jbutton class in java. Every time a button is pressed an action listener is added and corresponding action is performed.

Plot

Plot button is created using JButton class and is added in the command panel. In the action listener we are setting "Go= true" where recv_s is printed in the output text window and try catch box is used to look for the values of type double in recv_s and passes it to CreateChart class where it is plotted on the graphic panel.

Clear Chart

Clear chart button is added in command panel to clear the chart from graphic panel. Whenever this button is pressed the action listener perform the action of removing all points from mtraces and allowing the trace to start from zero using addPoint (0.0, 0.0) method of mtraces.

Save

Save button is added in command panel to download the ploted data to local directory as "poxi_data.dat" file.

Help

Help button is added in command panel to show user all possible command that user can type manually in text field and which will work as a terminal.

Clear Text

Clear chart button is added in command panel to clear the chart from graphic panel. Whenever this button is pressed the action listener perform the action of removing all text from texts window.

Devices Dropdown button

Devices Dropdown button is added in command panel which shows the all the available or connected devices with serial ports using getdevName method of SerialNetw class.

Connect

Connect button is added in command panel which will connect selected serial port from the Devices Dropdown button and print "OK" message on text window.

Disconnect

Disconnect button is added in command panel which will disconnect current active port connection.

Exit

Exit button will exit from GUI window.

6.3.3 Scroll Pane

Scroll pane is added in the main frame using JScrollPane. This class allows you to add scroll bars if necessary. E.g. if the text in output text pane is exceeded then automatically scroll bar appears.

Output Text Pane:

Output text pane is added in the main frame using JTextPane class. It enables the users to see the serial data from FPGA and also to see the results of typed commands or pressed buttons. Method of setViewportView is used to add output text pane in scroll pane. PrintTxtWin method is created to define style of the text and prints serial data in output text pane.

6.3.4 Graphical Panel

Graphic panel contains the chart and particular trace to plot the result in graphical form (that has been received from FPGA via serial communication). It uses a Chart2D class. Chart contains trace using addTrace method. Trace uses Trace2DSimple class. In this way serial data coming from FPGA detected using RXTX library at a specified port on host computer is stored in an array of type double then printed on output text pane and also plotted in graphic panel.

6.3.5 Command Handler

Besides using the GUI buttons, command handler method is also created which reads string of commands entered in command text field. Under this method required tasks are allocated for particular commands.

“clc” – this command clears the output textpane.

“clg” – this command clears the graphical pane. It calls ClearChart () method for this purpose.

“help” – this command shows the help instructions for command.

“dev” – this command shows the available or connected serial ports. It uses getNDev method of SerialNetw class.

“conn comX”– ”X” shows the port number. This command checks the status of the device using getDevStat method SerialNetw class, if the device is connected with the entered serial port. it connects your device with port using spConnect method of SerialNetw after a successful connection ”OK” will be printed in the output text pane.

“plot” – this command plots the heart pulse in the graphical panel. When it is typed UpdateDynamic() thread is executed. In which received string “recv_s” from serial data is obtained using getstring () method of SerialNetw class. Now this “recv_s” is passed to PrintTxtWin

where a try catch box is added to catch values of type double from the received string and passes it to CreateChart method.

"disconn" – this command disconnects the serial communication between FPGA and PC returning "OK" on output text pane showing successful disconnection.

"exit" – this command is used to close the application.

6.4 Output Result

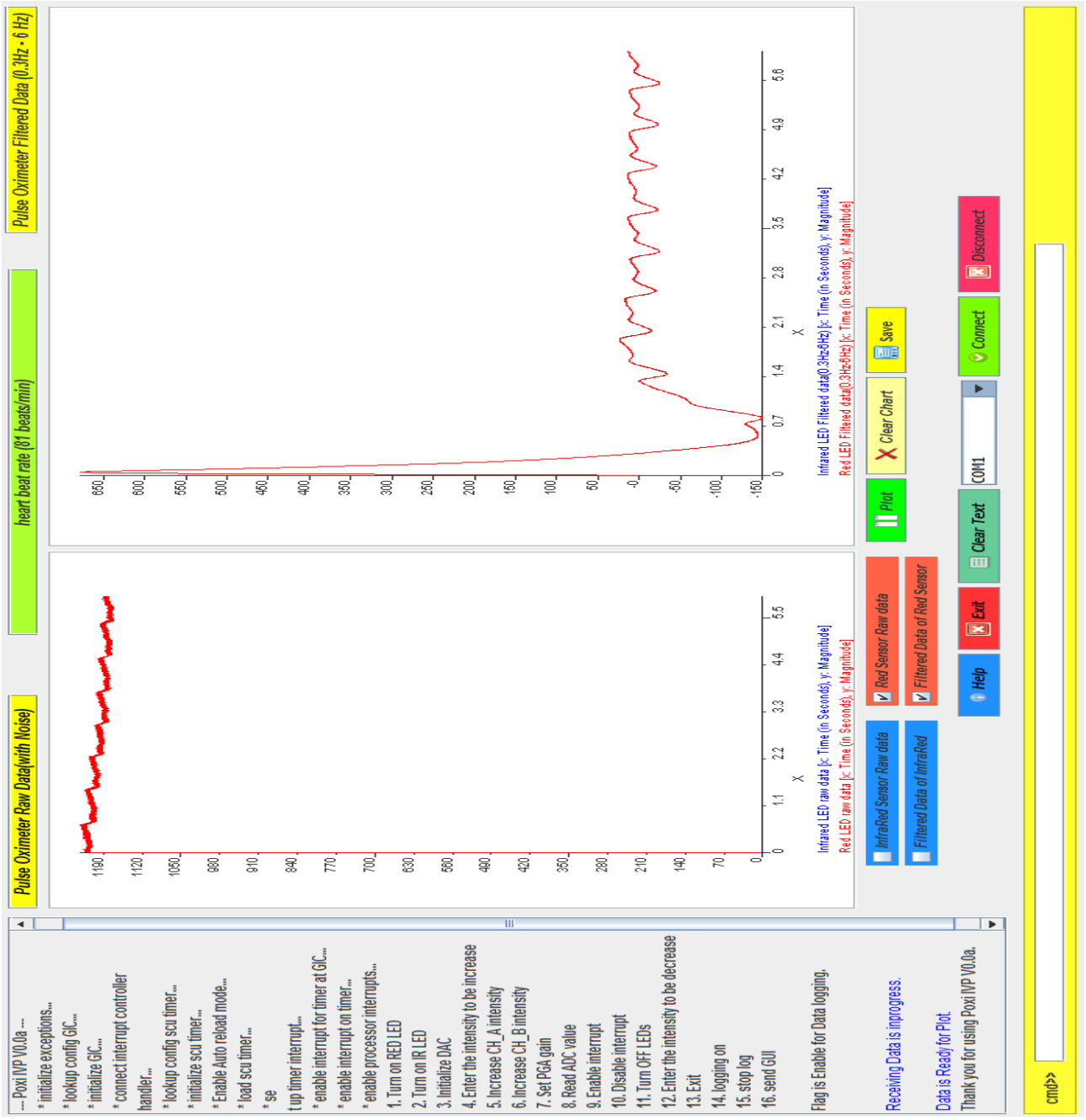


Figure 6.4: Pulse Oximeter Red Sensor output on GUI

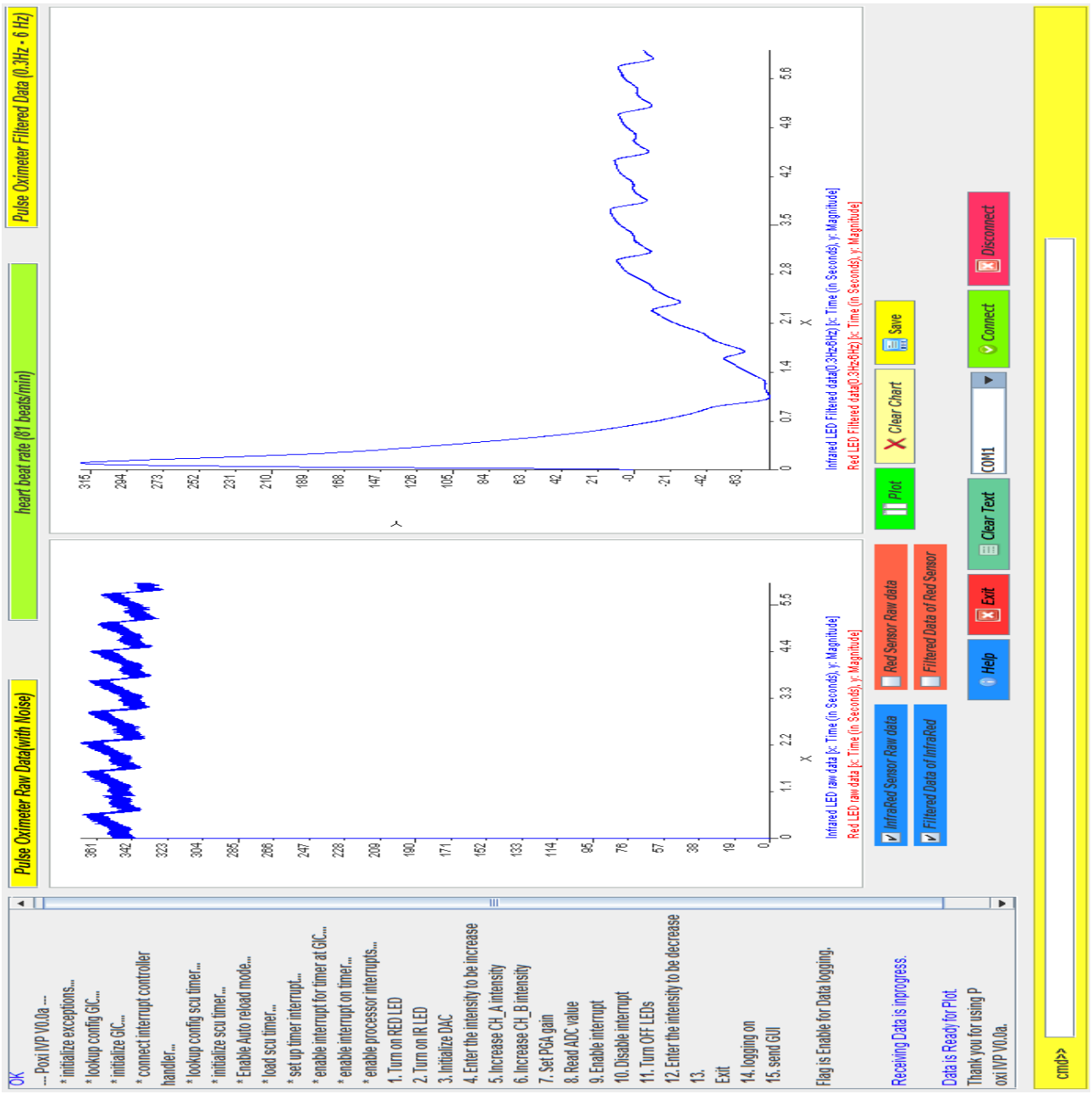


Figure 6.5: Pulse Oximeter InfraRed Sensor output on GUI

Above two figure shows the output results for both red and infrared sensors respectively. User can click on **"Save"** button to download selected data. The file name "poxi_data.dat" have been generated in local machine after downloading it.

Chapter 7

CONCLUSIONS

The project work covered the development of hardware SPI, software solution for DAC, PGA and ADC, DSP in MATLAB and C code, GUI configuration. The Java GUI application is intended to display data by the execution of commands sent from the POXI device, it serves as a display driven by the command communication.
The pulse oximeter application is accomplished by the usage of hardware, device drivers, digital filters, graphical user interface.

Future work

In future the following functionalities in the GUI can be made:

- Heart beat can be calculated from captured data using Zero crossing technic and display it to GUI.
- Spo2 calculation can be made from Red and IR AC,DC component and display result on GUI.

Bibliography

[1] Brand TM, Brand ME, Jay GD (February 2002) :
"Enamel nail polish does not interfere with pulse oximetry among normoxic volunteers". Journal of Clinical Monitoring and Computing.

[2] Severinghaus JW, Honda Y (April 1987) :
"History of blood gas analysis. VII. Pulse oximetry". Journal of Clinical Monitoring.

[3] Hypoxemia (low blood oxygen) :
https://www.mayoclinic.org/symptoms/hypoxemia/basics/definition/sym-20050930

[4] Pulse Oximeter measurement :
https://www.researchgate.net/publication/312279299_Pulse_Oximeter_Manufacturing_Wireless_Telemetry_for_Ventilation_Oxygen_Support/figures?lo=1

[5] SPI protocol:
https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/

[6] SPI Modes:
https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

[7] PMOD Connector and Interface board details:
Project Documentation, Prof. Dr.-Ing. Kai Mueller, October 2020
https://elli.hs-bremerhaven.de/goto.php?target=file_185084_download&client_id=elli

[8] DAC Data sheet:
https://www.analog.com/media/en/technical-documentation/data-sheets/AD5624_5664.pdf

[9] ADC Data sheet:
https://www.analog.com/media/en/technical-documentation/data-sheets/AD7887.pdf

[10] Artix-7 board Data sheet:
https://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf

Appendix A

```
----------------------------------------------------------------------------
-- Create Date:    11-01-2021
-- Created By: Dharmesh Patel(38073)
-- Module Name:    spifinal - Behavioral
-- Project Name: Pulse oximeter
-- Target Devices: FPGA
-- Description:    16,24 bit SPI state machine (for DAC,PGA and ADC)
----------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_arith.ALL;

ENTITY spifinal IS
GENERIC (DATA_LENGTH_BIT_SIZE   : INTEGER := 2;
         DATA_SIZE              : INTEGER := 32;
         BAUD_RATE_DIVIDER_SIZE : INTEGER := 8);
PORT ( sysclk         : IN STD_LOGIC;
reset                 : IN STD_LOGIC;
data_length           : IN STD_LOGIC_VECTOR (DATA_LENGTH_BIT_SIZE-1 DOWNTO 0);
baud_rate_divider     : IN STD_LOGIC_VECTOR (BAUD_RATE_DIVIDER_SIZE-1 DOWNTO 0);
clock_polarity        : IN STD_LOGIC;
clock_phase           : IN STD_LOGIC;
start_transmission    : IN STD_LOGIC;
transmission_done     : OUT STD_LOGIC;
data_tx   : IN  STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0);
data_rx   : OUT STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0) := (others => '0');
spi_clk   : OUT STD_LOGIC;
spi_MOSI  : OUT STD_LOGIC;
spi_MISO  : IN STD_LOGIC;
spi_cs    : OUT STD_LOGIC);
```

```
END spifinal;

ARCHITECTURE Behavioral OF spifinal IS

TYPE SPI_STATE_TYPE IS (SPI_IDLE, SPI_START, SPI_CLK_HI,SPI_CLK_LOW,SPI_LEADOUT);
SIGNAL current_state, next_state: SPI_STATE_TYPE;
SIGNAL clk_pulse : STD_LOGIC;
SIGNAL i_tx_buffer : STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0) :=(others => '0');
SIGNAL i_rx_buffer : STD_LOGIC_VECTOR (DATA_SIZE-1 DOWNTO 0) :=(others => '0');
SIGNAL counter : INTEGER RANGE 0 TO DATA_SIZE-1 := 0;

CONSTANT DATA_LENGTH_1 : INTEGER := 16;
CONSTANT DATA_LENGTH_2 : INTEGER := 24;

BEGIN

baud_rate_division_process: PROCESS (sysclk, reset,baud_rate_divider,clk_pulse)
VARIABLE baud_rate_counter : UNSIGNED (7 DOWNTO 0) := (others => '0');
BEGIN
IF rising_edge(sysclk) THEN
        clk_pulse <= '0';
IF reset = '1' OR current_state = SPI_IDLE THEN
        baud_rate_counter := (others => '0');
ELSIF baud_rate_divider = CONV_STD_LOGIC_VECTOR(baud_rate_counter,
                                BAUD_RATE_DIVIDER_SIZE) THEN
        baud_rate_counter := (others => '0');
        clk_pulse <= '1';
ELSE
        baud_rate_counter := baud_rate_counter + 1;
END IF;
END IF;
END PROCESS;

spi_switch_state : PROCESS (sysclk, current_state, next_state)
VARIABLE data_length_internal : UNSIGNED (1 DOWNTO 0);
BEGIN
IF rising_edge(sysclk) THEN
        IF reset = '1' THEN
```

```
            current_state <= SPI_IDLE;
            i_tx_buffer <= (others => '0');
            i_rx_buffer <= (others => '0');
            data_rx <= (others => '0');
    ELSIF next_state = SPI_START THEN -- Get redy immediately
            current_state <= SPI_START;
            data_rx <= (others => '0');
            i_rx_buffer <= (others => '0');
            CASE data_length IS
            WHEN "01" =>
            i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_1)
             <= data_tx(DATA_LENGTH_1-1 downto 0);
            counter <= DATA_LENGTH_1-1;
            WHEN "10" =>
            i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_2)
             <= data_tx(DATA_LENGTH_2-1 downto 0);
            counter <= DATA_LENGTH_2-1;
            WHEN OTHERS => NULL;
            END CASE;
    ELSIF clk_pulse = '1' THEN        -- Or Wait for the pulse

            IF clock_phase = '0' THEN
                    -- PUSH INPUT
                    IF next_state = SPI_CLK_HI THEN
                    i_rx_buffer <= i_rx_buffer(DATA_SIZE-2 DOWNTO 0) &
                                   spi_MISO;
                    END IF;
                    -- POP OUTPUT
                    IF next_state = SPI_CLK_LOW THEN
                    i_tx_buffer <= i_tx_buffer(DATA_SIZE-2 downto 0) & '-';
                    counter <= counter - 1;
                    END IF;
            END IF;

            IF clock_phase = '1' THEN
                    -- PUSH INPUT
                    IF current_state = SPI_CLK_HI THEN
                    i_rx_buffer <= i_rx_buffer(DATA_SIZE-2 DOWNTO 0) &
```

```
                        spi_MISO;
                        END IF;
                        -- POP OUTPUT
                        IF current_state = SPI_CLK_LOW AND
                        next_state = SPI_CLK_HI THEN
                        i_tx_buffer <= i_tx_buffer(DATA_SIZE-2 downto 0) & '-';
                        counter <= counter - 1;
                        END IF;
                END IF;

                IF current_state = SPI_LEADOUT THEN
                data_rx <= i_rx_buffer;
                END IF;

                current_state <= next_state;
        END IF;
END IF;
END PROCESS;


spi_mechanism :  process (next_state, clock_polarity, current_state,
 start_transmission, i_tx_buffer, clock_phase, counter)
BEGIN
next_state <= current_state;
spi_clk <= clock_polarity;
spi_cs <= '1';
spi_MOSI <= '0';
transmission_done <= '1';

CASE current_state IS
        WHEN SPI_IDLE =>
                IF start_transmission = '1' THEN
                        next_state <= SPI_START;
                END IF;
        WHEN SPI_START =>
                spi_cs <= '0';
                transmission_done <= '0';
                IF clock_phase = '0' THEN
```

```
                        spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                END IF;
                next_state <= SPI_CLK_HI;
        WHEN SPI_CLK_HI =>
                spi_cs <= '0';
                transmission_done <= '0';
                spi_clk <= NOT clock_polarity;
                spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                IF counter = 0 THEN
                        next_state <= SPI_LEADOUT;
                ELSE
                        next_state <= SPI_CLK_LOW;
                END IF;
        WHEN SPI_CLK_LOW =>
                spi_cs <= '0';
                transmission_done <= '0';
                spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                IF counter = 0 AND clock_phase = '1' THEN
                        next_state <= SPI_LEADOUT;
                ELSE
                        next_state <= SPI_CLK_HI;
                END IF;
        WHEN SPI_LEADOUT =>
                IF clock_phase = '1' THEN
                        spi_MOSI <= i_tx_buffer(DATA_SIZE-1);
                END IF;
                next_state <= SPI_IDLE;
                transmission_done <= '0';
                spi_cs <= '0';
END CASE;
END PROCESS;

END Behavioral;
```

Appendix B

```
/*
* poxivpmn.c: Poxi SW Solution file
* created by: Dharmesh Patel
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "xscutimer.h"
#include "sleep.h"

#define POXIREG(k) (*(unsigned int *)(XPAR_POXI4IF_0_S00_AXI_BASEADDR+4*k))
// 16th bit of this resister is used to check SPI done bit
#define SPI_DONE ((POXIREG(1)) & 0x00008000)
#define SPI_TX_BIT POXIREG(2) // used to transmit SPI data bits
#define ADC_VAL POXIREG(2) // used to read ADC conversion result
// 8,9,17,16,18,10,11 bits are used for CPOL,CPHA,CS,Data length
#define SPI_CONFIG POXIREG(1)
// 0th and 1st bits are used for Infrared and Red LED on/off
#define RedIred POXIREG(3)

#define ch_A 1 // DAC channels A,B,C,D (1,2,3,4)
#define ch_B 2
#define ch_C 3
#define ch_D 4
#define DAC 1 // To select device DAC,PGA,ADC (1,2,3)
#define PGA 2
```

```
#define ADC 3
#define Buffer_L 3000// Size of Internal Buffer

// following are the highpass and lowpass filter coefficients
#define BH_0 0.997337820133346
#define BH_1 -1.99467564026669
#define BH_2 0.997337820133346
#define AH_1 -1.99466855305249
#define AH_2 0.994682727480893
#define BL_0 0.00134871194835634
#define BL_1 0.00269742389671268
#define BL_2 0.00134871194835634
#define AL_1 -1.89346414636183
#define AL_2 0.898858994155252

typedef struct {
        double b0, b1, b2, a1, a2;
        double u_k1, u_k2, y_k1, y_k2;
        double y_out;
}Filterobj;

static Filterobj HP_f,LP_f;

static int R_size = 0, IR_size = 0;
static double IRbuffer_filt[Buffer_L],Rbuffer_filt[Buffer_L];
static short int IRbuffer[Buffer_L],Rbuffer[Buffer_L];
static double pofilt(double);
static void pofilt_init(void);
static short int ADC_Read(void);
static short int adc_reading = 0;
static double filt_out;
static short int sample = 0;
static int case_no = 0;
static unsigned int Logger_flag;

// ---- interrupt controller -----
static XScuGic  Intc;   // interrupt controller instance
static XScuGic_Config  *IntcConfig;     // configuration instance
```

```
// ---- scu timer -----
static XScuTimer  pTimer;       // private Timer instance
static XScuTimer_Config  *pTimerConfig; // configuration instance
// 100Hz => 3333333
#define TIMER_LOAD_VALUE  166666 // 666666 // 2ms for Red/IRd

static volatile unsigned int ISR_Count;

/*
* ------------------------------------------------------------
* Interrupt handler (ZYNQ private timer)
* ------------------------------------------------------------
*
* ------------------------------------------------------------
* Step 1: Turn ON Red LED
* Step 2: Read ADC value and update it to internal buffer and Turn OFF LEDs
* Step 3: Turn ON Infrared LED
* Step 4: Read ADC value and update it to internal buffer and Turn OFF LEDs
* ------------------------------------------------------------
*/
static void TimerIntrHandler(void *CallBackRef)
{
        XScuTimer *TimerInstance = (XScuTimer *)CallBackRef;

        XScuTimer_ClearInterruptStatus(TimerInstance);

        switch(ISR_Count)
        {
        case 0:
                RedIred = 0b01; // 24th bit is used for red LED in HW
                ISR_Count++;
                break;
        case 1:
                adc_reading = ADC_Read();
                filt_out = pofilt(adc_reading);
                RedIred = 0;
                if (R_size < Buffer_L)
```

```
                {
                Rbuffer_filt[R_size] = filt_out;
                Rbuffer[R_size] = adc_reading;
                R_size++;
                }
                ISR_Count++;
                break;
        case 2:
                RedIred = 0b10;// 25th bit is used for Infrared LED in HW
                ISR_Count++;
                break;
        case 3:
                adc_reading = ADC_Read();
                filt_out = pofilt(adc_reading);
                RedIred = 0;
                if (IR_size < Buffer_L)
                {
                IRbuffer_filt[IR_size] = filt_out;
                IRbuffer[IR_size] = adc_reading;
                IR_size++;
                }
                ISR_Count = 0;
                break;
        default:;
        }

}

static void pofilt_init() // To initialize filter objects
{
        HP_f.b0 = BH_0;
        HP_f.b1 = BH_1;
        HP_f.b2 = BH_2;
        HP_f.a1 = AH_1;
        HP_f.a2 = AH_2;
        HP_f.u_k1 = 0.0;
        HP_f.u_k2 = 0.0;
        HP_f.y_k1 = 0.0;
```

```
        HP_f.y_k2 = 0.0;
        HP_f.y_out = 0.0;
        LP_f.b0 = BL_0;
        LP_f.b1 = BL_1;
        LP_f.b2 = BL_2;
        LP_f.a1 = AL_1;
        LP_f.a2 = AL_2;
        LP_f.u_k1 = 0.0;
        LP_f.u_k2 = 0.0;
        LP_f.y_k1 = 0.0;
        LP_f.y_k2 = 0.0;
        LP_f.y_out = 0.0;
}
//Filter for poxi data (High pass 0.3 Hz and Low pass 6 Hz)
static double pofilt(double u)
{
        HP_f.y_out = HP_f.b0*u + HP_f.b1*HP_f.u_k1 + HP_f.b2*HP_f.u_k2
        - HP_f.a1*HP_f.y_k1 - HP_f.a2* HP_f.y_k2;
        HP_f.u_k2 = HP_f.u_k1;
        HP_f.u_k1 = u;
        HP_f.y_k2 = HP_f.y_k1;
        HP_f.y_k1 = HP_f.y_out;
        LP_f.y_out = LP_f.b0*HP_f.y_out + LP_f.b1*LP_f.u_k1 + LP_f.b2*LP_f.u_k2
        - LP_f.a1*LP_f.y_k1 - LP_f.a2* LP_f.y_k2;
        LP_f.u_k2 = LP_f.u_k1;
        LP_f.u_k1 = HP_f.y_out;
        LP_f.y_k2 = LP_f.y_k1;
        LP_f.y_k1 = LP_f.y_out;

        return LP_f.y_out;
}

// Function for DAC configuration
static void DAC_config()
{
        while(!SPI_DONE);
        SPI_CONFIG = 0x00010a10; // (Data Length = 24("10"),CPOL = 1, CPHA = 0)
        SPI_TX_BIT = 0x00280000; // Reset DAC internal registers and channels
```

```
        while(!SPI_DONE);
        SPI_CONFIG = 0x00010a10;
        SPI_TX_BIT = 0x0020000F; // Power-up DAC channels
        while(!SPI_DONE);
        SPI_CONFIG = 0x00010a10;
        SPI_TX_BIT = 0x00300000; // setting-up LDAC register
        while(!SPI_DONE);
        SPI_CONFIG = 0x00010a10;
        SPI_TX_BIT = 0x00380001; // setting-up DAC internal reference
}

// DAC write Function based on input Channel and Value
static void DAC_Write(int chan, int val)
{
        unsigned int xval;

        xval = (val & 0x0fff) << 4;
        if (chan == ch_A) {
                xval |= 0x0180000;
        } else if (chan == ch_B) {
                xval |= 0x0190000;
        } else if (chan == ch_C) {
                xval |= 0x01a0000;
        } else if (chan == ch_D) {
                xval |= 0x01b0000;
        }

        while(!SPI_DONE);
        SPI_CONFIG = 0x00010a10;
        SPI_TX_BIT = xval;
}

// PGA write Function to Set Gain
static void PGA_Write()
{
        while(!SPI_DONE);
        SPI_CONFIG = 0x00020710;// (Data Length = 16("01"),CPOL = 1, CPHA = 1)
        SPI_TX_BIT = 0x4004; // Set PGA gain to 8 (gain register:="100")
```

```
}

// ADC Read Function, Conversion value will be updated to ADC_VAL resister
// after transmission is done
static short int ADC_Read()
{
        while(!SPI_DONE);
        SPI_CONFIG = 0x00030680;// (Data Length = 16("01"),CPOL = 1, CPHA = 0)
        SPI_TX_BIT = 0b0000000100000000;
        while(!SPI_DONE);
        return 0x0fff & ADC_VAL;
}

// clear outputs and turn off Interface devices on exit
static void ClearAll()
{
        RedIred = 0; // Turn OFF LEDs
        while(!SPI_DONE);
        SPI_CONFIG = 0x00010a10;// (Data Length = 24("01"),CPOL = 1, CPHA = 0)
        SPI_TX_BIT = 0x0020003F;// Power-Down DAC channels
        while(!SPI_DONE);
        SPI_CONFIG = 0x00030680;// (Data Length = 16("01"),CPOL = 1, CPHA = 0)
        SPI_TX_BIT = 0b0010000100000000;  // Power-Down ADC channels
        while(!SPI_DONE);
        SPI_CONFIG = 0x00020710;// (Data Length = 16("01"),CPOL = 1, CPHA = 0)
        SPI_TX_BIT = 0x2000;
        SPI_CONFIG = 0;
        SPI_TX_BIT = 0;
        RedIred = 0;
        R_size = 0;
        IR_size = 0;
}

static void TransferTo_GUI(){
if (Logger_flag){
// For logging data start string with "~"
        if(sample<Buffer_L)
        {
```

```
        printf("~ %f %f %f %f\n",Rbuffer[sample],IRbuffer[sample],
                Rbuffer_filt[sample],IRbuffer_filt[sample]);
        sample++;
        }
        else
        {
        printf("@ %f\n",13.000);
        }
}
}
// To initialize Interrupt handler function
static void Poxi_initInterruptHandlers(void)
{
        printf(" * initialize exceptions...\n\r");
        Xil_ExceptionInit();

        printf(" * lookup config GIC...\n\r");
        IntcConfig = XScuGic_LookupConfig(XPAR_SCUGIC_0_DEVICE_ID);
        printf(" * initialize GIC...\n\r");
        XScuGic_CfgInitialize(&Intc, IntcConfig, IntcConfig->CpuBaseAddress);

        // Connect the interrupt controller interrupt handler to the hardware
        printf(" * connect interrupt controller handler...\n\r");
        Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
        (Xil_ExceptionHandler)XScuGic_InterruptHandler, &Intc);

        printf(" * lookup config scu timer...\n\r");
        pTimerConfig = XScuTimer_LookupConfig(XPAR_XSCUTIMER_0_DEVICE_ID);
        printf(" * initialize scu timer...\n\r");
        XScuTimer_CfgInitialize(&pTimer, pTimerConfig, pTimerConfig->BaseAddr);
        printf(" * Enable Auto reload mode...\n\r");
        XScuTimer_EnableAutoReload(&pTimer);
        printf(" * load scu timer...\n\r");
        XScuTimer_LoadTimer(&pTimer, TIMER_LOAD_VALUE);

        printf(" * set up timer interrupt...\n\r");
        XScuGic_Connect(&Intc, XPAR_SCUTIMER_INTR,
        (Xil_ExceptionHandler)TimerIntrHandler,
```

```
        (void *)&pTimer);
        printf(" * enable interrupt for timer at GIC...\n\r");
        XScuGic_Enable(&Intc, XPAR_SCUTIMER_INTR);
        printf(" * enable interrupt on timer...\n\r");
        XScuTimer_EnableInterrupt(&pTimer);

        // Enable interrupts in the Processor.
        printf(" * enable processor interrupts...\n\r");
        Xil_ExceptionEnable();
}

int main()
{
        init_platform();
        print("--- Poxi IVP V0.0a ---\n\r");
        double redarray[3000]; // just for Testing
        ISR_Count = 0;
        int case_no = 0,rc = 0,intensity = 0, n;
        Logger_flag = 0;

        Poxi_initInterruptHandlers();

        printf(" 1. Turn on RED LED\n 2. Turn on IR LED\n 3. Initialize DAC\n
        4. Enter the intensity to be increase\n 5. Increase CH_A intensity\n");
        printf(" 6. Increase CH_B intensity\n 7. Set PGA gain\n
        8. Read ADC value\n 9. Enable interrupt\n
        10. Disable interrupt\n");
        printf(" 11. Turn OFF LEDs\n 12. Enter the intensity to be decrease\n
        13. Exit\n 14. logging on\n 15. send to GUI\n");

        while(!(rc == 10))
        {
          if(rc!= 11)
          {
                printf("btst>> ");
                scanf("%d",&case_no);
          }
                switch(case_no)
```

```
{
case 1: printf("1. RED LED is ON\n");
RedIred = 0b10;
break;
case 2: printf("2. IR LED is ON\n");
RedIred = 0b01;
break;
case 3: printf("3. Initialized DAC\n");
DAC_config();
break;
case 4: printf("4. Enter the intensity to be increased\n");
scanf("%d",&n);
intensity = intensity + n;
break;
case 5: printf("5. Set CH_A intensity by %d\n",intensity);
DAC_Write(ch_A,intensity);
break;
case 6: printf("6. Set CH_B intensity by %d\n",intensity);
DAC_Write(ch_B,intensity);
break;
case 7: printf("7. Set PGA gain by 2\n");
PGA_Write(1);
break;
case 8: for(int i=0;i<10;i++) // just for Testing
{
        redarray[i] = ADC_Read();
        printf("ADC value is: %lf\n", redarray[i]);
}
case_no = 13;
break;
case 9: printf("9. Interrupt is Enabled\n");
printf(" * start timer...\n\r");
XScuTimer_Start(&pTimer);
do
{
if(IR_size==Buffer_L)
{
rc = 11;
```

```
                case_no = 10;
                }
                }while(case_no!=10);
                break;
                case 10: printf("10. Interrupt is Disabled\n");
                XScuTimer_Stop(&pTimer);
                rc = 9;
                break;
                case 11: printf("11. LEDs are Turned off\n");
                RedIred = 0b00;
                break;
                case 12: printf("12. Enter the intensity to be decrease\n");
                scanf("%d",&n);
                intensity = intensity - n;
                break;
                case 13: printf("13. Exit\n");
                ClearAll();
                Logger_flag = 0;
                rc = 10;
                break;
                case 14: // To start sending data to GUI for logging
                Logger_flag = 1;
                printf("Flag is Enable for Data logging.\n");
                break;
                case 15: TransferTo_GUI();
                break;
                default:printf("Entered selection is out of the List:\n");
                }
        };
        Xil_ExceptionDisable();
        XScuTimer_DisableInterrupt(&pTimer);
        XScuGic_Disable(&Intc, XPAR_SCUTIMER_INTR);
        usleep(200000);
        print("Thank you for using Poxi IVP V0.0a.\n\r");
        cleanup_platform();
        return 0;
}
```