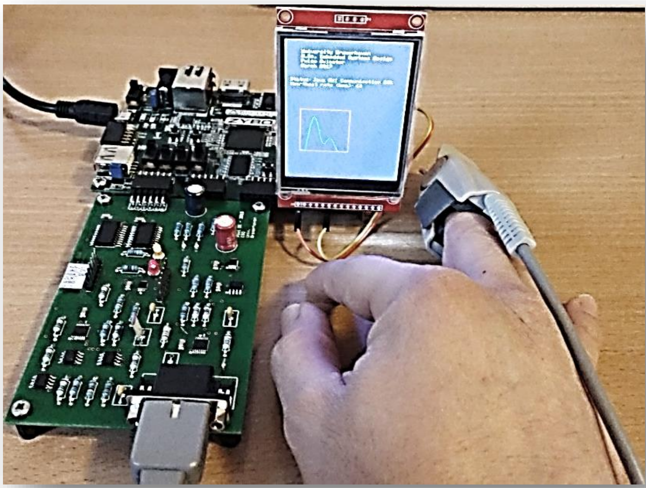
Hochschule Bremerhaven

M.Sc. Embedded system design.

Hardware Software base libraries for pulse oximeter



Students:

Yarib Nevárez 34399

Supervisor:

Prof. Dr.-Ing. Kai Müller

Abstract:

Embedded computer systems have driven many advances in almost every aspect of life, including medical applications. In many cases, these applications require data acquisition, digital signal processing (DSP), graphical user interfaces (GUI), communication protocols and more. This work achieves the implementation of a pulse oximeter through the usage of a base hardware and software libraries. The focus of this work is the development of these base libraries, it can be described as a generic hardware-software application framework adequate for rapid and reliable development.

The Hardware-software application framework covers the implementation of a configurable enhanced SPI hardware, DSP software libraries and its automated design and simulation in MATLAB Simulink with legacy code, device drivers for analogue to digital converters (ADC), digital to analogue converter (DAC), programmable gain amplifier (PGA), colour display, and high level software libraries for graphics, UI widget components, and a serial communication link layer.

The software libraries are completely written in C language using object oriented programming (C++ style), software design patterns, and data structures. Dynamic memory allocation is used and extendable for a memory management unit, the memory handling approach is free of memory leaks and NULL pointer exceptions.

23 September 2017

Contents

List of figures

List of tables

1 INTRODUCTION

Pulse oximetry is about measuring the 'Oxygen Saturation' (the total percentage of total hemoglobin that is carrying oxygen) in blood.

Hemoglobin bound to oxygen is called oxygenated hemoglobin (HbO2). Hemoglobin not bound to oxygen is called deoxygenated hemoglobin (Hb). The oxygen saturation is the ratio of the oxygenated hemoglobin to the hemoglobin in the blood, as defined by the following equation.

$$\text{Oxygen saturation} = \frac{\text{C (HbO2)}}{\text{C (HbO2)} + \text{C (Hb)}} \times 100\ (\%)$$

C (Hb) = Concentration of deoxygenated hemoglobin

C (HbO2) = Concentration of oxygenated hemoglobin

This project is an implementation of a Pulse Oximeter using ZYBO board to achieve the required functionality. The ZYBO (ZYnq BOard) is a feature-rich SoC development platform built around the smallest member of the Xilinx Zynq-7000 family, the Z-7010.

https://windward.hawaii.edu/facstaff/miliefsky-m/ZOOL%20142L/aboutPulseOximetry.pdf

1.1 HISTORY OF PULSE OXIMETER

Pulse oximeters are in common use because they are:

- Non-invasive
- Economical and reliable
- Can be very compact
- Detects hypoxemia earlier than using your eyes to see cyanosis

Oximetry is the measurement of transmitted light through a translucent measuring site to determine a patient's oxygen status non-invasively. Oximetry measurements can be traced to the early 1930's when German investigators used spectrophotometers (instruments that measure different wavelengths and intensities of light) to research light transmission through human skin.

The following briefly outlines the development of this important device.

1864: Geory Gabriel Stokes discovered that hemoglobin is the oxygen carrier in blood.

1935: Matthes developed the first oxygen saturation meter. It used a 2-wavelength light source with red and green filters, which was later changed to red and infrared filters.

1941: "Oximetry testing" is first used to measure oxygen saturation level with a pulse oximeter.

1940's: Millikan, a British scientist, used a dual light source to create the first practical aviation ear oxygen meter. During Second World War, many pilots were saved from under pressurized cabins by using oximetry testing.

1964: Hewlett Packard built the first ear oximeter by using eight wavelengths of light. The oximeter was used primary in sleep laboratories and in pulmonary functions. The unit was expensive, clumsy, and large.

1972: Takuo Aoyagi, a Japanese bio-engineer at Nihon Kohden, developed a pulse oximeter based on the ratio of red to infrared light absorption in blood. He obtained a Japanese patent. Another Japanese research, Minolta, obtained an US patent based on the same concept. Oximetry became clinically feasible.

1981: Biox introduced the first commercial pulse oximeter. Initially it was focused on respiratory care and later expanded into operating rooms. Since then, other manufacturers have entered the market and the pulse oximeter technology has improved significantly.

1987: Pulse Oximetry becomes part of a standard procedure in administrating general anesthetic in US. The use of oximetry quickly spread to other hospital units, such as emergency rooms, recovery rooms, neonatal units, and intensive care units.

1995: Fingertip pulse oximeters first appeared on the market.

2000: Medicare accepted physicians billing for in-office oximeter readings.

2007: FDA published a notice in Federal Register (Vol. 72, No. 138 / Thursday, July 19, 2007) titled "Draft Guidance for Industry and Food and Drug Administration Staff; Pulse Oximeters Premarket Notification Submissions [510(k)s]; Availability" for comment by October, 2007. Shortly after, FDA approved pulse oximeters appeared on the market.

Note: When arterial Oxyhemoglobin saturation is measured by an arterial blood gas it is referred to as SaO2. When arterial Oxyhemoglobin saturation is measured non-invasively by pulse Oximetry, it is referred to as SpO2.

https://www.amperordirect.com/pc/help-pulse-oximeter/z-pulse-oximeter-history.html

1.2 Background

To understand how pulse oximeter works, let us understand how human body is involved with physiological processes and what is oxygen saturation related to our project.

1.2.1 Blood oxygenation

Oxygen enters the lungs and then is passed on into blood. The blood carries the oxygen to the various organs in our body. The main way oxygen is carried in our blood is by means of hemoglobin. This process of supplying oxygen to all part of blood is called blood oxygenation.

Both oxygen and carbon dioxide are transported around the body in the blood through arteries, veins and capillaries. They bind to hemoglobin in red blood cells, although oxygen does so more effectively. Carbon dioxide also dissolves in the plasma or combines with water to form bicarbonate ions (HCO−3). This reaction is catalyzed by the carbonic anhydrase enzyme in red blood cells. The main respiratory surface in humans is the alveoli, which are small air sacs branching off from the bronchioles in the lungs. They are one cell thick and provide a moist and extremely large surface area for gas exchange to occur. Capillaries carrying deoxygenated blood from the pulmonary artery run across the alveoli. They are also extremely thin, so the total distance gases must diffuse across is only around 2 cells thick. An adult male has about 300 million alveoli, each ranging in diameter from 75 to 300 µm.

Inhaled oxygen can diffuse into the capillaries from the alveoli, while CO2 from the blood diffuses in the opposite direction into the alveoli. The waste CO2 can then be exhaled out of the body. Continuous blood flow in the capillaries and constant breathing maintain a steep concentration gradient. This complete process is called the Alveolus Gas Exchange shown below.

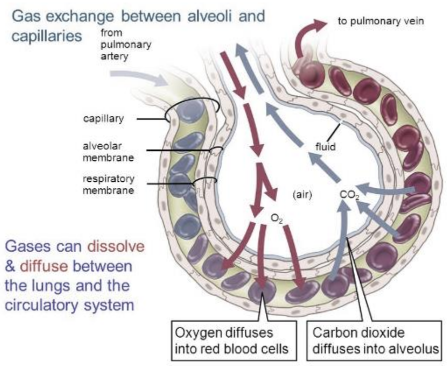


Figure 1. Alveolus Gas Exchange

1.2.2 Blood circulation

The circulatory system, also called the cardiovascular system or the vascular system, is an organ system that permits blood to circulate and transport nutrients (such as amino acids and electrolytes), oxygen, carbon dioxide, hormones, and blood cells to and from the cells in the body to provide nourishment and help in fighting diseases, stabilize temperature and pH, and maintain homeostasis**.** In the systemic circulation, the left ventricle pumps oxygen-rich blood into the main artery (aorta). The blood travels from the main artery to larger and smaller arteries into the capillary network. There the blood releases oxygen, nutrients and other important substances and takes on carbon dioxide and waste substances. The blood, which is now low in oxygen, is now collected in veins and travels to the right atrium and into the right ventricle.

Now pulmonary circulation starts: The right ventricle pumps blood that carries little oxygen into the pulmonary artery, which branches off into smaller and smaller arteries and capillaries. The capillaries form a fine network around the pulmonary vesicles, grape-like air sacs at the end of the airways. This is where carbon dioxide is released from the blood into the air contained in the pulmonary vesicles and fresh oxygen enters the bloodstream. When we breathe out, carbon dioxide leaves our body. Oxygen-rich blood travels through the pulmonary vein and the left atrium into the left ventricle. The next heart beat starts a new cycle of systemic circulation.

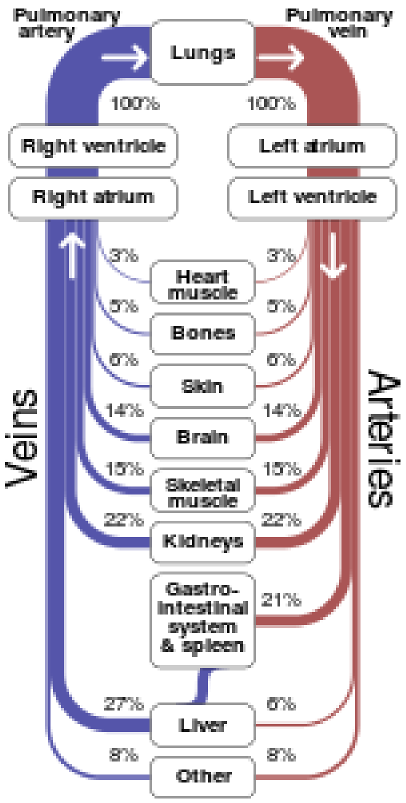


Figure 2. Blood Circulation Diagram

1.3 Implementation Methods

Pulse oximeters are of two operating kinds: Transmission and reflection modes.

In transmission-mode the emitter and photodetector are opposite of each other with the measuring site in-between. The light can then pass through the site.

In reflection-mode or backscatter type pulse oximetry, the emitter and photodetector are next to each other on top the measuring site. The light bounces from the emitter to the detector across the site.

This arrangement allows for measuring SpO_2 from multiple convenient locations on the body (e.g. the head, torso, or upper limbs), where conventional transmission-mode measurements are not feasible. For this reason, non-invasive reflectance pulse oximetry has recently become an important new clinical technique with potential benefits in fetal and neonatal monitoring.

The transmission method is the most common type used and for this discussion the transmission method will be implied.

Figure 3. SpO2 sensor

1.4 BEER – LAMBERT LAW

The Beer-Lambert law (or Beer's law) is the linear relationship between absorbance and concentration of an absorbing species.

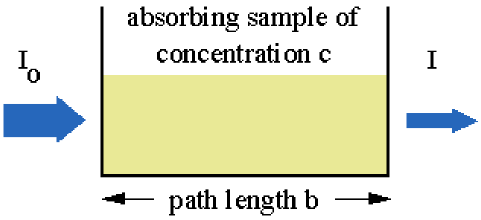


Figure 4. Absorption according to BLL

The general Beer-Lambert law is usually written as:

$$A = a(\lambda) * b * c$$

Where **A** is the measured absorbance, **a** **(λ)** is a wavelength-dependent absorptivity coefficient, **b** is the path length, and **c** is the analyte concentration.

In physics, the Beer-Lambert law has very strict criteria to be accurate. The blood is not an ideal liquid, it is irregular and makes the light scatter instead of going through straight line. So, Beer-Lambert's law cannot be applied strictly instead the method of finding saturation with Deoxy hemoglobin (Hb) and Oxyhemoglobin (HbO2) is more accurate.

1.5 FUNDAMENTAL WORKING PRINCIPLE

Pulse oximeter sensors have red and infrared low voltage light emitting diodes (LEDs) which serve as light sources. The emitted light is transmitted through the tissue, then detected by the photodetector and sent to the microprocessor of the pulse oximeter. All constituents of the human body, venous and arterial blood, and tissue absorb light (Figure 4). The pulsating of arterial blood results in changes in the absorption to to deoxygenated haemoglobin (Hb) and oxygenated haemoglobin (HbO2) in the path of the light. Since HbO2 and Hb absorb light to varying degrees, this varying absorption is translated into plethysmographic waveforms at both red and infrared wavelengths (Figure 5). The amount of light received by the detector indicates the amount of oxygen bound to the haemoglobin in the blood. Oxygenated haemoglobin (oxyhemoglobin or HbO2) absorbs more infrared light than red light. Deoxygenated haemoglobin (Hb) absorbs more red light than infrared light. By comparing the amounts of red and infrared light received, the instrument can calculate the SpO2 reading. For example, when the plethysmographic amplitude ad 660nm and 910nm are equal and the ratio R/IR=1, the SpO2 is approximately 85% (Figure 6).

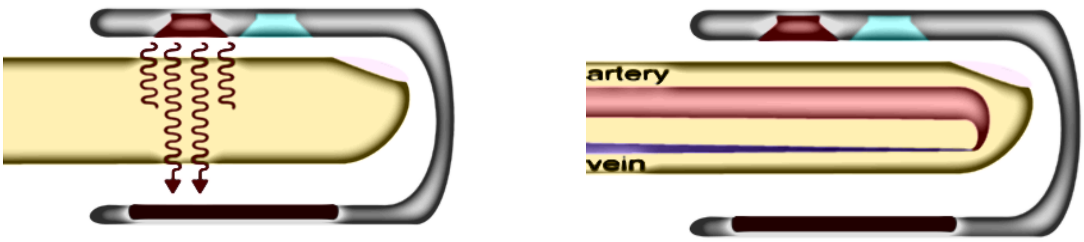


Figure 5. Vein, artery and tissue absorbing the light

The amount of arterial blood does change over short periods of time due to pulsation (although there is some constant level of arterial blood). Because the arterial blood is usually the only light absorbing component which is changing over short periods of time, it can be isolated from the other components. The amount of light absorbed depends on the following:

1. Concentration of the light absorbing substance.
2. Length of the light path in the absorbing substance
3. Oxygenated haemoglobin and deoxygenated haemoglobin absorbs red and infrared light differently.

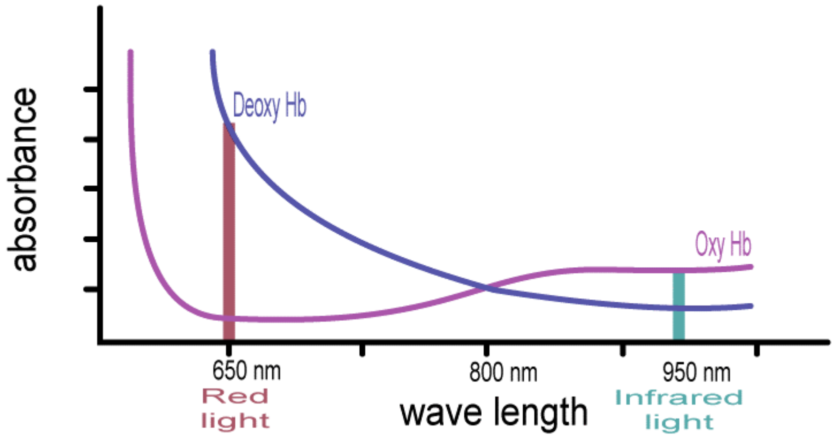


Figure 6. Absorption of red and infrared light at different wavelengths

1.6 Measuring pulse rate

When your heart beats it pumps blood through your body. During each heartbeat, the blood gets squeezed into capillaries, whose volume increases very slightly. Between heartbeats, the volume decreases. This change in volume affects the amount of light, such as the amount of red or infrared light, that will transmit through the tissue. Though this fluctuation is very small, it can be measured by a pulse oximeter using the same type of setup that is employed to measure blood oxygen saturation.

Heart beat rate can be measured by the time (T) in seconds, between two consecutive pulses, and converting the time into beats/min, using the formula beat/min = 60/T.

2 Implementation

The pulse oximeter is a hardware and software implementation, the hardware part is implemented in the programmable logic of a SoC, and it is intended to give proper interface communication to the ADC, DAC, and PGA devices; the software part is intended to control the devices and obtain the desired information, and present it to the user.

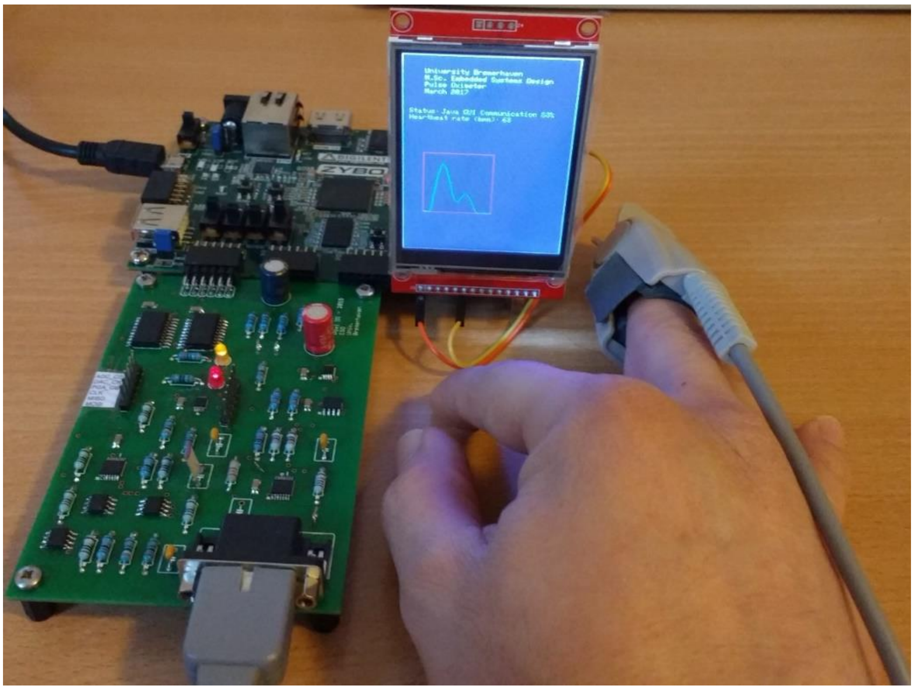


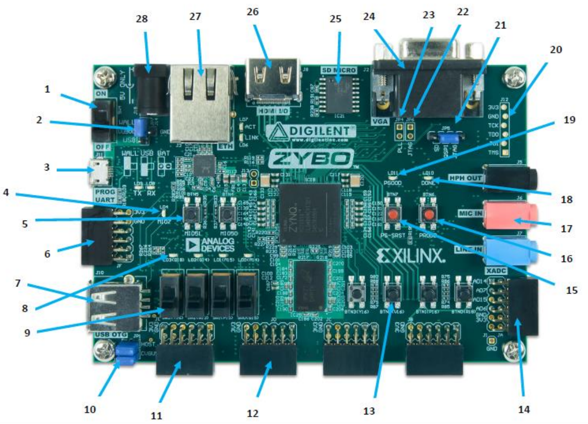
Figure 7. POXI implementation

2.1 HARDWARE

2.1.1 ZYBO FPGA SoC board

The ZYBO (ZYnq BOard) is a feature-rich, ready-to-use, entry-level embedded software and digital circuit development platform built around the smallest member of the Xilinx Zynq-7000 family, the Z-7010. The Z-7010 is based on the Xilinx All Programmable System-on-Chip (AP SoC) architecture, which tightly integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic. When coupled with the rich set of multimedia and connectivity peripherals available on the ZYBO, the Zynq Z-7010 can host a whole system design. The on-board memories, video and audio I/O, dual-role USB, Ethernet, and SD slot will have your design up-and-ready with no additional hardware needed. Additionally, six Pmod ports are available to put any design on an easy growth path. (Digilent, 2016)

Table 1. ZYBO Device Diagram.



Callout	Component Description	Callout	Component Description
1	Power Switch	15	Processor Reset Pushbutton
2	Power Select Jumper and battery header	16	Logic configuration reset Pushbutton
3	Shared UART/JTAG USB port	17	Audio Codec Connectors
4	MIO LED	18	Logic Configuration Done LED
5	MIO Pushbuttons (2)	19	Board Power Good LED
6	MIO Pmod	20	JTAG Port for optional external cable
7	USB OTG Connectors	21	Programming Mode Jumper
8	Logic LEDs (4)	22	Independent JTAG Mode Enable Jumper
9	Logic Slide switches (4)	23	PLL Bypass Jumper
10	USB OTG Host/Device Select Jumpers	24	VGA connector
11	Standard Pmod	25	microSD connector (Reverse side)
12	High-speed Pmods (3)	26	HDMI Sink/Source Connector
13	Logic Pushbuttons (4)	27	Ethernet RJ45 Connector
14	XADC Pmod	28	Power Jack

As software development platform (for hardware development and VHDL implementation) it was used Vivado v2016.3 (64-bit) provided by Xilinx. For software development it was used Xilinx SDK v2016.3.

2.1.2 Base POXI hardware platform

The pulse oximeter project was developed based on the provided hardware platform. This base hardware platform consist of PMOD connector, signal buffers, DAC, ADC, MOSFET H-Bridge driver, Intensity control, PGA, TIA, and finger clip.

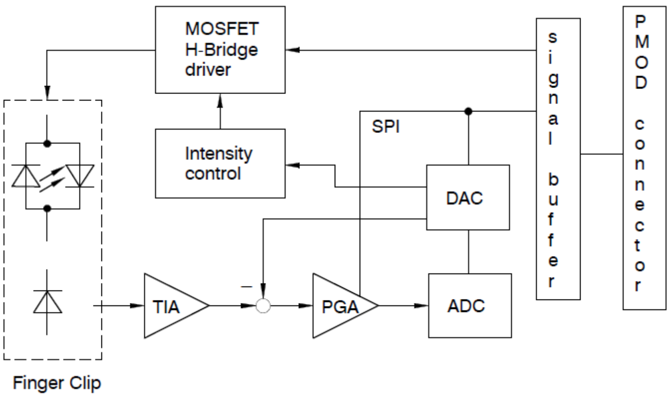


Figure 8. POXI Block diagram

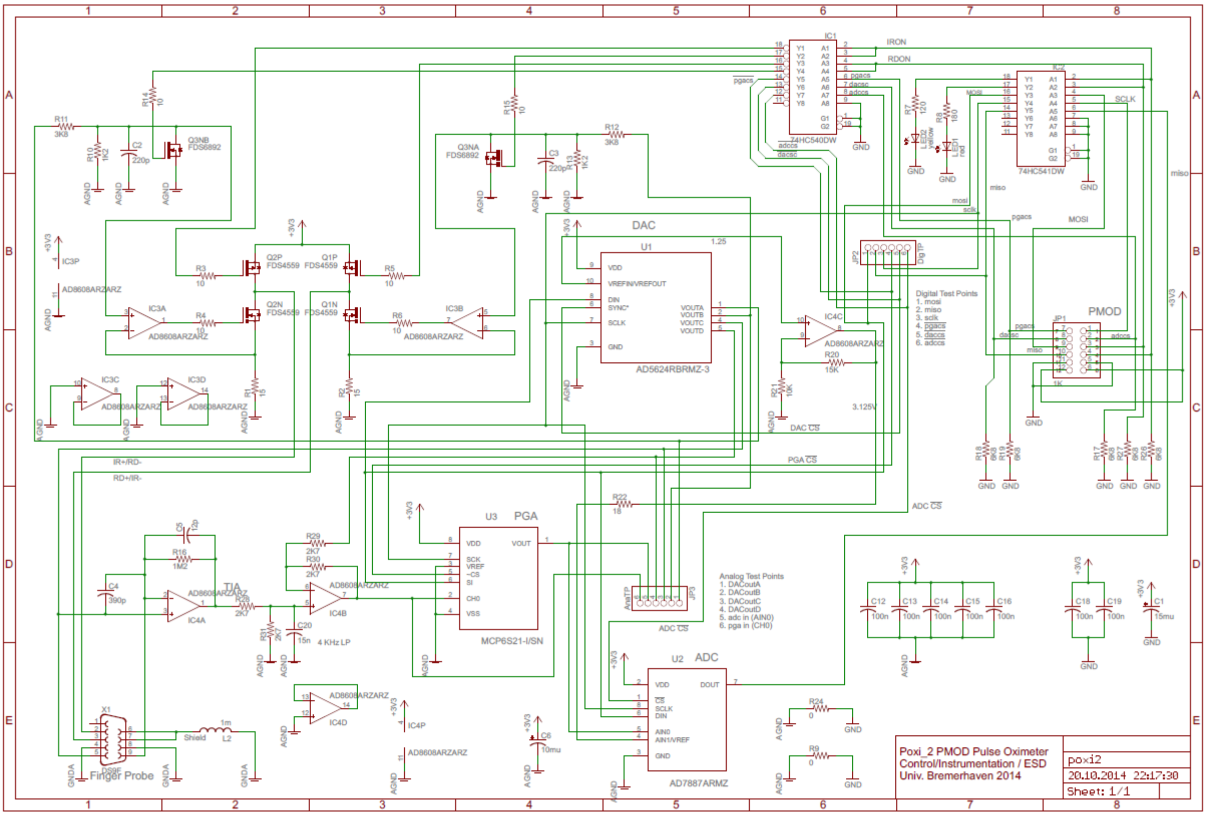


Figure 9. POXI board

For detailed information regarding POXI board and its features it can be referred in “espro_all” and ESD university documentation.

2.1.3 POXI SoC design

The POXI SoC design is basically built up by two peripheral interfaces, an AXI bus, and a processing system.

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

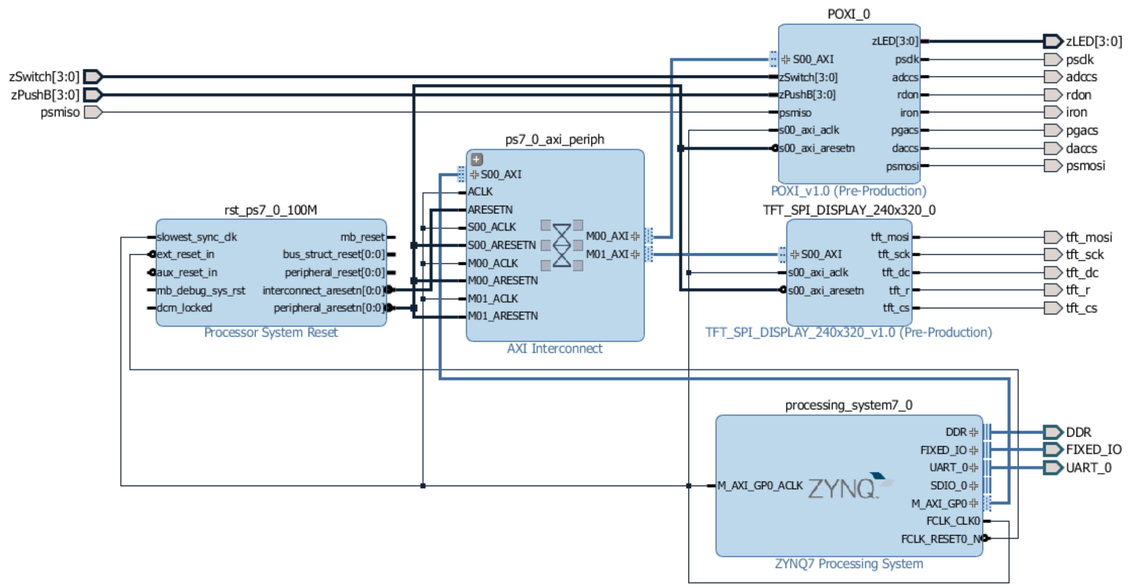


Figure 10. SoC design

2.1.4 IP Component "POXI"

Custom IP, target language VHDL, AXI peripheral 4 registers (32 bits each). The POXI_0 (as shown in figure 3) block is intended to establish communication and control to the base POXI Board and ZYBO. 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, ZYBO LEDs drivers, ZYBO switches and pushbuttons.

The figure below shows detailed information of the POXI ports and interfaces.

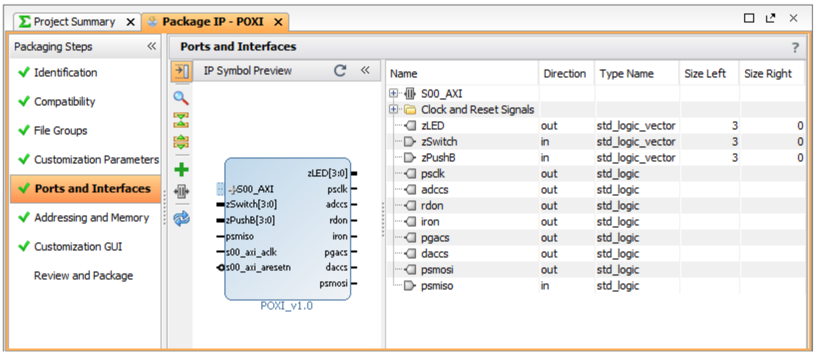


Figure 11. POXI - IP component interface

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 "spi.vhd" contains the design for the SPI protocol, and "POXI_v1_0_S00_AXI.vhd" implements and instance for SPI communication and the rest of the logic, "POXI_v1_0.vhd" can be considered as wrapper-interface for the AXI peripheral.

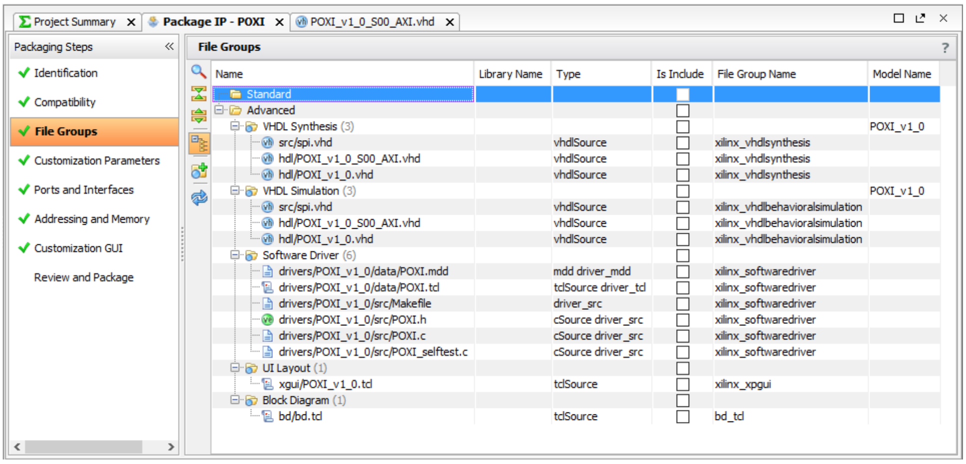
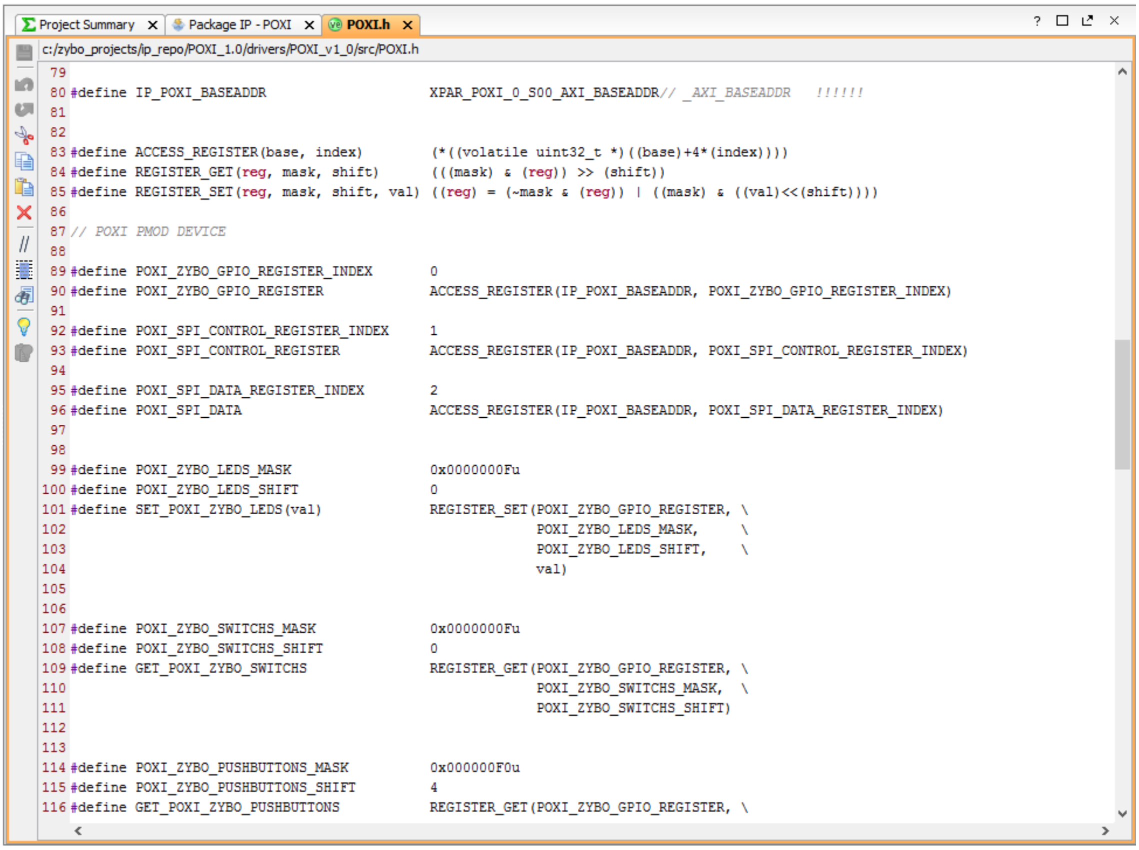


Figure 12. POXI IP file groups

For software interface it was written "POXI.h" which contains the C/C++ macros for setting and getting register data of the custom peripheral.



```
#define IP_POXI_BASEADDR                  XPAR_POXI_0_S00_AXI_BASEADDR// _AXI_BASEADDR   !!!!!!


#define ACCESS_REGISTER(base, index)       (*((volatile uint32_t *)((base)+4*(index))))
#define REGISTER_GET(reg, mask, shift)     (((mask) & (reg)) >> (shift))
#define REGISTER_SET(reg, mask, shift, val) ((reg) = (~mask & (reg)) | ((mask) & ((val)<<(shift))))

// POXI PMOD DEVICE

#define POXI_ZYBO_GPIO_REGISTER_INDEX      0
#define POXI_ZYBO_GPIO_REGISTER            ACCESS_REGISTER(IP_POXI_BASEADDR, POXI_ZYBO_GPIO_REGISTER_INDEX)

#define POXI_SPI_CONTROL_REGISTER_INDEX    1
#define POXI_SPI_CONTROL_REGISTER          ACCESS_REGISTER(IP_POXI_BASEADDR, POXI_SPI_CONTROL_REGISTER_INDEX)

#define POXI_SPI_DATA_REGISTER_INDEX       2
#define POXI_SPI_DATA                      ACCESS_REGISTER(IP_POXI_BASEADDR, POXI_SPI_DATA_REGISTER_INDEX)


#define POXI_ZYBO_LEDS_MASK                0x0000000Fu
#define POXI_ZYBO_LEDS_SHIFT               0
#define SET_POXI_ZYBO_LEDS(val)            REGISTER_SET(POXI_ZYBO_GPIO_REGISTER, \
                                                        POXI_ZYBO_LEDS_MASK,     \
                                                        POXI_ZYBO_LEDS_SHIFT,    \
                                                        val)


#define POXI_ZYBO_SWITCHS_MASK             0x0000000Fu
#define POXI_ZYBO_SWITCHS_SHIFT            0
#define GET_POXI_ZYBO_SWITCHS              REGISTER_GET(POXI_ZYBO_GPIO_REGISTER, \
                                                        POXI_ZYBO_SWITCHS_MASK,  \
                                                        POXI_ZYBO_SWITCHS_SHIFT)


#define POXI_ZYBO_PUSHBUTTONS_MASK         0x000000F0u
#define POXI_ZYBO_PUSHBUTTONS_SHIFT        4
#define GET_POXI_ZYBO_PUSHBUTTONS          REGISTER_GET(POXI_ZYBO_GPIO_REGISTER, \
```

Figure 13. POXI IP software driver

The main logic of the custom IP is contained in “POXI_v1_0_S00_AXI.vhd”.

VHDL process for reading registers.

```
process (slv_reg3,
uzPushB, uzSwitch, upsmiso,
poxi_LED_infrared, poxi_LED_red, poxi_spi_slave_select, poxi_spi_transmission_done,
poxi_spi_data_length, poxi_spi_clock_polarity, poxi_spi_clock_phase,
poxi_spi_baud_rate_divider, poxi_spi_data_rx,
axi_araddr, S_AXI_ARESETN, slv_reg_rden)

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"00000" & "000" & upsmiso & uzPushB & uzSwitch;
      when b"01" =>
        reg_data_out <= "000000" & poxi_LED_infrared & poxi_LED_red &
                        "000000" & poxi_spi_slave_select &
                        poxi_spi_transmission_done & "000" & poxi_spi_data_length & poxi_spi_clock_polarity & poxi_spi_clock_phase &
                        poxi_spi_baud_rate_divider;
      when b"10" =>
        reg_data_out <= poxi_spi_data_rx;
      when b"11" =>
        reg_data_out <= slv_reg3;
      when others =>
        reg_data_out  <= (others => '0');
    end case;
end process;
```

VHDL logic for SPI instantiation, CS distribution, red and infrared LED drivers, ZYBO LEDs drivers, ZYBO switches and pushbuttons.

```vhdl
-- Add user logic here
spi_reset_high <= NOT S_AXI_ARESETN;

----------------------- ZYBO LEDs --------------------------------------
uzLED <= slv_reg0(3 downto 0);
-------------------------------------------------------------------------

----------------------- POXI BOARD -------------------------------------
poxi_spi_baud_rate_divider  <= slv_reg1(07 downto 00);
poxi_spi_clock_phase        <= slv_reg1(8);
poxi_spi_clock_polarity     <= slv_reg1(9);
poxi_spi_data_length        <= slv_reg1(11 downto 10);
poxi_spi_slave_select       <= slv_reg1(17 downto 16);
poxi_LED_red                <= slv_reg1(24);
poxi_LED_infrared           <= slv_reg1(25);


udaccs <= NOT poxi_spi_cs WHEN (poxi_spi_slave_select = POXI_SPI_DAC_CS) ELSE '0';
upgacs <= NOT poxi_spi_cs WHEN (poxi_spi_slave_select = POXI_SPI_PGA_CS) ELSE '0';
uadccs <= NOT poxi_spi_cs WHEN (poxi_spi_slave_select = POXI_SPI_ADC_CS) ELSE '0';
uiron  <= poxi_LED_infrared;
urdon  <= poxi_LED_red;

poxi_spi_instance : spi
GENERIC MAP (DATA_LENGTH_BIT_SIZE   => POXI_SPI_DATA_LENGTH_BIT_SIZE,
             DATA_SIZE              => C_S_AXI_DATA_WIDTH,
             BAUD_RATE_DIVIDER_SIZE => POXI_SPI_BAUD_RATE_DIVIDER_SIZE)
PORT MAP ( clk                => S_AXI_ACLK,
           reset              => spi_reset_high ,
           data_length        => poxi_spi_data_length,
           baud_rate_divider  => poxi_spi_baud_rate_divider,
           clock_polarity     => poxi_spi_clock_polarity,
           clock_phase        => poxi_spi_clock_phase,
           start_transmission => poxi_spi_start_transmission,
           transmission_done  => poxi_spi_transmission_done,
           data_tx  => slv_reg2,
           data_rx  => poxi_spi_data_rx,
           spi_clk  => upsclk,
           spi_MOSI => upsmosi,
           spi_MISO => upsmiso,
           spi_cs   => poxi_spi_cs);
-------------------------------------------------------------------------

-- User logic ends
```

2.1.5 TFT LCD 240RGBx320 (ILI9341 driver)

ILI9341 is a 262,144-color single-chip SOC driver for a-TFT liquid crystal display with resolution of 240RGBx320 dots, comprising a 720-channel source driver, a 320-channel gate driver, 172,800 bytes GRAM for graphic display data of 240RGBx320 dots, and power supply circuit. ILI9341 supports parallel 8-/9-/16-/18-bit data bus MCU interface, 6-/16-/18-bit data bus RGB interface and 3-/4-line serial peripheral interface (SPI). The moving picture area can be specified in internal GRAM by window address function. The specified window area can be updated selectively, so that moving picture can be displayed simultaneously independent of still picture area. ILI9341 can operate with 1.65V ~ 3.3V I/O interface voltage and an incorporated voltage follower circuit to generate voltage levels for driving an LCD. ILI9341 supports full colour, 8-color display mode and sleep mode for precise power control by software and these features make the ILI9341 an ideal LCD driver for medium or small size portable products such as digital cellular phones, smart phone, MP3 and PMP where long battery life is a major concern.

For communication between the processing system and the LCD it is implemented 4-line serial interface (reset line is not considered), described in the following section.

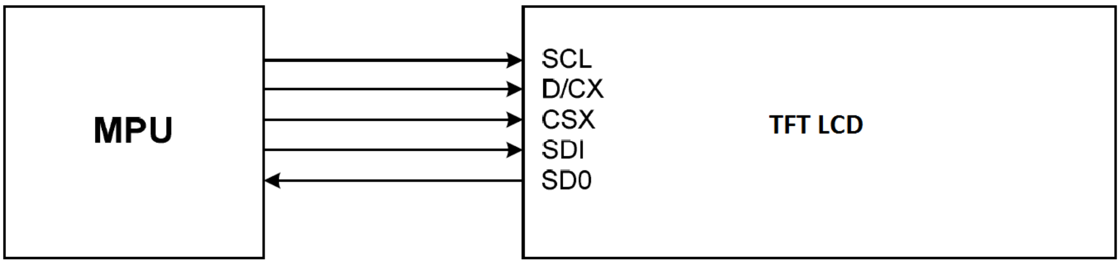


Figure 14. Serial interface TFT LCD

The set-up, configuration, and operation of the TFT LCD is made by the usage of commands that are sent serially through SPI. For more detailed information about the commands and product characteristics it can be reviewed the data sheet of the ILI9341 controller. (ILITEK, 2016)

The following diagram shows the frame sent for setting color in a pixel (this is sent after sending the desired command).

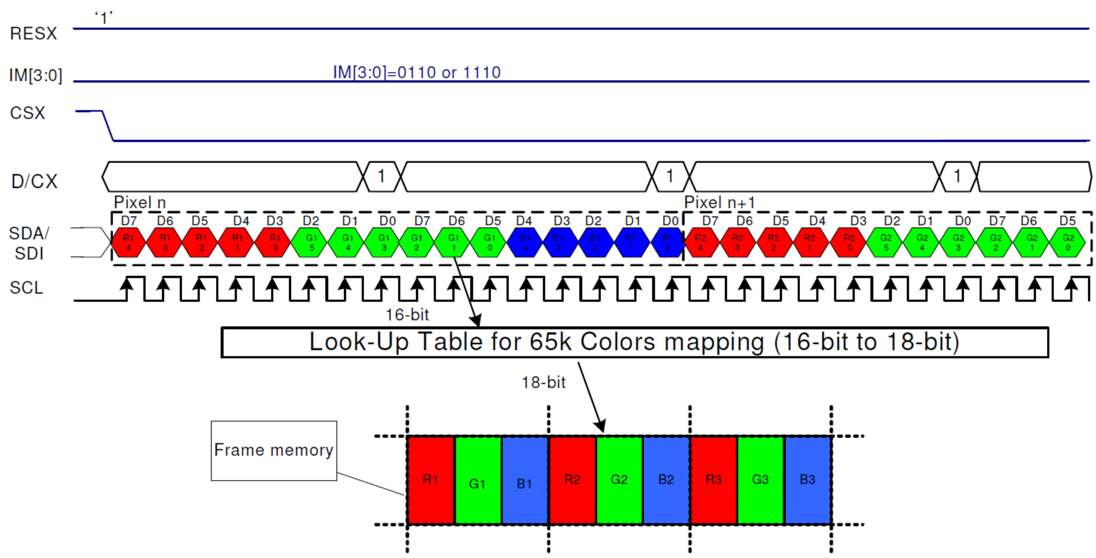


Figure 15. SPI frame, 16 bit pixel color

ILI9341 commands.

```
/* Level 1 Commands ILI9341 Display controller, Yarib */
#define ILI9341_CMD_NOP                             0x00
#define ILI9341_CMD_SOFTWARE_RESET                  0x01
#define ILI9341_CMD_READ_DISP_ID                    0x04
#define ILI9341_CMD_READ_DISP_STATUS                0x09
#define ILI9341_CMD_READ_DISP_MADCTRL               0x0B
#define ILI9341_CMD_READ_DISP_PIXEL_FORMAT          0x0C
#define ILI9341_CMD_READ_DISP_IMAGE_FORMAT          0x0D
#define ILI9341_CMD_READ_DISP_SIGNAL_MODE           0x0E
#define ILI9341_CMD_READ_DISP_SELF_DIAGNOSTIC       0x0F
#define ILI9341_CMD_ENTER_SLEEP_MODE                0x10
#define ILI9341_CMD_SLEEP_OUT                       0x11
#define ILI9341_CMD_PARTIAL_MODE_ON                 0x12
#define ILI9341_CMD_NORMAL_DISP_MODE_ON             0x13
#define ILI9341_CMD_DISP_INVERSION_OFF              0x20
#define ILI9341_CMD_DISP_INVERSION_ON               0x21
#define ILI9341_CMD_GAMMA_SET                       0x26
#define ILI9341_CMD_DISPLAY_OFF                     0x28
#define ILI9341_CMD_DISPLAY_ON                      0x29
#define ILI9341_CMD_COLUMN_ADDRESS_SET              0x2A
#define ILI9341_CMD_PAGE_ADDRESS_SET                0x2B
#define ILI9341_CMD_MEMORY_WRITE                    0x2C
#define ILI9341_CMD_COLOR_SET                       0x2D
#define ILI9341_CMD_MEMORY_READ                     0x2E
#define ILI9341_CMD_PARTIAL_AREA                    0x30
#define ILI9341_CMD_VERT_SCROLL_DEFINITION          0x33
#define ILI9341_CMD_TEARING_EFFECT_LINE_OFF         0x34
#define ILI9341_CMD_TEARING_EFFECT_LINE_ON          0x35
#define ILI9341_CMD_MEMORY_ACCESS_CONTROL           0x36
#define ILI9341_CMD_VERT_SCROLL_START_ADDRESS       0x37
#define ILI9341_CMD_IDLE_MODE_OFF                   0x38
#define ILI9341_CMD_IDLE_MODE_ON                    0x39
#define ILI9341_CMD_COLMOD_PIXEL_FORMAT_SET         0x3A
#define ILI9341_CMD_WRITE_MEMORY_CONTINUE           0x3C
#define ILI9341_CMD_READ_MEMORY_CONTINUE            0x3E
#define ILI9341_CMD_SET_TEAR_SCANLINE               0x44
#define ILI9341_CMD_GET_SCANLINE                    0x45
#define ILI9341_CMD_WRITE_DISPLAY_BRIGHTNESS        0x51
#define ILI9341_CMD_READ_DISPLAY_BRIGHTNESS         0x52
#define ILI9341_CMD_WRITE_CTRL_DISPLAY              0x53
#define ILI9341_CMD_READ_CTRL_DISPLAY               0x54
#define ILI9341_CMD_WRITE_CONTENT_ADAPT_BRIGHTNESS  0x55
#define ILI9341_CMD_READ_CONTENT_ADAPT_BRIGHTNESS   0x56
#define ILI9341_CMD_WRITE_MIN_CAB_LEVEL             0x5E
#define ILI9341_CMD_READ_MIN_CAB_LEVEL              0x5F
#define ILI9341_CMD_READ_ID1                        0xDA
#define ILI9341_CMD_READ_ID2                        0xDB
#define ILI9341_CMD_READ_ID3                        0xDC

// Color definitions
#define TRANSPARENT                                  0
#define BLACK                                        0x0001
#define BLUE                                         0x001F
```

```
#define GREEN                                          0x07E0
#define RED                                            0xF800
#define WHITE                                          0xFFFF
```

2.1.6 IP Component “TFT_SPI_DISPLAY_240x320”

Custom IP, target language VHDL, AXI peripheral 4 registers (32 bits each). The TFT_SPI_DISPLAY_240x320_0 (as shown in figure 3) block is intended to establish communication and control to the TFT SPI display. Inside the custom IP, it is implemented a SPI module.

The figure below shows detailed information of the TFT_SPI_DISPLAY_240x320 ports and interfaces.

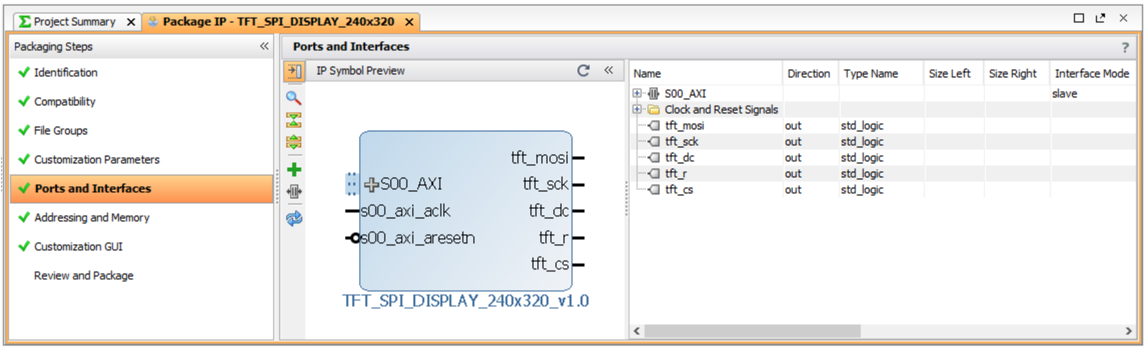


Figure 16. TFT_SPI_DISPLAY_240x320 - IP component interface

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 “spi.vhd” contains the design for the SPI protocol, and “TFT_SPI_DISPLAY_240x320_v1_0_S00_AXI.vhd” implements an instance for SPI communication and the rest of the logic, “TFT_SPI_DISPLAY_240x320_v1_0.vhd” can be considered as wrapper-interface for the AXI peripheral.

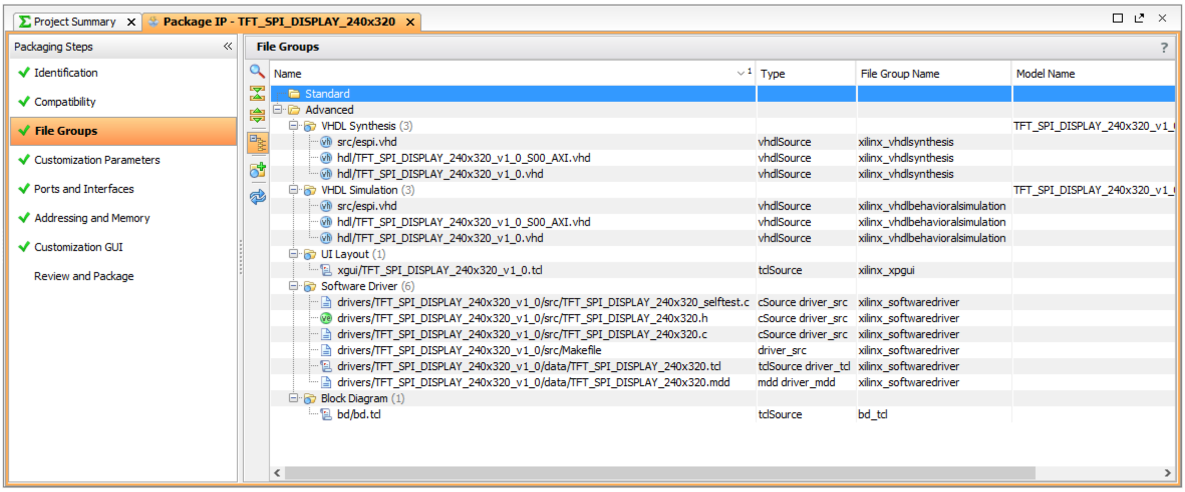
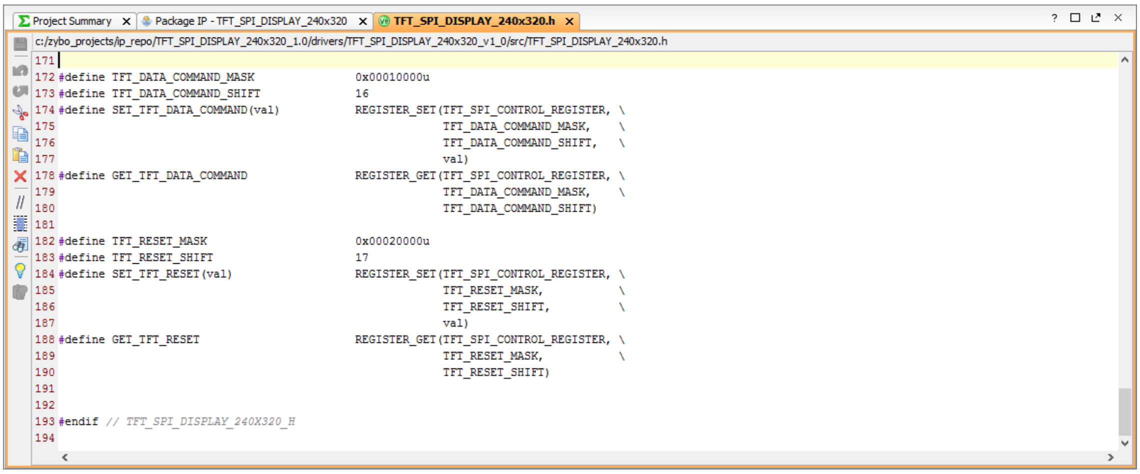


Figure 17. TFT_SPI_DISPLAY_240x320 IP file groups

For software interface it was written “TFT_SPI_DISPLAY_240x320.h” which contains the C/C++ macros for setting and getting data of the custom peripheral.



Project Summary | Package IP - TFT_SPI_DISPLAY_240x320 | TFT_SPI_DISPLAY_240x320.h

c:/zybo_projects/ip_repo/TFT_SPI_DISPLAY_240x320_1.0/drivers/TFT_SPI_DISPLAY_240x320_v1_0/src/TFT_SPI_DISPLAY_240x320.h

```c

#define TFT_DATA_COMMAND_MASK           0x00010000u
#define TFT_DATA_COMMAND_SHIFT          16
#define SET_TFT_DATA_COMMAND(val)       REGISTER_SET(TFT_SPI_CONTROL_REGISTER, \
                                                    TFT_DATA_COMMAND_MASK,    \
                                                    TFT_DATA_COMMAND_SHIFT,   \
                                                    val)
#define GET_TFT_DATA_COMMAND            REGISTER_GET(TFT_SPI_CONTROL_REGISTER, \
                                                    TFT_DATA_COMMAND_MASK,    \
                                                    TFT_DATA_COMMAND_SHIFT)

#define TFT_RESET_MASK                  0x00020000u
#define TFT_RESET_SHIFT                 17
#define SET_TFT_RESET(val)              REGISTER_SET(TFT_SPI_CONTROL_REGISTER, \
                                                    TFT_RESET_MASK,           \
                                                    TFT_RESET_SHIFT,          \
                                                    val)
#define GET_TFT_RESET                   REGISTER_GET(TFT_SPI_CONTROL_REGISTER, \
                                                    TFT_RESET_MASK,           \
                                                    TFT_RESET_SHIFT)


#endif // TFT_SPI_DISPLAY_240X320_H

```

Figure 18. TFT_SPI_DISPLAY_240x320 IP software driver

The main logic of the custom IP is contained in “TFT_SPI_DISPLAY_240x320_v1_0_S00_AXI.vhd”.

VHDL process for reading registers.

```vhdl
process (slv_reg1, slv_reg2, slv_reg3, axi_araddr, S_AXI_ARESETN, slv_reg_rden,
    tft_reset, tft_data_command, tft_spi_transmission_done, tft_spi_settle_time, tft_spi_cs_force,
    tft_spi_data_length, tft_spi_clock_polarity, tft_spi_clock_phase, tft_spi_baud_rate_divider)
    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"00" &
                            "000000" & tft_reset & tft_data_command &
                            tft_spi_transmission_done & tft_spi_settle_time(TFT_SPI_SETTLE_TIME_SIZE-1 DOWNTO TFT_SPI_SETTLE_TIME_SIZE-2) &
                            tft_spi_cs_force & tft_spi_data_length & tft_spi_clock_polarity & tft_spi_clock_phase &
                            tft_spi_baud_rate_divider;
          when b"01" =>
            reg_data_out <= slv_reg1;
          when b"10" =>
            reg_data_out <= slv_reg2;
          when b"11" =>
            reg_data_out <= slv_reg3;
          when others =>
            reg_data_out  <= (others => '0');
        end case;
    end process;
```

VHDL logic for data or command handling (TFT display), and SPI instantiation.

```vhdl
----------------------- TFT DISPLAY -----------------------------------
tft_spi_baud_rate_divider <= slv_reg0(07 DOWNTO 0);
tft_spi_clock_phase       <= slv_reg0(8);
tft_spi_clock_polarity    <= slv_reg0(9);
tft_spi_data_length       <= slv_reg0(11 DOWNTO 10);
tft_spi_cs_force          <= slv_reg0(12);
tft_spi_settle_time       <= slv_reg0(14 DOWNTO 13) & "00";
tft_data_command          <= slv_reg0(16);
tft_reset                 <= slv_reg0(17);

utft_r  <= tft_reset;

utft_cs <= spi_cs_internal AND (NOT tft_spi_cs_force);
```

The following process gives proper control for the command line in the TFT display.

```
command_data_bit_block: BLOCK
    SIGNAL tft_dc_current_state, tft_dc_next_state: TFT_DC_STATE_TYPE := DC_HIGH;
BEGIN
    command_data_bit_process : process (S_AXI_ACLK, spi_cs_internal, tft_data_command, tft_dc_current_state, tft_dc_next_state)
    BEGIN
        IF RISING_EDGE(S_AXI_ACLK) THEN
            IF S_AXI_ARESETN = '0' THEN
                tft_dc_next_state <= DC_HIGH;
            ELSE
                tft_dc_current_state <= tft_dc_next_state;
            END IF;

            tft_dc_next_state <= tft_dc_current_state;
            CASE tft_dc_current_state IS
                WHEN DC_LOW  =>
                    utft_dc <= '0';
                    IF tft_data_command  = '1' AND tft_spi_transmission_done  = '1' THEN
                        tft_dc_next_state <= DC_HIGH;
                    END IF;
                WHEN DC_HIGH =>
                    utft_dc <= '1';
                    IF tft_data_command  = '0' AND tft_spi_transmission_done  = '1' THEN
                        tft_dc_next_state <= DC_LOW;
                    END IF;
                END CASE;

        END IF;
    END PROCESS;
END BLOCK command_data_bit_block;

tft_espi_instance : espi
GENERIC MAP (DATA_LENGTH_BIT_SIZE   => TFT_SPI_DATA_LENGTH_BIT_SIZE,
             SETTLE_TIME_SIZE       => TFT_SPI_SETTLE_TIME_SIZE,
             DATA_SIZE              => C_S_AXI_DATA_WIDTH,
             BAUD_RATE_DIVIDER_SIZE => TFT_SPI_BAUD_RATE_DIVIDER_SIZE)
PORT MAP ( clk                => S_AXI_ACLK,
           reset              => spi_reset_high,
           data_length        => tft_spi_data_length,
           baud_rate_divider  => tft_spi_baud_rate_divider,
           settle_time        => tft_spi_settle_time,
           clock_polarity     => tft_spi_clock_polarity,
           clock_phase        => tft_spi_clock_phase,
           start_transmission => tft_spi_start_transmission,
           transmission_done  => tft_spi_transmission_done,
           data_tx  => slv_reg1,
           data_rx  => OPEN,
           spi_clk  => utft_sck,
           spi_MOSI => utft_mosi,
           spi_MISO => '-',
           spi_cs   => spi_cs_internal);
------------------------------------------------------------------------
-- User logic ends
```

2.1.7 Enhanced SPI

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

- Baud rate divider for output clock signal (SCLK)
- Configurable data length (8, 16, 24 and 32 bits)
- Flexible Settle-time for specific devices
- Configurable clock polarity (CPOL) and clock phase (CPHA).
- Full duplex data transmission.

The implemented VHDL code for the Enhanced SPI is listed below.

```
ENTITY spi IS
    GENERIC (DATA_LENGTH_BIT_SIZE   : INTEGER := 2;
             SETTLE_TIME_SIZE       : INTEGER := 2;
             DATA_SIZE              : INTEGER := 32;
             BAUD_RATE_DIVIDER_SIZE : INTEGER := 8);
    PORT ( clk                : 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);
           settle_time        : IN STD_LOGIC_VECTOR (SETTLE_TIME_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 spi;
```

The baud rate divider is implemented in the following process.

```
baud_rate_division_process: PROCESS (clk, reset, baud_rate_divider, settle_time, clk_pulse)
VARIABLE baud_rate_counter   : UNSIGNED (BAUD_RATE_DIVIDER_SIZE-1 DOWNTO 0)  := (others => '0');
VARIABLE settle_time_counter : UNSIGNED (SETTLE_TIME_SIZE-1 DOWNTO 0)        := (others => '0');
BEGIN
    IF falling_edge(clk) THEN
        clk_pulse <= '0';
        IF reset = '1' OR current_state = SPI_IDLE THEN
            baud_rate_counter := (others => '0');
            settle_time_counter := (others => '0');
        ELSIF baud_rate_divider = CONV_STD_LOGIC_VECTOR(baud_rate_counter, BAUD_RATE_DIVIDER_SIZE) THEN
            baud_rate_counter := (others => '0');
            ----------------------------------------------------------------
            IF current_state = SPI_REDY OR current_state = SPI_STOP THEN
                IF settle_time = CONV_STD_LOGIC_VECTOR(settle_time_counter, SETTLE_TIME_SIZE) THEN
                    clk_pulse <= '1';
                    settle_time_counter := (others => '0');
                ELSE
                    settle_time_counter := settle_time_counter + 1;
                END IF;
            ELSE
                clk_pulse <= '1';
            END IF;
            ----------------------------------------------------------------
        ELSE
            baud_rate_counter := baud_rate_counter + 1;
        END IF;
    END IF;
END PROCESS;
```

The mechanism for switching from states (SPI as state machine) is implemented in the following process. This process determine the number of bits that should be transmitted-received based on the selected data length, the shifting data in the internal buffers, and the setup for the next state.

```
spi_switch_state : PROCESS (clk, current_state, next_state)
VARIABLE data_length_internal : UNSIGNED (1 DOWNTO 0);
BEGIN
    IF rising_edge(clk) 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_REDY THEN -- Get redy immediately
            current_state <= SPI_REDY;
            data_rx <= (others => '0');
            i_rx_buffer <= (others => '0');

            CASE data_length IS
                WHEN "00" =>
                    i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_0) <= data_tx(DATA_LENGTH_0-1 downto 0);
                    counter <= DATA_LENGTH_0-1;
                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 "11" =>
                    i_tx_buffer(DATA_SIZE-1 downto DATA_SIZE-DATA_LENGTH_3) <= data_tx(DATA_LENGTH_3-1 downto 0);
                    counter <= DATA_LENGTH_3-1;
                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_ACTIVE THEN
                    i_rx_buffer <= i_rx_buffer(DATA_SIZE-2 DOWNTO 0) & spi_MISO;
                END IF;
                -- POP OUTPUT
                IF next_state = SPI_CLK_IDLE 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_ACTIVE THEN
                    i_rx_buffer <= i_rx_buffer(DATA_SIZE-2 DOWNTO 0) & spi_MISO;
                END IF;
                -- POP OUTPUT
                IF current_state = SPI_CLK_IDLE AND next_state = SPI_CLK_ACTIVE THEN
                    i_tx_buffer <= i_tx_buffer(DATA_SIZE-2 downto 0) & '-';
                    counter <= counter - 1;
                END IF;
            END IF;

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

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

The SPI state-machine is implemented in the following process.

```vhdl
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_REDY;
            END IF;
        WHEN SPI_REDY =>
            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_ACTIVE;

        WHEN SPI_CLK_ACTIVE =>
            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_STOP;
            ELSE
                next_state <= SPI_CLK_IDLE;
            END IF;
        WHEN SPI_CLK_IDLE =>
            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_STOP;
            ELSE
                next_state <= SPI_CLK_ACTIVE;
            END IF;
        WHEN SPI_STOP =>
            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;
```

The internal mechanism is summarized in the following state diagram.

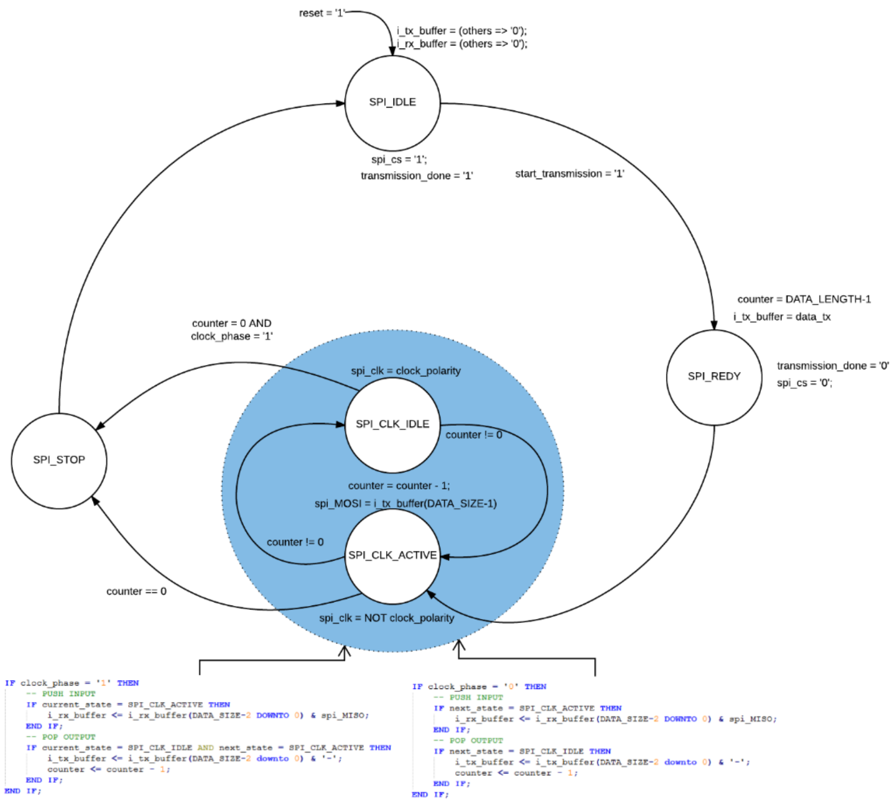


Figure 19. Enhanced SPI – State machine diagram

For the simulation it was instantiated an Enhanced SPI, it was connected `MOSI` and `MISO` lines so the given value in the `data_tx` register appears in `data_rx` register after the transmission (full duplex).

- `data_length` set to two so 24 bits will be transmitted
- `settle_time` set to three to start the transmission after four clock cycles (40ns)
- `baud_rate_divider` set to zero for no clock division (full speed)
- `clock_polarity` set to one making high the idle state of the output clock signal
- `clock_phase` set to zero in order to start the transmission with no phase delay

The data to be transmitted is set to the `data_tx` register (32 bits), in this simulation it set to `a51188a5`, since the data length was setup for having a transmission of 24 bits (8 bytes) and `MOSI` and `MISO` lines are connected, the received data in `data_rx` is `001188a5` (24 bits) after 24 clock cycles.

For more detailed information regarding the SPI protocol it can be referred to the documentation of standardized SPI protocol.

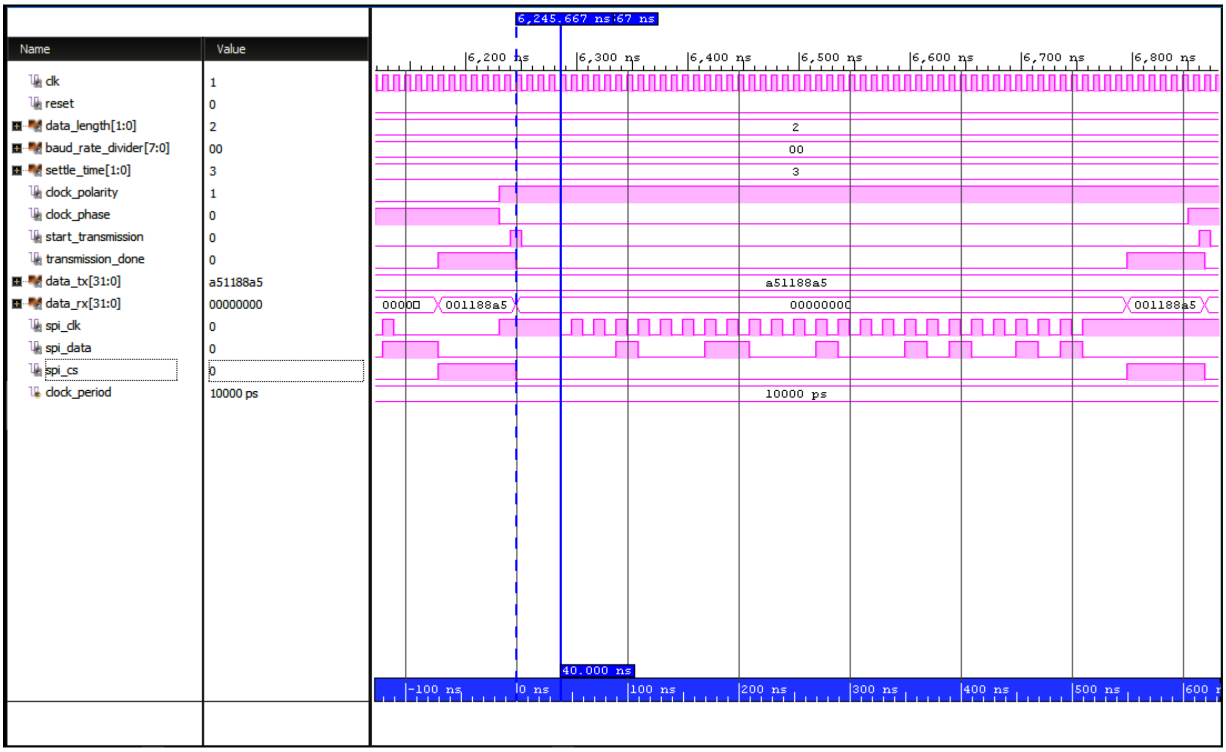


Figure 20. Enhanced SPI - Simulation

2.2 SOFTWARE.

The implemented software for the pulse oximeter project consist on two software sections: the firmware application, and the desktop application. These applications run on different systems, these are communicated through serial UART protocol communication.

The Firmware application (embedded software) runs in the ZYBO board on one ARM Core of the Zynq device. This software is implemented in C language, for this project the code is written following C++ style. This software implementation is robust, it incorporates device drivers for POXI board and TFT color display, middle ware, protocol communicator, signal processing, graphics library and UI components. The firmware is an standalone application (no operating system).

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. This application is not robust, its function is to show information from POXI board.

The following table shows the software architecture diagram.

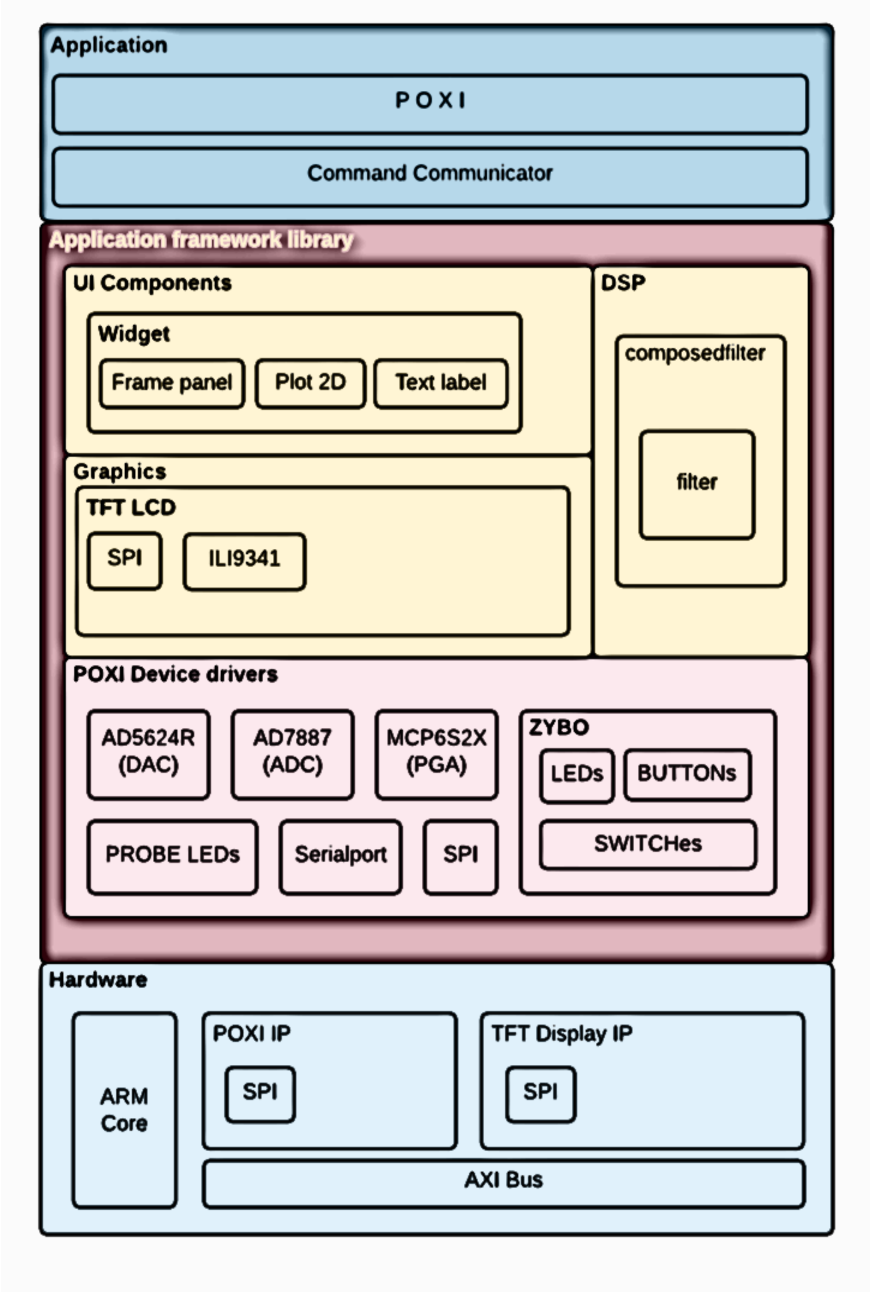


Figure 21. Firmware architecture diagram

The communication between the POXI device and the desktop system is made by UART serial communication, as shown in the following figure.

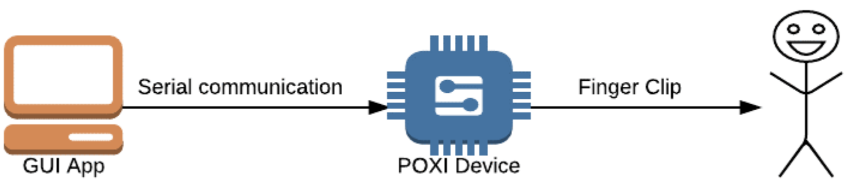


Figure 22. Communication diagram

2.2.1 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 it 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. It was implemented the device driver software for the ZYBO LEDs, push buttons, and switches in the same manner.

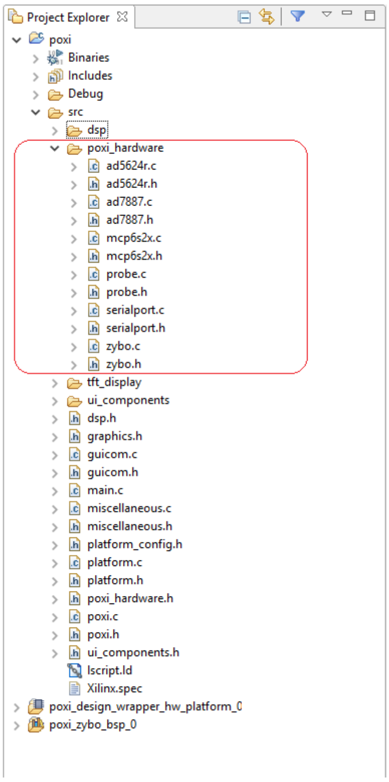


Figure 23. Device drivers - File tree structure

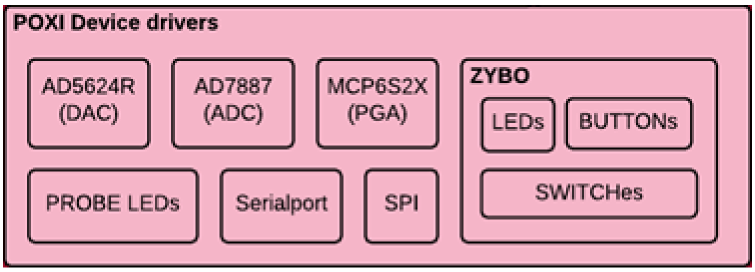


Figure 24. Device drivers – Software architecture diagram

2.2.1.1 ZYBO IO

The ZYBO drivers unit consists on a singleton design pattern encapsulating internal mechanisms and interfacing the significant functions.

Interface definition "zybo.h":

```
typedef struct
{
    void     (*leds)     (uint8_t val);
    void     (*led)      (uint8_t ld, uint8_t val);
    uint8_t (*switches) (void);
    uint8_t (*switch_)  (uint8_t sw);
    uint8_t (*buttons)  (void);
    uint8_t (*button)   (uint8_t btn);
```

```
} ZYBO;

ZYBO * ZYBO_instance();
```

Implementation “zybo.c":

(NOTE: This is only a piece of code to illustrate the singleton implementation)

```
static ZYBO ZYBO_obj = { ZYBO_leds,
                         ZYBO_led,
                         ZYBO_switches,
                         ZYBO_switch_,
                         ZYBO_buttons,
                         ZYBO_button };

ZYBO * ZYBO_instance()
{
    return & ZYBO_obj;
}
…

static uint8_t ZYBO_switches (void)
{
    return GET_POXI_ZYBO_SWITCHS;
}
```

In the function `ZYBO_switches`, the switches state is simply obtained by reading the AXI address of the custom IP (POXI).

The definition of “`GET_POXI_ZYBO_SWITCHS`” is located in “POXI.h” which is part of the POXI IP (part of the custom IP).

```
#define POXI_ZYBO_SWITCHS_MASK        0x0000000Fu
#define POXI_ZYBO_SWITCHS_SHIFT        0
#define GET_POXI_ZYBO_SWITCHS         REGISTER_GET(POXI_ZYBO_GPIO_REGISTER, \
                               POXI_ZYBO_SWITCHS_MASK,   \
        POXI_ZYBO_SWITCHS_SHIFT)

…
#define IP_POXI_BASEADDR   XPAR_POXI_0_S00_AXI_BASEADDR
```

Where the “`XPAR_POXI_0_S00_AXI_BASEADDR`” is taken from “xparameters.h”:

```
#define XPAR_POXI_0_S00_AXI_BASEADDR 0x43C00000
```

The following UML figure exhibits the class interface.

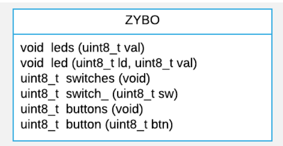


Figure 25. ZYBO - UML class diagram

Here is presented the usage in a fragment of code (self-explanatory). It is not needed for initialization routine.

```
// Obtains driver singleton instance
static ZYBO * Poxi_ZYBO = ZYBO_instance();

// Checks the last switsh
if (Poxi_ZYBO->button(3))
{   // Clears statistics
    Poxi_DSP.ired->resetStatistics(Poxi_DSP.ired);
}

// Turns ON all LEDs
Poxi_ZYBO->leds(0xF);

// Turns OFF all LEDs
Poxi_ZYBO->leds(0x0);
```

2.2.1.2 Digital to analogue converter (AD5624R)

This class performs as a software interface of the digital to analogue converter (the class name is the same as the hardware device AD5624R). This driver implements the SPI communication with the hardware. The class is defined in AD5624R.h.

The SPI initialization is done by the following function.

```
inline static void AD5624R_init_spi(void)
{
    while (!GET_POXI_SPI_TRANSMISSION_DONE);
    SET_POXI_SPI_SLAVE_SELECT(POXI_SPI_DAC_CS);
    SET_POXI_SPI_DATA_LENGTH(POXI_SPI_DATA_LENGTH_24_BITS);
    SET_POXI_SPI_BAUD_RATE_DIVIDER(AD5624R_SPI_BAUD_RATE);
    SET_POXI_SPI_CLOCK_POLARITY(1);
    SET_POXI_SPI_CLOCK_PHASE(0);
}
```

It can be seen in the function that the SPI protocol setup, CPOL = 1, CPHA = 0, 24 bits. For more details regarding this implementation, it can be inferred from the code.

The following diagram displays the AD5624R class in UML manner.

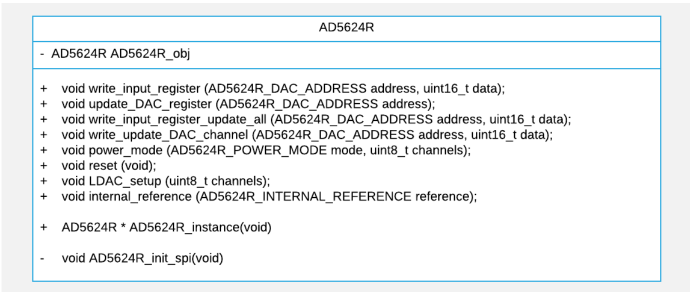


Figure 26. AD5624R - UML class diagram

The following code initialize the DAC device.

```
static AD5624R * Poxi_DAC = AD5624R_instance();

Poxi_DAC->reset();
Poxi_DAC->LDAC_setup(0);
Poxi_DAC->power_mode(NORMAL_OPERATION, 0xF);
Poxi_DAC->internal_reference(REFERENCE_ON);
```

The following type enum indicates the power modes of the DAC, these are used as the first parameter of the `power_mode` function.

```
typedef enum
{
    NORMAL_OPERATION        = 0,
    POWER_DOWN_1K_GND       = 1,
    POWER_DOWN_100K_GND     = 2,
    POWER_DOWN_THREE_STATE = 3,
} AD5624R_POWER_MODE;
```

The following code writes a value in channel *A* (it can be seen the options in the header file to write a value to another channel).

```
Poxi_DAC->write_input_register(DAC_A, value);
Poxi_DAC->update_DAC_register(DAC_A);
```

The following code turns off the DAC device. The second parameter corresponds bitwise to the four DAC channels.

```
Poxi_DAC->power_mode(POWER_DOWN_THREE_STATE, 0xF);
```

For detailed information it can be referred the actual code.

2.2.1.3 Analogue to digital converter (AD7887)

This class is the device driver for the ADC. As well as the AD5624R this class accomplish the SPI communication to the ADC device (AD7887). The class is defined in AD7887.h.

The SPI initialization is performed as exhibits the following function.

```
inline static void AD7887_init_spi(void)
{
    while (!GET_POXI_SPI_TRANSMISSION_DONE);
    SET_POXI_SPI_SLAVE_SELECT(POXI_SPI_ADC_CS);
    SET_POXI_SPI_DATA_LENGTH(POXI_SPI_DATA_LENGTH_16_BITS);
    SET_POXI_SPI_BAUD_RATE_DIVIDER(AD7887_SPI_BAUD_RATE);
    SET_POXI_SPI_CLOCK_POLARITY(1);
    SET_POXI_SPI_CLOCK_PHASE(0);
}
```

From this code is clearly seen the SPI configuration for the device: CPOL = 1, CPHA = 0, 16 bits.

The following diagram reveals the class in UML style.

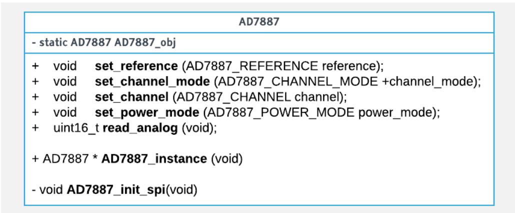


Figure 27. AD7887 - UML class diagram

Initialization code.

```
static AD7887 * Poxi_ADC = AD7887_instance();
Poxi_ADC->set_reference(REF_ENABLED);
Poxi_ADC->set_channel_mode(SINGLE_CHANNEL);
Poxi_ADC->set_power_mode(MODE2);
```

The following table present the power mode options.

Table 2. AD7887 Power modes.

Mode	Description
MODE1	The AD7887 enters shutdown if the CS input is 1 and is in full power mode when CS is 0.
MODE2	The AD7887 is always fully powered up, regardless of the status of any of the logic inputs.
MODE3	The AD7887 automatically enters shutdown mode at the end of each conversion, regardless of the state of CS.
MODE4	In this standby mode, portions of the AD7887 are powered down but the on-chip reference voltage remains powered up.

The incoming code illustrates how to get the analogue voltage conversion.

```
primitiveSignal = Poxi_ADC->read_analog();
```

The following code maintains in shutdowns condition in the ADC while its CS is in low state.

```
Poxi_ADC->set_power_mode(MODE1);
```

For detailed information it can be referred the actual code.

2.2.1.4 Programmable gain amplifier (MCP6S2X)

In the same way as the previous device drivers, this class implements the interfaces the functions for the programmable gain amplifier and executes the SPI configuration and communication. The class is defined in MCP6S2X.h.

The following code exhibits the SPI setup to establish SPI communication with MCP6S2X device.

```
inline static void MCP6S2X_init_spi(void)
{
        while (!GET_POXI_SPI_TRANSMISSION_DONE);
        SET_POXI_SPI_SLAVE_SELECT(POXI_SPI_PGA_CS);
        SET_POXI_SPI_DATA_LENGTH(POXI_SPI_DATA_LENGTH_16_BITS);
        SET_POXI_SPI_BAUD_RATE_DIVIDER(MCP6S2X_SPI_BAUD_RATE);
        SET_POXI_SPI_CLOCK_POLARITY(1);
        SET_POXI_SPI_CLOCK_PHASE(1);
}
```

In the previous code it can be seen the SPI configuration, CPOL = 1, CPHA = 1, 16 bits.

The incoming figure displays the MCP6S2X class in UML manner.

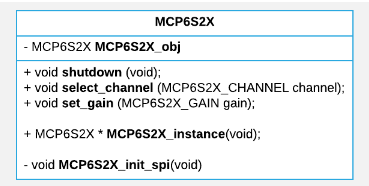


Figure 28. MCP6S2X - UML class diagram

The instance of this class does not need to be initialized.

In the next line is shown the code to set the gain in the PGA.

```
Poxi_PGA->set_gain(value);
```

The next table enlists the defined values accessible for the device (the only parameter of the `set_gain` function).

```
typedef enum
{
        GAIN_1 = 0,
        GAIN_2 = 1,
        GAIN_4 = 2,
        GAIN_5 = 3,
        GAIN_8 = 4,
        GAIN_10 = 5,
        GAIN_16 = 6,
        GAIN_32 = 7
} MCP6S2X_GAIN;
```

The next function shuts down the PGA device.

```
Poxi_PGA->shutdown();
```

For detailed information it can be referred the actual code.

2.2.1.5 Light probe

The Light probe class is intended to provide a proper device driver layer to control the finguer clip (light probe). This driver turns on-off both red and infrared LEDs. The class interface is defined in probe.h.

The communication to the AXI peripheral (POXI) is made by just performing a writing to its register, no SPI communication and no initialization are needed.

The figure below presents the LightProbe class defined in UML style.

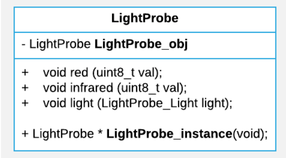


Figure 29. LightProbe - UML class diagram

The code below illustrates a common usage of the LightProbe class instance.

```
static LightProbe * Poxi_clip = LightProbe_instance();
switch (Poxi_interruptState)
{
case REDLIGHT_ON:
    Poxi_clip->light(LIGHTPROBE_RED);
    break;

case SAMPLERED_OFF:
    Poxi_clip->light(LIGHTPROBE_OFF);
    break;

case IREDLIGHT_ON:
    Poxi_clip->light(LIGHTPROBE_INFRARED);
    break;

case SAMPLEIRED_OFF:
    Poxi_clip->light(LIGHTPROBE_OFF);
    break;
default:;
}
```

For detailed information it can be referred the real code.

2.2.1.6 Serial communicator

This 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. This class interface is defined in serialport.h.

For a suitable and reliable communication between desktop system and POXI device it was developed a based command protocol communication. This protocol communication is exhibit in the following figures.

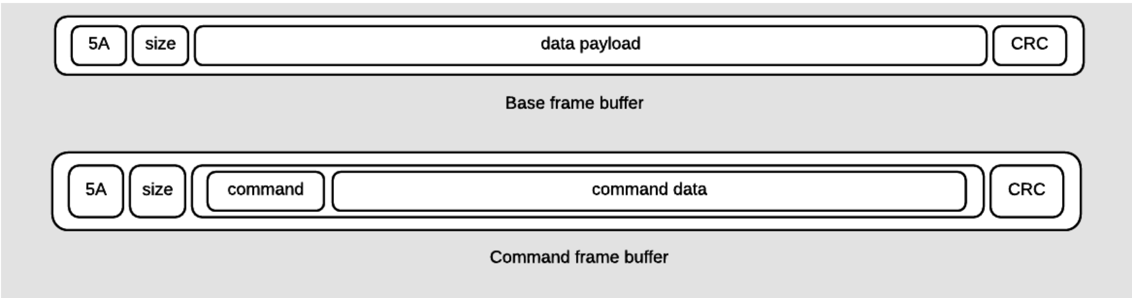


Figure 30. Frame buffer

The next tables describe the items contained in a base frame and command frame buffers.

Table 3. Frame buffer, item description

Item	Description
0x5A	Byte signature. Indicates the beginning of a transmission for both sender and receiver.
Size	Indicates the size of the data payload. In the case of a command frame it indicates the added size of both the command and command data.
Data payload	Contains the core data or actual information that is intended to be transmitted. It may contain a command to give a more specific meaning or purpose.
CRC	Cyclic Redundancy Check. A cyclic redundancy check (CRC) is an error detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data.

This class is intended to handle the serial UART communication, it uses the serial communication drivers provided by the board support package, it gives a reliable communication interface which solves problems of synchrony and data integrity.

The following code is a fragment of the class implementation.

```
static uint32_t SerialPort_stdinAddress    = STDIN_BASEADDRESS;
static uint32_t SerialPort_stdoutAddress   = STDOUT_BASEADDRESS;
static uint32_t SerialPort_timeOut         = 10000000;
static uint8_t  SerialPort_maximumRetries = 3;

static const uint8_t SerialPort_frameSignature = 0x5A;

static uint8_t SerialPort_CRC(uint8_t * buffer, uint8_t size)
{
    uint8_t crc = 0;
    if (buffer != NULL)
    {
        uint8_t i;
        for (i = 0; i < size; i ++)
        {
            crc += buffer[i];
        }
    }

    return crc;
}

static void SerialPort_sendByte(uint8_t byte)
{
    XUartPs_SendByte(SerialPort_stdoutAddress, byte);
}

static void SerialPort_sendBuffer(uint8_t * buffer, uint8_t size)
{
    if (buffer != NULL)
    {
        uint8_t i;
        for (i = 0; i < size; i ++)
        {
            SerialPort_sendByte(buffer[i]);
        }
    }
}

static uint8_t SerialPort_sendFrameBuffer(uint8_t * buffer, uint8_t size)
{
    uint8_t rc = 0;

    if (buffer != NULL)
    {
        uint8_t crc_r = 0;
        uint8_t attempts = 0;
        uint8_t crc = SerialPort_CRC(buffer, size);
        do
        {
            SerialPort_purge();
            SerialPort_sendByte(SerialPort_frameSignature);
            SerialPort_sendByte(size);
            SerialPort_sendBuffer(buffer, size);
            SerialPort_sendByte(crc);

            rc = SerialPort_receiveByte(&crc_r) && (crc_r == (crc + size));
        } while(!rc && (attempts++) < SerialPort_maximumRetries);
    }

    return rc;
}
```

In the fragment of code it can be seen that the class is making usage of the `XUartPs_SendByte` function from the board support package, this function writes a byte in the corresponding UART transmission register, the UART module is part of the SoC device, it is already available in the chip.

The next figure displays the class in UML style.

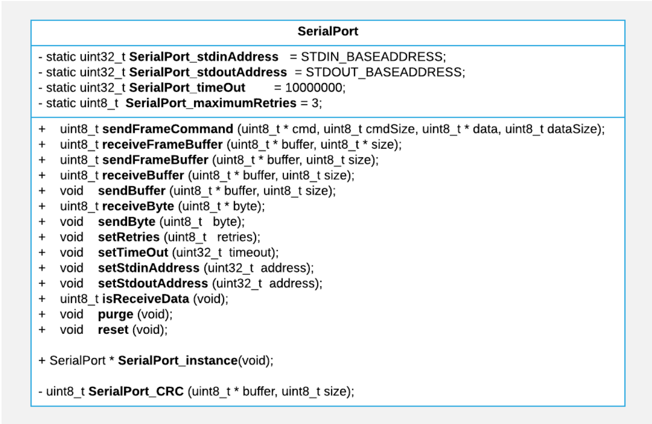


Figure 31. SerialPort - UML class diagram

The incoming lines of code shows a function implemented to use the SerialPort instance and send a command to display text in the Desktop Java GUI.

```
#define CMD_TEXT_MSG        0x05
static void     GUICom_textMsg     (uint8_t id, char * msg)
{
    uint8_t cmd[] = {CMD_TEXT_MSG, 0, 0};
    cmd[1] = id;
    cmd[2] = strlen(msg);
    SerialPort_instance()->sendFrameCommand(cmd,
                                  sizeof(cmd),
                                  (uint8_t *)msg,
                                  sizeof(char) * strlen(msg));
}
```

For more details regarding this class, it can be reviewed its source code.

2.2.2 Signal processing

The signal processing unit consist of a generic digital filter structure, it is implemented in the class filter and extended in the class composed filter. The filter and composed filter classes mainly employ polymorphism, encapsulation OOP characteristics, as well as composite and factory design patterns. It can be instantiated as many filters and composes filters as needed. The composed filter is capable to drive as many filters as desired.

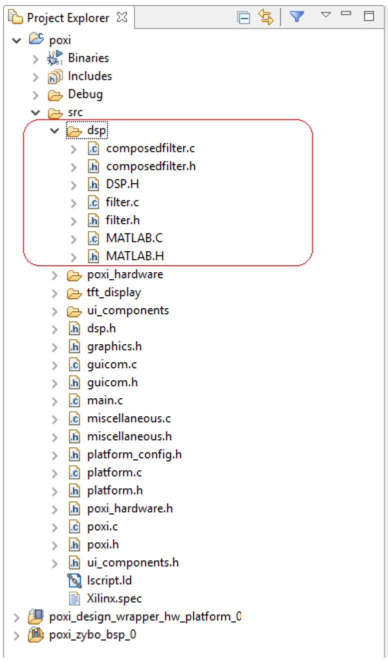


Figure 32. DSP - File tree structure

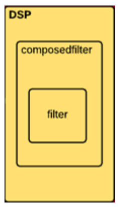


Figure 33. DSP – Software architecture diagram

2.2.2.1 Discrete filter (DSP)

This class implements a generic discrete filter. This implementation executes a linear system mechanism with feedback and feedforward coefficients in double precision. The mechanism implemented follows the same structure as any straightforward filter code. The filter class is defined in filter.h.

In the next code is exhibited the filter class definitions and interfaces.

```
typedef struct

{
    const unsigned char order;  // Filter order
    const double *      a_coef; // Feedback filter coefficients
    const double *      b_coef; // Feedforward filter coefficients
} FilterParameters;

typedef struct Filter_public Filter;

struct Filter_public
{
    void   (* reset)   (Filter * obj);
    double (* process) (Filter * obj, double u);
    void   (* delete)  (Filter ** obj);
    double (* output)  (Filter * obj);
    const FilterParameters * (* access_parameters) (Filter * obj);
};

Filter * Filter_new(const FilterParameters * parameters);
```

The encapsulation of data and function members is made by type casting. First, it is allocated memory needed for the bases private class, then it is fulfilled with proper data types (data and function pointers as virtual table); secondly, it is out-casted to a base type class reducing its data member accessibility.

The following code shows the private implementation.

```
typedef struct // class
{       // public:
        void   (* reset)   (Filter * obj);
        double (* process) (Filter * obj, double u);
        void   (* delete)  (Filter ** obj);
        double (* output)  (Filter * obj);
        const FilterParameters * (* access_parameters) (Filter * obj);
        // private:
        const FilterParameters * parameters;
        double   * states;
        double     y;
} Filter_private;


static void Filter_init_virtual_table(Filter_private * filter)
{
    if (filter != NULL)
    {
        filter->reset = Filter_reset;
        filter->process = Filter_process;
        filter->output = Filter_output;
        filter->access_parameters = Filter_access_parameters;
        filter->delete = Filter_delete;
    }
}

Filter * Filter_new(const FilterParameters * parameters)
{
    Filter_private * filter = NULL;

    if (   parameters != NULL
        && parameters->a_coef != NULL
        && parameters->b_coef != NULL
        && parameters->order > 0)
    {
        filter = (Filter_private *) malloc(sizeof(Filter_private));

        if (filter != NULL)
        {
            memset(filter, 0x00, sizeof(Filter_private));
```

```
            Filter_init_virtual_table(filter);

            filter->parameters = parameters;

            Filter_reset((Filter *) filter);
        }
    }

    return (Filter *) filter;
}

static void   Filter_delete (Filter ** obj)
{
    if (obj != NULL && *obj != NULL)
    {
        Filter_private * filter = (Filter_private *) *obj;

        if (filter->states != NULL)
        {
            free(filter->states);
        }

        free(filter);

        *obj = NULL;
    }
}
```

The filter process is implemented in the following function.

```
static double Filter_process(Filter * obj, double u)
{
    double y = (double)0.0;
    if (obj != NULL)
    {
        Filter_private * filter = (Filter_private *) obj;

        if (filter->parameters != NULL)
        {
            const double * A = filter->parameters->a_coef;
            const double * B = filter->parameters->b_coef;
            const unsigned char n = filter->parameters->order;

            double * x = filter->states;

            if ((n > 0) && (A != NULL) && (B != NULL) && (x != NULL))
            {
                unsigned char i;

                y = B[0] * u + x[0];

                for (i = 1; i < n; i ++) { x[i-1] = B[i] * u - A[i] * y + x[i]; }

                x[n-1] = B[n] * u - A[n] * y;

                filter->y = y;
            }
        }
    }
    return y;
}
```

The next code present a factory creation of a filter instance, it is created by using its constructor and destroyed by using its destructor. The filter coefficients have been taken from MATLAB.

```
#define HIGH_PASS_CUTOFF_FREC_HZ  0.83
#define HIGH_PASS_FILTER_ORDER    4

#define F0_A0   ((const double)0000000000000001)
#define F0_A1   ((const double)-3.9863177122115889e+00)
#define F0_A2   ((const double)5.9590466614474735e+00)
#define F0_A3   ((const double)-3.9591398122141528e+00)
#define F0_A4   ((const double)9.8641086372476394e-01)

#define F0_B0   ((const double)9.9318219059987367e-01)
#define F0_B1   ((const double)-3.9727287623994947e+00)
#define F0_B2   ((const double)5.9590931435992420e+00)
#define F0_B3   ((const double)-3.9727287623994947e+00)
#define F0_B4   ((const double)9.9318219059987367e-01)
```

```
const double A_high_pass [] = {F0_A0, F0_A1, F0_A2, F0_A3, F0_A4};
const double B_high_pass [] = {F0_B0, F0_B1, F0_B2, F0_B3, F0_B4};

const FilterParameters high_pass_filter_parameters = {HIGH_PASS_FILTER_ORDER, A_high_pass,
B_high_pass};

Filter * filter = Filter_new(&high_pass_filter_parameters);
```

The next line of code shows the usage of the filter to process a signal.

```
void AnyClass::timerInterruptHandler(void)
{
primitiveSignal = Poxi_ADC->read_analog()

filteredSignal = filter->process(filter, primitiveSignal);

stdIO::fprintf(file, "%.4f  %.4f\n", primitiveSignal, filteredSignal);
}
```

This line of code is used to delete a filter instance.

```
filter->delete(&filter);
```

For more details about the code implementation it can be referred the actual source code.

2.2.3 Composed discrete filter

The composed filter is intended to be an extension of the base filter class and to provide more features. The composed filter basically connects and wraps any number of filter instances, and takes statistics when processing signals. The filter statistics, after the signal processing is one of the most useful feature of this class, and this can be reconfigurable on flight. The composed filter class is defined in composedfilter.h.

Class definition.

```
#include "filter.h"

typedef struct
{
    double magnitude;
    double time;
} DiscretePoint;

typedef struct
{
    DiscretePoint maxOutput;
    DiscretePoint minOutput;
    DiscretePoint maxInput;
    DiscretePoint minInput;
    DiscretePoint currentOutput;
    DiscretePoint lastInput;
    DiscretePoint lastZeroCross;
    DiscretePoint secLastZeroCross;
    double        fundamentalFrec;
} FilterStatistics;

typedef enum
{
    STATISTICS_OFF   = 0x00,
    MAX_OUTPUT       = 0x01,
    MIN_OUTPUT       = 0x02,
    MAX_INPUT        = 0x04,
    MIN_INPUT        = 0x08,
    FUNDAMENTAL_FREC = 0x10
} FilterStatisticsFlags;

typedef struct ComposedFilter_public ComposedFilter;

struct ComposedFilter_public // class
{
    void   (* reset)   (ComposedFilter * obj);
    double (* process)(ComposedFilter * obj, double u);
    void   (* delete)          (ComposedFilter ** obj);
    double (* processSignal)   (ComposedFilter * obj,
                                double * input, double * output, unsigned int length);
    void   (* setSampleTime)   (ComposedFilter * obj, double time);
```

```
    void   (* setupStatistics)(ComposedFilter * obj, int flags);
    void   (* resetStatistics)(ComposedFilter * obj);
    FilterStatistics (* getStatistics)(ComposedFilter * obj);
};

ComposedFilter * ComposedFilter_new (int number_of_filters, ...);
```

The constructor of the class is a variable parameter function, it receives a variable number of filter instances, and the first parameter is the number of filters that will build up the composed filter.

Here is an example of two filters given to a composed filter, low and high pass filter, giving as a result a band pass filter.

```
ComposedFilter * bandpassFilter;
Filter *         highpassfilter;
Filter *         lowpassfilter;

highpassfilter = Filter_new(&high_pass_filter_parameters);
lowpassfilter = Filter_new(&low_pass_filter_parameters);

bandpassFilter = ComposedFilter_new(2, highpassfilter, lowpassfilter);
```

Here is listed the code to process signal with a composed filter.

```
void AnyClass_timerInterruptHandler(void)
{
primitiveSignal = Poxi_ADC->read_analog()

filteredSignal = bandpassFilter->process(bandpassFilter, primitiveSignal);

fprintf(file,"%.10f  %.10f\n", primitiveSignal, filteredSignal);
}
```

Deletion of a composed filter.

```
bandpassFilter->delete(&bandpassFilter);
```

The composed filter is able to obtain statistics when processing a signal. The following table lists the statistics flags.

Table 4. Composedfilter statistic flags

Flag	Description
STATISTICS_OFF	Turns off the statistics feature
MAX_OUTPUT	Obtains the maximum level of the output signal (processed signal)
MIN_OUTPUT	Obtains the minimum level of the output signal (processed signal)
MAX_INPUT	Obtains the maximum level of the input signal
MIN_INPUT	Obtains the minimum level of the input signal
FUNDAMENTAL_FREC	Obtains the fundamental frequency of the output signal.

Setup of statistics flags. To get proper statistics it is needed to set the sample time used for the design of the discreet filters. The statistics are setup by the usage of the statistics flags.

```
bandpassFilter->setSampleTime(bandpassFilter, DSP_SAMPLE_TIME);

bandpassFilter->setupStatistics(bandpassFilter,
                                FUNDAMENTAL_FREC | MAX_OUTPUT | MAX_INPUT | MIN_INPUT);
```

In order to obtain the statistics, it can be used the `getStatistics` member function.

```
static double fundFrec = bandpassFilter->getStatistics(bandpassFilter).fundamentalFrec;
sprintf(msg,"Heartbeat rate = %.2f bpm", fundFrec * 60);
GUICom_instance()->textMsg(0, msg);
```

For more detailed information it can be reviewed the actual source code.

2.2.3.1 Filter design and simulation

The pulse oximeter uses a DSP block implemented in software, this DSP is designed, calculated and simulated using MATLAB. The Butterworth filter is the filter type selected for filter coefficients calculation. MATLAB scripts are written to perform automated filter calculations and simulations.

The Butterworth filter calculation and its step and frequency responses are coded in the script file POXI_INIT.M.

```
clc
disp('Poxi initialization ...')
clear
format long
%%%%%%%%%%%%%%%%%%%%%%% Parameters %%%%%%%%%%%%%%%%%%%%%%%%
fh = 3;      % Low pass cut-off frecuency
Lowpass_Order = 6;
fl = 50/60;     % High pass cut-off frecuency
Highpass_Order = 4;
T = .001;     % Sample time

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Generate DSP.H for firmware implementation
 Butterworth_POXI(fh, Lowpass_Order, fl, Highpass_Order, T)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F = 1/T;

[BH,AH]=butter(Highpass_Order,fl/(F/2),'high');
[BL,AL]=butter(Lowpass_Order,fh/(F/2),'low');

%%%%%%%%%%%%%%%%%%%% Transfer functions %%%%%%%%%%%%%%%%%%%%
DHighPass = filt(BH,AH,T)
DLowPass = filt(BL,AL,T)

CHighPass = d2c(DHighPass);
CLowPass = d2c(DLowPass);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[BHC,AHC] = tfdata(CHighPass,'v');
[BLC,ALC] = tfdata(CLowPass,'v');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Frecuency responce
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% High Pass filter
w = linspace(0, 2*fl, 100)';
magH = freqresp(DHighPass, w, 'Hz');
magH = abs(magH(:));

figure
subplot(6,2,1);
plot(w,magH);
grid
title('POXI - High pass filter frequency response')
xlabel('Hz');
ylabel('Mag');
subplot(6,2,2);
step(DHighPass);
% Low Pass filter
w = linspace(0, 10*fh, 100)';
magL = freqresp(DLowPass, w, 'Hz');
magL = abs(magL(:));
subplot(6,2,3);
plot(w,magL);
grid
title('POXI - Low pass filter frequency response')
xlabel('Hz');
ylabel('Mag');
subplot(6,2,4);
step(DLowPass);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
load ..\JAVA_APP\poxi\poxi_adc.dat
subplot(6,2,[5,6]);
time = poxi_adc(:,1);
u    = poxi_adc(:,2);
plot(time,u)
title('POXI ADC SIGNAL')
ylabel('Mag');
```

```
legend('ADC Original')
grid

subplot(6,2,[7,8]);
plot(time,filter(BH,AH,filter(BL,AL,u)));
title('LOW-HIGH PASS FILTERED SIGNAL')
ylabel('Mag');
legend('Low High Pass Filtered')
grid

subplot(6,2,[9,10]);
plot(time,filter(BL,AL,u));
title('LOW PASS FILTERED SIGNAL')
ylabel('Mag');
legend('Low pass Filtered')
grid

subplot(6,2,[11,12]);
plot(time,filter(BH,AH,u));
title('HIGH PASS FILTERED SIGNAL')
xlabel('Time (S)');
ylabel('Mag');
legend('High pass Filtered')
grid

%signal.time = (T:T:length(u)*T)';
signal.time = time;
signal.signals.values = [u];
signal.signals.dimensions = 1;

disp('Done !')
```

At the beginning of the script, it was defined the filter parameters for design: Low pas filter cut-off frequency = 3Hz, filter order = 6; High pass filter cut-off frequency = 0.833Hz, filter order = 4. Sample time T = 0.001 Seconds.

```
%%%%%%%%%%%%%%%%%%%%%%% Parameters %%%%%%%%%%%%%%%%%%%%%%%%
fh = 3;      % Low pass cut-off frecuency
Lowpass_Order = 6;
fl = 50/60;     % High pass cut-off frecuency
Highpass_Order = 4;
T = .001;     % Sample time
```

The following figure displays the graphs obtained from the script. The script has access to real input data (POXI ADC SIGNAL) stored in a file for simulation purposes, from this data after its processing it is obtained the heartbeat signal, it can be seen as well in the next graph.

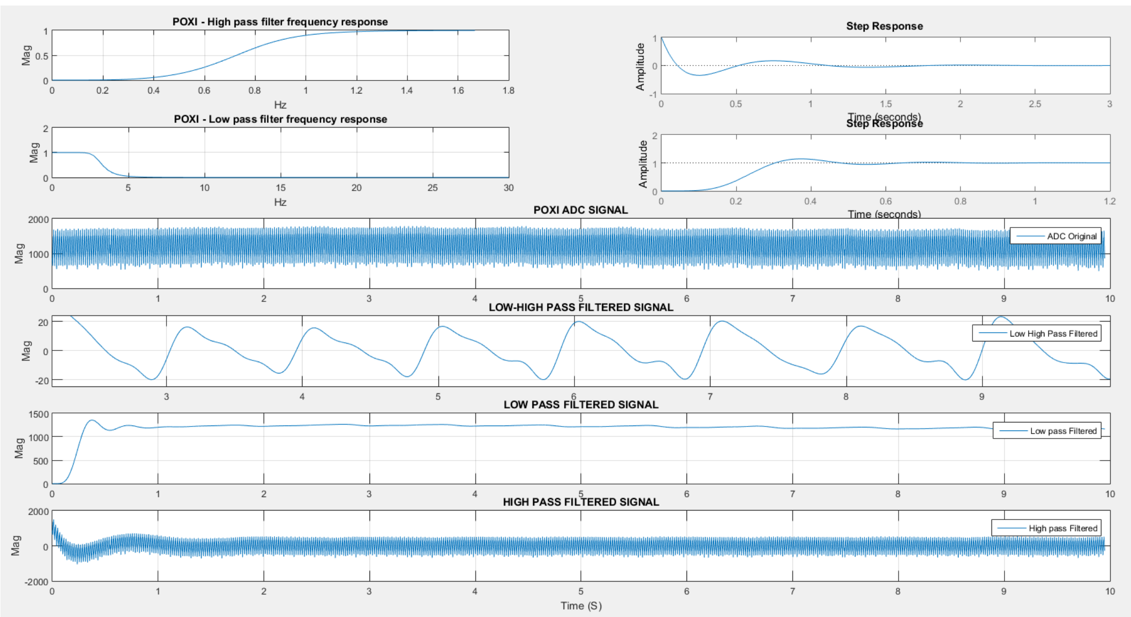


Figure 34. DSP - Design characteristics

```
Poxi initialization ...
```

```
Poxi filter design ...
Generating "..\poxi.sdk\poxi\src\dsp\DSP.H" ...
Done !

DHighPass =

   0.9932 - 3.973 z^-1 + 5.959 z^-2 - 3.973 z^-3 + 0.9932 z^-4
  -------------------------------------------------------------
    1 - 3.986 z^-1 + 5.959 z^-2 - 3.959 z^-3 + 0.9864 z^-4

Sample time: 0.001 seconds
Discrete-time transfer function.


DLowPass =

   6.759e-13 + 4.055e-12 z^-1 + 1.014e-11 z^-2 + 1.352e-11 z^-3 + 1.014e-11 z^-4 + 4.055e-12 z^-5 + 6.759e-13 z^-6
  ---------------------------------------------------------------------------------------------------------------
                 1 - 5.927 z^-1 + 14.64 z^-2 - 19.28 z^-3 + 14.29 z^-4 - 5.646 z^-5 + 0.9298 z^-6

Sample time: 0.001 seconds
Discrete-time transfer function.

Done !
```

The results from the previous script are plotted and given to more scripts for automated filer coefficient header file (for DSP firmware implementation) and legacy code instances (for MATLAB Simulink).

Automated generation of filter coefficient header file. The next MATLAB script receives parameter designs and generates DPS.H containing the filter coefficients used for the DSP software implementation.

```
unction Butterworth_POXI(fh, Lowpass_Order, fl, Highpass_Order, T)
% Butterworth IIR filter design.
%
% fh             = Low pass cut-off frecuency
% Lowpass_Order  = Low pass filter order
% fl             = High pass cut-off frecuency
% Highpass_Order = Low pass filter order
% T              = Sample time

if nargin ~= 5
    error('We need five arguments !');
end

disp('Poxi filter design ...')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
format long
F = 1/T;

[BH,AH]=butter(Highpass_Order,fl/(F/2),'high');
[BL,AL]=butter(Lowpass_Order,fh/(F/2),'low');

disp('Generating "..\poxi.sdk\poxi\src\dsp\DSP.H" ...')
file = fopen('..\poxi.sdk\poxi\src\dsp\DSP.H', 'w');
fprintf(file, '/*************************************************************************/\n');
fprintf(file, '/**                                                                     **/\n');
fprintf(file, '/** DSP.H                                                               **/\n');
fprintf(file, '/**                                                                     **/\n');
fprintf(file, '/** Butterworth IIR filter design.                                      **/\n');
fprintf(file, '/** This file was auto generated by a MATLAB script.                    **/\n');
fprintf(file, '/**                                                                     **/\n');
fprintf(file, '/** Created on: %s                                              **/\n', date());
fprintf(file, '/**                                                                     **/\n');
fprintf(file, '/** Author: Yarib Nevárez                                               **/\n');
fprintf(file, '/**         yarib_007@hotmail.com                                       **/\n');
fprintf(file, '/**                                                                     **/\n');
fprintf(file, '/*************************************************************************/\n');
fprintf(file, '\n#ifndef DSP_H_');
fprintf(file, '\n#define DSP_H_');
fprintf(file, '\n/*************************************************************************/\n');
fprintf(file, '#define DSP_SAMPLE_TIME          %.16f', T);
fprintf(file, '\n/*************************************************************************/\n\n');
fprintf(file, '#define HIGH_PASS_CUTOFF_FREC_HZ  %.2f \n', fl);
fprintf(file, '#define HIGH_PASS_FILTER_ORDER    %d \n\n', Highpass_Order);

for x = 1:length(AH)
    fprintf(file, '#define F0_A%d  ((const double)%.16d)\n', x-1, AH(x));
end

fprintf(file, '\n');
for x = 1:length(BH)
    fprintf(file, '#define F0_B%d  ((const double)%.16d)\n', x-1, BH(x));
end

fprintf(file, '\nconst double A_high_pass [] = {');
for x = 1:(length(AH)-1)
    fprintf(file, 'F0_A%d, ', x-1);
```

```
end
fprintf(file, 'F0_A%d};',x);

fprintf(file, '\nconst double B_high_pass [] = {');
for x = 1:(length(BH)-1)
    fprintf(file, 'F0_B%d, ', x-1);
end
fprintf(file, 'F0_B%d};',x);

fprintf(file,  '\nconst  FilterParameters  high_pass_filter_parameters  =  {HIGH_PASS_FILTER_ORDER,  A_high_pass,
B_high_pass};');
fprintf(file, '\n\n/*****************************************************************************/\n');
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fprintf(file, '\n');
fprintf(file, '#define LOW_PASS_CUTOFF_FREC_HZ   %.2f \n', fh);
fprintf(file, '#define LOW_PASS_FILTER_ORDER     %d \n\n', Lowpass_Order);
for x = 1:length(AL)
    fprintf(file, '#define F1_A%d   ((const double)%.16d)\n', x-1, AL(x));
end

fprintf(file, '\n');
for x = 1:length(BL)
    fprintf(file, '#define F1_B%d   ((const double)%.16d)\n', x-1, BL(x));
end

fprintf(file, '\nconst double A_low_pass [] = {');
for x = 1:(length(AL)-1)
    fprintf(file, 'F1_A%d, ', x-1);
end
fprintf(file, 'F1_A%d};',x);

fprintf(file, '\nconst double B_low_pass [] = {');
for x = 1:(length(BL)-1)
    fprintf(file, 'F1_B%d, ', x-1);
end
fprintf(file, 'F1_B%d};',x);

fprintf(file,  '\nconst  FilterParameters  low_pass_filter_parameters  =  {LOW_PASS_FILTER_ORDER,  A_low_pass,
B_low_pass};');

fprintf(file, '\n\n/*****************************************************************************/\n');
fprintf(file, '\n#endif /* DSP_H_ */');
fclose(file);

disp('Done !')
```

From the previous given parameter, the script result is listed below as C code header file, DSP.H.

```
/****************************************************************************/
/**                                                                        **/
/** DSP.H                                                                  **/
/**                                                                        **/
/** Butterworth IIR filter design.                                         **/
/** This file was auto generated by a MATLAB script.                       **/
/**                                                                        **/
/** Created on: 16-Mar-2017                                                **/
/**                                                                        **/
/** Author: Yarib Nevárez                                                  **/
/**         yarib_007@hotmail.com                                          **/
/**                                                                        **/
/****************************************************************************/

#ifndef DSP_H_
#define DSP_H_
/****************************************************************************/
#define DSP_SAMPLE_TIME          0.0010000000000000
/****************************************************************************/

#define HIGH_PASS_CUTOFF_FREC_HZ  0.83
#define HIGH_PASS_FILTER_ORDER    4

#define F0_A0   ((const double)0000000000000001)
#define F0_A1   ((const double)-3.9863177122115889e+00)
#define F0_A2   ((const double)5.9590466614474735e+00)
#define F0_A3   ((const double)-3.9591398122141528e+00)
#define F0_A4   ((const double)9.8641086372476394e-01)

#define F0_B0   ((const double)9.9318219059987367e-01)
#define F0_B1   ((const double)-3.9727287623994947e+00)
#define F0_B2   ((const double)5.9590931435992420e+00)
#define F0_B3   ((const double)-3.9727287623994947e+00)
#define F0_B4   ((const double)9.9318219059987367e-01)

const double A_high_pass [] = {F0_A0, F0_A1, F0_A2, F0_A3, F0_A4};
const double B_high_pass [] = {F0_B0, F0_B1, F0_B2, F0_B3, F0_B4};
```

```
const   FilterParameters   high_pass_filter_parameters   =   {HIGH_PASS_FILTER_ORDER,   A_high_pass,
B_high_pass};

/****************************************************************************/

#define LOW_PASS_CUTOFF_FREC_HZ   3.00
#define LOW_PASS_FILTER_ORDER     6

#define F1_A0   ((const double)0000000000000001)
#define F1_A1   ((const double)-5.927171119954841le+00)
#define F1_A2   ((const double)1.463850288710805 5e+01)
#define F1_A3   ((const double)-1.928223946869570 0e+01)
#define F1_A4   ((const double)1.428741321401181 5e+01)
#define F1_A5   ((const double)-5.646263961598141 8e+00)
#define F1_A6   ((const double)9.297584491720694 0e-01)

#define F1_B0   ((const double)6.758760218161796 7e-13)
#define F1_B1   ((const double)4.055256130897078 0e-12)
#define F1_B2   ((const double)1.013814032724269 5e-11)
#define F1_B3   ((const double)1.351752043632359 3e-11)
#define F1_B4   ((const double)1.013814032724269 5e-11)
#define F1_B5   ((const double)4.055256130897078 0e-12)
#define F1_B6   ((const double)6.758760218161796 7e-13)

const double A_low_pass [] = {F1_A0, F1_A1, F1_A2, F1_A3, F1_A4, F1_A5, F1_A6};
const double B_low_pass [] = {F1_B0, F1_B1, F1_B2, F1_B3, F1_B4, F1_B5, F1_B6};
const FilterParameters low_pass_filter_parameters = {LOW_PASS_FILTER_ORDER, A_low_pass, B_low_pass};

/****************************************************************************/

#endif /* DSP_H_ */
```

Once the header file is modified, it is detected by the SDK and it recompiles the target code for the ARM core. The symbols (variables and constants names) utilized in the header file are already employed in the POXI implementation, it is important to keep or maintain these symbols.

Finally, the following script takes the DSP library (firmware C language implementation) and generates the filter blocks for MATLAB Simulink.

```
% create legacy mex function for cart control
clc

% High Pass Filter
specs_hpf = legacy_code('initialize');
specs_hpf.SourceFiles = {'MATLAB.C', 'filter.c'};
specs_hpf.SrcPaths={'..\poxi.sdk\poxi\src\dsp\'};
specs_hpf.HeaderFiles = {'MATLAB.H', 'filter.h'};
specs_hpf.IncPaths={'..\poxi.sdk\poxi\src\dsp\'};
specs_hpf.SFunctionName = 'High_Pass_Filter';
specs_hpf.OutputFcnSpec = 'double y1 = high_pass_filter_wrapper(double u1)';
legacy_code('sfcn_cmex_generate', specs_hpf)
legacy_code('compile', specs_hpf)
%legacy_code('slblock_generate', specs_hpf)  % Comment out this line


% LOw Pass Filter
specs_lpf = legacy_code('initialize');
specs_lpf.SourceFiles = {'MATLAB.C', 'filter.c'};
specs_lpf.SrcPaths={'..\poxi.sdk\poxi\src\dsp\'};
specs_lpf.HeaderFiles = {'MATLAB.H', 'filter.h'};
specs_lpf.IncPaths={'..\poxi.sdk\poxi\src\dsp\'};
specs_lpf.SFunctionName = 'Low_Pass_Filter';
specs_lpf.OutputFcnSpec = 'double y1 = low_pass_filter_wrapper(double u1)';
legacy_code('sfcn_cmex_generate', specs_lpf)
legacy_code('compile', specs_lpf)
%legacy_code('slblock_generate', specs_lpf) % Comment out this line

disp('Poxi filter C legacy... Done')
```

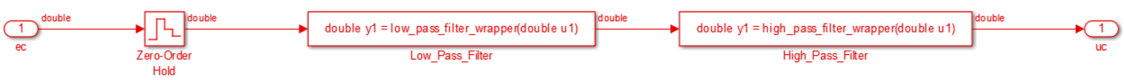


Figure 35. DSP Legacy code – MATLAB Simulink blocks

The next lines of code implement the filter instances contained in the legacy block.

```
/*
 * MATLAB.C
 *
 *  Created on: 26 de ene. de 2017
 *      Author: Yarib Nevárez
 */
#include "filter.h"
#include "DSP.H"

double high_pass_filter_wrapper(double u1)
{
    static Filter * filter = (Filter *)0;

    if (filter == 0)
    {
        filter = Filter_new(&high_pass_filter_parameters);
    }

    return filter->process(filter, u1);
}

double low_pass_filter_wrapper(double u1)
{
    static Filter * filter = (Filter *)0;

    if (filter == 0)
    {
        filter = Filter_new(&low_pass_filter_parameters);
    }

    return filter->process(filter, u1);
}
```

The generated filter blocks are used in a simulation, the simulation has three independent filter blocks, continues time, discrete time, and legacy code. It displays the filter results of all these for comparison.

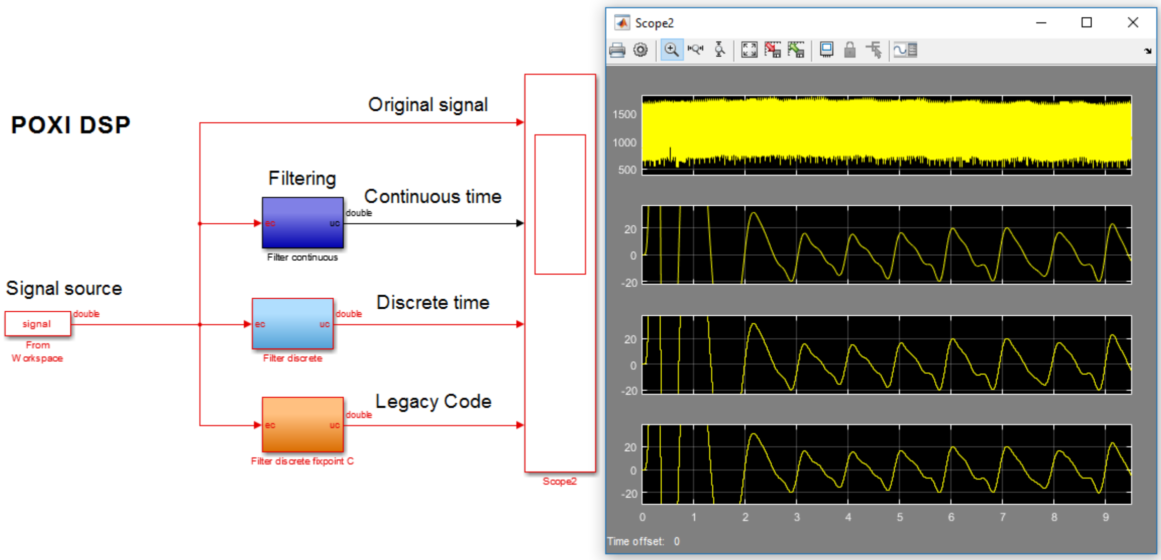


Figure 36. POXI DSP - MATLAB Simulink

2.2.4 Graphics

The graphics library is intended to provide capabilities to drive the TFT color display, it implements the low level driver SPI communication, data and command transmission to the LCD, and presents the middleware graphics layer. The final result is a library to perform drawing of basic graphics, with colours and shapes like circles, lines, rectangles, text, etc.

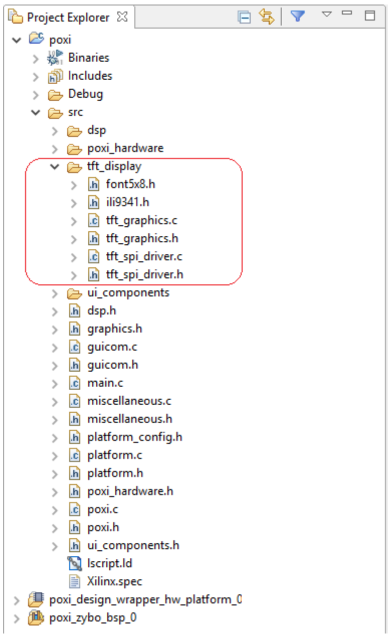


Figure 37. Graphics file tree structure

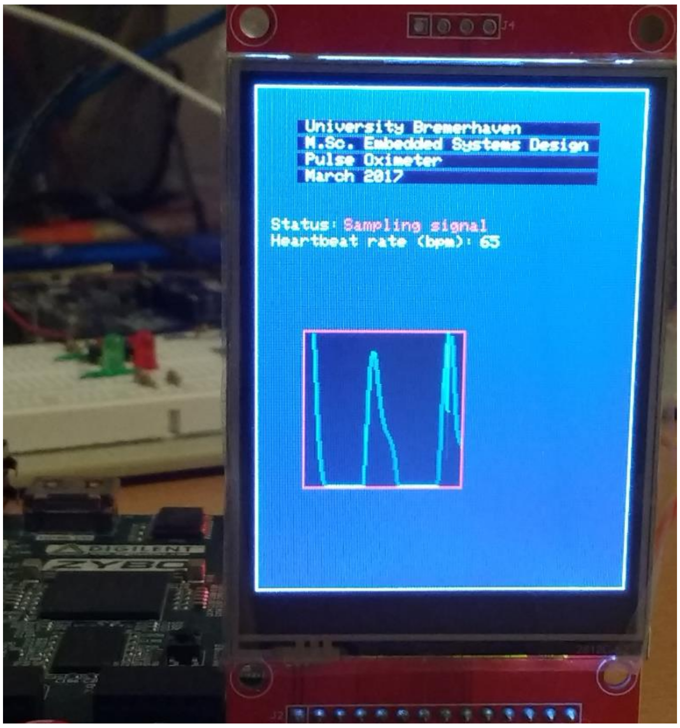


Figure 38. TFT LCD - Graphics

2.2.4.1 SPI driver for ILI9341

The SPI drivers for the ILI9341 controller is the lowest abstraction level of the graphics library, this implements the SPI setup, for sending data and the graphics commands. The code is implemented in tft_spi_driver.c.

```
/*
 * tft_spi_driver.c
 *
 *  Created on: 2 de dic. de 2016
 *      Author: Yarib Nevárez
 */
#include "xparameters.h"
#include "TFT_SPI_DISPLAY_240x320.h"
#include "tft_spi_driver.h"

void tft_spi_initialize(void)
{
    SET_TFT_SPI_BAUD_RATE_DIVIDER(0xFF);
    SET_TFT_SPI_SETTLE_TIME(3);

    SET_TFT_DATA_COMMAND(1);

    SET_TFT_SPI_CLOCK_POLARITY(1);
    SET_TFT_SPI_CLOCK_PHASE(1);

    SET_TFT_CS_FORCE(0);
    SET_TFT_SPI_DATA_LENGTH(0);
}

void tft_spi_baud_rate(uint8_t baud_rate)
{
    SET_TFT_SPI_BAUD_RATE_DIVIDER(baud_rate);
    SET_TFT_SPI_SETTLE_TIME(2);
}

void tft_hardware_initialize(void)
{
    int nop = 10000;
    SET_TFT_DATA_COMMAND(1);
    SET_TFT_RESET(0);
    while(nop--);
    SET_TFT_RESET(1);
}

void tft_spi_write_data(uint8_t data)
{
```

```
    while(!GET_TFT_TRANSMISSION_DONE);
    SET_TFT_SPI_DATA_LENGTH(0);

    TFT_SPI_DATA = data;
}

void tft_spi_write_data16(uint16_t word)
{
    while(!GET_TFT_TRANSMISSION_DONE);
    SET_TFT_SPI_DATA_LENGTH(1);

    TFT_SPI_DATA = word;
}

void tft_spi_write_command(uint8_t cmd)
{
    SET_TFT_DATA_COMMAND(0);
    tft_spi_write_data(cmd);
    SET_TFT_DATA_COMMAND(1);
}
```

In the previous code it can be seen that the SPI configuration: CPOL = 1, CPHA = 1, 8 and 16 bits.

2.2.4.2 Graphics routines library

The graphics library provides the middle ware software layer for graphics. The routines to draw pixels, lines, rectangles, circles, text, etc, are implemented in this layer. This layer is implemented in a C++ style using OOP design, the class is defined in tft_graphics.h.

The next code exhibit the class definition, it has reduced font size for better visual-overview.

```
/*
 * tft_graphics.h
 *
 *  Created on: 2 de dic. de 2016
 *      Author: Yarib Nevárez    yarib_007@hotmail.com
 */

#ifndef SRC_TFT_DISPLAY_TFT_GRAPHICS_H_
#define SRC_TFT_DISPLAY_TFT_GRAPHICS_H_

#include "xil_types.h"
#include "ili9341.h"

typedef enum {
    ROT0 = 0,   // Portrait
    ROT90 = 1,  // Landscape
    ROT180 = 2, // Flipped portrait
    ROT270 = 3  // Flipped landscape
} TFTGraphics_Rotation;

typedef struct
{
    void (*initialize)      (void);
    void (*speed)           (uint8_t rate);
    void (*setAddress)      (uint16_t x1,uint16_t y1,uint16_t x2,uint16_t y2);
    void (*drawRectFilled)  (uint16_t x, uint16_t y, uint16_t w, uint16_t h, uint16_t colour);
    void (*drawPixel)       (uint16_t x, uint16_t y, uint16_t colour);
    void (*drawPixel_2)     (uint16_t x, uint16_t y, uint8_t size, uint16_t colour);
    void (*drawLine)        (uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1, uint16_t colour);
    void (*drawRect)        (uint16_t x,uint16_t y,uint16_t w,uint16_t h,uint16_t colour);
    void (*drawClear)       (uint16_t colour);
    void (*setRotation)     (TFTGraphics_Rotation rotation);
    void (*drawCircle)      (uint16_t poX, uint16_t poY, uint16_t radius, uint16_t colour);
    void (*drawChar)        (uint16_t x, uint16_t y, char c, uint8_t size, uint16_t colour, uint16_t bg);
    void (*drawString)      (uint16_t x, uint16_t y, const char *string, uint8_t size, uint16_t colour, uint16_t bg);
    void (*setupScrollArea) (uint16_t TFA, uint16_t BFA);
    void (*scrollAddress)   (uint16_t VSP);
    int  (*scrollLine)      (void);
} TFTGraphics;

TFTGraphics * TFTGraphics_instance(void);

#endif /* SRC_TFT_DISPLAY_TFT_GRAPHICS_H_ */
```

The following lines show the initialization procedure, this code initializes the TFT LCD and draws a red line in diagonal of the screen.

```
static uint8_t Poxi_initGraphics(void)
{
    Poxi_graphics = TFTGraphics_instance();
    uint8_t rc = Poxi_graphics != NULL;

    if (rc)
    {
```

```
        Poxi_graphics->initialize();
        Poxi_graphics->speed(0xF0); // Regular SPI speed
        Poxi_graphics->drawLine(00, 00, ILI9341_WIDTH, ILI9341_HEIGHT, RED); // TEST!
        Poxi_graphics->speed(0xFE); // Speed up !!!
    }

    return rc;
}
```

For more detailed information it can be referred to the actual code.

2.2.5 UI components

The UI components library basically provides high level software classes for graphical user interface, there are three components: Frame Panel, Plot 2D, and text label. All UI components extends a base class Widget, this base class defines the basic interface for the extended components. The extended components basically utilize the software design patterns of factory, and composed (class extension), encapsulation and polymorphism. It can be created any number of UI components as needed, and they can be treated as base widgets having their own behaviour (polymorphism). The code is implemented in C++ style and uses dynamic memory allocation for the component instantiation.

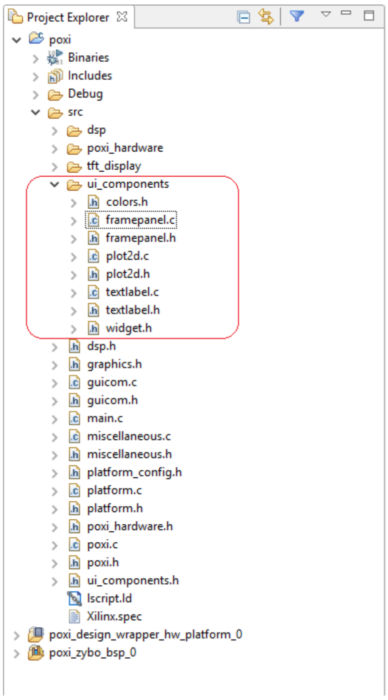


Figure 39. UI components File tree

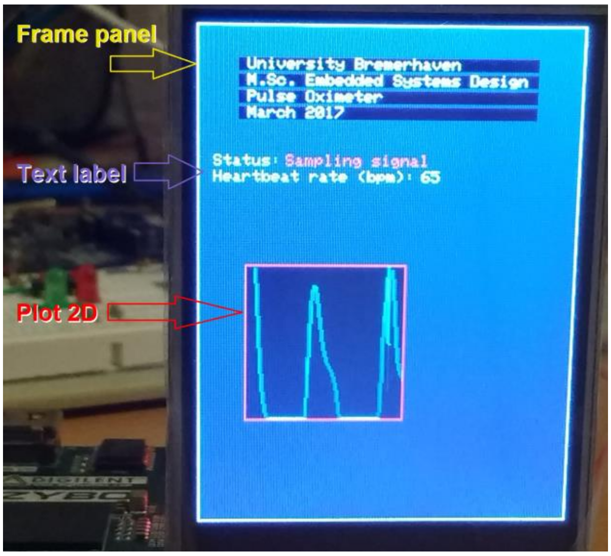


Figure 40. UI components in TFT LCD

2.2.5.1 Widget

The widget class serves as the base UI component class, and it defines the basic component interface. It can be said that all and every UI component is a widget. This is an abstract class and cannot be instantiated.

```
/*
 * widget.h
 *
 *  Created on: 9 de ene. de 2017
 *      Author: Yarib Nevárez
 */

#ifndef SRC_UI_COMPONENTS_WIDGET_H_
#define SRC_UI_COMPONENTS_WIDGET_H_

#include "xil_types.h"

typedef struct Widget_public Widget;

typedef void (* WidgetDrawFunc)    (Widget * obj);
typedef void (* WidgetRefreshFunc) (Widget * obj);
typedef void (* WidgetClearFunc)   (Widget * obj);
typedef void (* WidgetDeleteFunc)  (Widget ** obj);
```

```
#define WIDGET_VIRTUALTABLE_MEMBERS \
        WidgetDrawFunc    draw; \
        WidgetRefreshFunc refresh; \
        WidgetClearFunc   clear; \
        WidgetDeleteFunc  delete; \
        Widget *          parent; \
        Widget *          prev; \
        Widget *          next; \
        Widget *          first_child;


struct Widget_public
{
    WIDGET_VIRTUALTABLE_MEMBERS
};

#endif /* SRC_UI_COMPONENTS_WIDGET_H_ */
```

2.2.5.2 Plot 2D

The Plot 2D is a widget intended to display data values in a graphical manner, the data is inserted as a dot and it will be displayed in the plot, then the following added dots will be displayed and connected by a line giving the effect of a line graph. This class is defined in plot2d.h.

```
/*
 * plot2d.h
 *
 *  Created on: 3 de ene. de 2017
 *      Author: Yarib Nevárez
 */

#ifndef PLOT2D_H_
#define PLOT2D_H_

#include "xil_types.h"
#include "widget.h"

typedef struct Plot2D_public Plot2D;

#define PLOT2D_VIRTUALTABLE_MEMBERS \
        WIDGET_VIRTUALTABLE_MEMBERS \
        void     (* add_point) (Plot2D * obj, uint16_t value, uint32_t color);

struct Plot2D_public
{
    PLOT2D_VIRTUALTABLE_MEMBERS
};

Plot2D * Plot2D_new(uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint32_t line_color,
uint32_t background_color);

#endif /* PLOT2D_H_ */
```

The important function members of this class are the draw function and add_point. These functions are exhibited below.

```
static void Plot2D_draw(Plot2D * obj)
{
    if (obj != NULL)
    {
        Plot2D_private * thiz = (Plot2D_private *) obj;
        TFTGraphics *    canvas = thiz->canvas;
        if ((thiz->plot2D_space != NULL) && (canvas != NULL))
        {
            int16_t i;

            canvas->drawRectFilled(thiz->x_pos + 1, thiz->y_pos + 1, thiz->size,
                                   thiz->height, thiz->background_color);

            canvas->drawRect(thiz->x_pos, thiz->y_pos, thiz->size + 1, thiz->height + 1,
                             thiz->line_color);

            for (i = 0; i < thiz->index - 1; i ++)
            {
                canvas->drawLine(thiz->x_pos + i + 1,
                         thiz->y_pos + thiz->height - thiz->plot2D_space[i].y,
                         thiz->x_pos + i + 1 + 1,
                         thiz->y_pos + thiz->height - thiz->plot2D_space[i+1].y,
                         thiz->plot2D_space[i].c);
```

```
            }

            memcpy(thiz->plot2D_old_space,
                   thiz->plot2D_space,
                   sizeof(Point2D) * thiz->index);

            thiz->old_index = thiz->index;
        }
    }
}


static void Plot2D_add(Plot2D * obj, uint16_t y, uint32_t c)
{
    if (obj != NULL)
    {
        Plot2D_private * thiz = (Plot2D_private *) obj;
        if(thiz->plot2D_space != NULL)
        {
            if (thiz->height <= y)
            {
                y = 0;
                c = thiz->background_color;
            }

            if (thiz->index < thiz->size)
            {
                thiz->plot2D_space[thiz->index].y = y;
                thiz->plot2D_space[thiz->index].c = c;

                thiz->index ++;
            }
            else
            {
                memcpy(thiz->plot2D_space,
                       thiz->plot2D_space + 1,
                       sizeof(Point2D)*(thiz->size - 1));

                thiz->plot2D_space[thiz->size - 1].y = y;
                thiz->plot2D_space[thiz->size - 1].c = c;
            }
            Plot2D_refresh(obj);
        }
    }
}
```

2.2.5.3 Text label

The text label is a widget intended to display text, it provides the functionality to set text and back ground colours, size and clean refresh of its graphic content. This class is defined in the textlabel.h.

The next lines of code shows the class definition.

```
/*
 * textlabel.h
 *
 *  Created on: 4 de ene. de 2017
 *      Author: Yarib Nevárez
 */

#ifndef TEXTLABEL_H_
#define TEXTLABEL_H_

#include "xil_types.h"
#include "widget.h"

typedef struct TextLabel_public TextLabel;

typedef enum
{
    TEXT,
    NUMBER
} TextLabel_Type;

#define TEXTLABEL_VIRTUALTABLE_MEMBERS \
        WIDGET_VIRTUALTABLE_MEMBERS \
        void        (* set_x)     (TextLabel * obj, int32_t x_pos); \
        void        (* set_y)     (TextLabel * obj, int32_t y_pos); \
        void        (* set_width) (TextLabel * obj, int32_t width); \
```

```
        void        (* set_height)(TextLabel * obj, int32_t height); \
        void        (* set_text)  (TextLabel * obj, char * str); \
        void        (* set_number)(TextLabel * obj, int32_t number); \
        void        (* set_label_type)      (TextLabel * obj, TextLabel_Type type); \
        void        (* set_font_size)       (TextLabel * obj, uint8_t font_size); \
        void        (* set_text_color)      (TextLabel * obj, int32_t text_color); \
        void        (* set_background_color)(TextLabel * obj, int32_t background_color);

struct TextLabel_public
{
    TEXTLABEL_VIRTUALTABLE_MEMBERS
};

TextLabel * TextLabel_new (uint16_t x, uint16_t y, uint16_t width, uint16_t height, char * str,
uint32_t text_color, uint32_t background_color);


#endif /* TEXTLABEL_H_ */
```

2.2.5.4 Frame panel

The frame panel is a widget container, it holds other widgets as children in an internal tree data structure, and every leaf is a widget (Plot 2D, Text label, or even a Frame Panel). The widget tree can grow as bigger as needed. The Frame panel also displays border and background colours. The frame panel class is defined in framepanel.h.

```
/*
 * framepanel.h
 *
 *  Created on: 10 de ene. del 2017
 *      Author: Yarib Nevárez
 */

#ifndef SRC_UI_COMPONENTS_FRAMEPANEL_H_
#define SRC_UI_COMPONENTS_FRAMEPANEL_H_

#include "xil_types.h"
#include "widget.h"

#define MAX_HEIGHT  320
#define MAX_WIDTH   240

typedef struct FramePanel_public FramePanel;

#define FRAMEPANEL_VIRTUALTABLE_MEMBERS \
        WIDGET_VIRTUALTABLE_MEMBERS \
        void (* set_x)      (FramePanel * obj, int32_t x_pos); \
        void (* set_y)      (FramePanel * obj, int32_t y_pos); \
        void (* set_width)  (FramePanel * obj, int32_t width); \
        void (* set_height)(FramePanel * obj, int32_t height); \
        void (* set_lineColor)       (FramePanel * obj, int32_t text_color); \
        void (* set_backgroundColor)(FramePanel * obj, int32_t background_color);\
        void (* give_widget)         (FramePanel * obj, Widget * child);

struct FramePanel_public
{
    FRAMEPANEL_VIRTUALTABLE_MEMBERS
};

FramePanel * FramePanel_new (uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint32_t
line_color, uint32_t background_color);


#endif /* SRC_UI_COMPONENTS_FRAMEPANEL_H_ */
```

The following lines of code revels the easiness in the usage of UI components in the POXI application. This code shows the instantiation and deletion of the UI components.

```
typedef struct
{
    FramePanel * framePanel;
    TextLabel  * statusLabel;
    TextLabel  * hbrLabel;
    Plot2D     * tracePlot;
} UIContent;

static UIContent Poxi_UI;


#define UI_LABEL_QTY (6)
```

```c
static uint8_t Poxi_initUIContent(void)
{
    FramePanel * frame = FramePanel_new(0, 0, MAX_WIDTH -1, MAX_HEIGHT -1, WHITE, NAVY);
    TextLabel  * statusLabel = TextLabel_new(55, 80, 0, 0, "Starting Up", PINK, NAVY);
    TextLabel  * hbrLabel = TextLabel_new(140, 90, 0, 0, "-", WHITE, NAVY);
    Plot2D     * plot = Plot2D_new(30, 150, 100, 100, RED, BLACK);
    TextLabel  * label[UI_LABEL_QTY];
    uint8_t      i;
    uint8_t      rc;

    label[0] = TextLabel_new(25, 20, 0, 0, " University Bremerhaven        ", YELLOW, BLACK);
    label[1] = TextLabel_new(25, 30, 0, 0, " M.Sc. Embedded Systems Design ", WHITE, BLACK);
    label[2] = TextLabel_new(25, 40, 0, 0, " Pulse Oximeter                ", WHITE, BLACK);
    label[3] = TextLabel_new(25, 50, 0, 0, " March 2017                    ", WHITE, BLACK);

    label[4] = TextLabel_new(10, 80, 0, 0, "Status:", YELLOW, NAVY);
    label[5] = TextLabel_new(10, 90, 0, 0, "Heartbeat rate (bpm):", YELLOW, NAVY);

    rc = frame != NULL
      && statusLabel != NULL
      && hbrLabel != NULL
      && plot != NULL;

    for (i = 0; rc && i < sizeof(label)/sizeof(TextLabel *); i++)
        rc = rc && label[i] != NULL;

    if (rc)
    {
        frame->give_widget(frame, (Widget *)statusLabel);
        frame->give_widget(frame, (Widget *)plot);
        frame->give_widget(frame, (Widget *)hbrLabel);
        hbrLabel->set_label_type(hbrLabel, NUMBER);

        for (i = 0; i < sizeof(label)/sizeof(TextLabel *); i++)
            frame->give_widget(frame, (Widget *)label[i]);

        Poxi_UI.framePanel  = frame;
        Poxi_UI.statusLabel = statusLabel;
        Poxi_UI.tracePlot   = plot;
        Poxi_UI.hbrLabel    = hbrLabel;
    }

    return rc;
}

static uint8_t Poxi_run(void)
{
    Poxi_UI.framePanel->draw((Widget *)Poxi_UI.framePanel);

    Poxi_interruptDutyEnable = 1;
    do
    {
        Poxi_refreshUI();
        if (Poxi_ZYBO->switch_(0))
        {
            Poxi_JavaGUI_report();
        }
    } while (Poxi_ZYBO->switch_(3));
    Poxi_interruptDutyEnable = 0;

    Poxi_clip->light(LIGHTPROBE_OFF);
    Poxi_ZYBO->leds(0);

    Poxi_UI.statusLabel->set_text(Poxi_UI.statusLabel,"* STOP *");

    return 0;
}

static void Poxi_refreshUI(int variable)
{
    uint8_t hbr;

    if (Poxi_sampleStatus == UNAVAILABLE)
    {
        Poxi_UI.statusLabel->set_text_color(Poxi_UI.statusLabel, RED);
        Poxi_UI.statusLabel->set_text(Poxi_UI.statusLabel,"Sampling signal");
    }

    if (!Poxi_ZYBO->switch_(0))
    {
        Poxi_UI.tracePlot->add_point(Poxi_UI.tracePlot, variable*10, CYAN);
    }


    hbr = Poxi_DSP.ired->getStatistics(Poxi_DSP.ired).fundamentalFrec * 60;
    if (hbr <= 120)
    {
        if (hbr < 50)
            Poxi_UI.hbrLabel->set_text_color(Poxi_UI.hbrLabel, RED);
        else if (90 < hbr)
            Poxi_UI.hbrLabel->set_text_color(Poxi_UI.hbrLabel, MAGENTA);
        else
            Poxi_UI.hbrLabel->set_text_color(Poxi_UI.hbrLabel, WHITE);
```

```
        Poxi_UI.hbrLabel->set_number(Poxi_UI.hbrLabel, hbr);
    }
}

static void Poxi_dispose(void)
{
    //More code goes here
    if (Poxi_UI.framePanel != NULL)
        Poxi_UI.framePanel->delete((Widget **) &Poxi_UI.framePanel);
}
```

2.2.6 Command communication protocol - Data link layer

The command communication protocol is developed to serve as an interface between the POXI device and the desktop Java application. This object is a GUI communicator, it implements the command protocol layer for a proper serial communication, and it provides the interface or function members in a higher level software layer. The specific communication command are implemented and used in this layer class. This definition is in guicom.h.

```
/*
 * GUICom.h
 *
 *  Created on: 3 de feb. de 2017
 *      Author: Yarib Nevárez
 */

#ifndef SRC_GUICOM_H_
#define SRC_GUICOM_H_

#include "xil_types.h"

typedef enum
{
    TRACE_0 = 0,
    TRACE_1,
    TRACE_2,
    TRACE_3,
    TRACE_4,
    TRACE_5,
    TRACE_6,
    TRACE_7,
    TRACE_8,
    TRACE_9,
    TRACE_ALL = 0xFF
} GUITrace;

typedef uint8_t (* GUIProgressCallback)(void * data, uint32_t progress, uint32_t total);

typedef struct //class
{
    void    (* clearTrace)  (GUITrace trace);
    uint8_t (* plotSamples)(GUITrace trace, double * array, uint32_t length);
    void    (* setVisible)  (GUITrace trace, uint8_t visible);
    void    (* setTime)     (GUITrace trace, double time);
    void    (* setStepTime)(GUITrace trace, double time);
    void    (* textMsg)     (uint8_t id, char * msg);
    void    (* setProgressCallback)(GUIProgressCallback function, void * data);
} GUICom;

GUICom * GUICom_instance(void);

#endif /* SRC_GUICOM_H_ */
```

In this class definition it be seen the function member names, and it can be inferred their tasks. These function members implement the mechanisms for command communication by the usage of the serial port instance.

In the next lines is exhibited the source code of the GUI communicator (in reduced text font size).

```
/*
 * GUICom.c
 *
 *  Created on: 3 de feb. de 2017
 *      Author: Yarib Nevárez
 */

#include "guicom.h"
#include "poxi_hardware.h"
#include "miscellaneous.h"
#include "string.h"

static void    GUICom_clearTrace (GUITrace trace);
```

```
static uint8_t GUICom_plotSamples(GUITrace trace, double * array, uint32_t length);
static void    GUICom_setVisible (GUITrace trace, uint8_t visible);
static void    GUICom_setTime     (GUITrace trace, double time);
static void    GUICom_setStepTime(GUITrace trace, double time);
static void    GUICom_textMsg     (uint8_t id, char * msg);
static void    GUICom_setProgressCallback(GUIProgressCallback function, void * data);

static GUIProgressCallback progressCallback = NULL;
static void * progressCallbackData = NULL;

static uint8_t GUICom_doubleBulkLength = 16;

#define CMD_CLEAR          0x00
#define CMD_PLOT           0x01
#define CMD_SET_VISIBLE    0x02
#define CMD_SET_STEP_TIME 0x03
#define CMD_SET_TIME       0x04
#define CMD_TEXT_MSG       0x05

static GUICom GUICom_obj  = { GUICom_clearTrace,
                              GUICom_plotSamples,
                              GUICom_setVisible,
                              GUICom_setTime,
                              GUICom_setStepTime,
                              GUICom_textMsg,
                              GUICom_setProgressCallback };

GUICom * GUICom_instance(void)
{
    return & GUICom_obj;
}

static void    GUICom_clearTrace(GUITrace trace)
{
    uint8_t cmd[] = {CMD_CLEAR, 0};
    cmd[1] = trace;
    SerialPort_instance()->sendFrameCommand(cmd, sizeof(cmd), NULL, 0);
}

static uint8_t GUICom_plotSamples(GUITrace trace, double * array, uint32_t length)
{
    uint8_t rc;
    uint8_t  cmd[] = {CMD_PLOT, 0, 0};
    uint32_t i, len;

    cmd[1] = trace;

    rc = array != NULL;

    if (rc)
    {
        for (i = 0; i < length && rc; i += len)
        {
            if(i + GUICom_doubleBulkLength < length)
            {
                len = GUICom_doubleBulkLength;
            }
            else
            {
                len = length - i;
            }
            cmd[2] = len;
            SerialPort_instance()->sendFrameCommand(cmd,
                                                    sizeof(cmd),
                                                    (uint8_t *) &array[i],
                                                    sizeof(double) * len);

            if (progressCallback != NULL)
            {
                rc = progressCallback(progressCallbackData, i + len, length);
            }
            delay_ms(50);
        }
    }

    return rc;
}

static void    GUICom_setVisible (GUITrace trace, uint8_t visible)
{
    uint8_t cmd[] = {CMD_SET_VISIBLE, 0, 0};
    cmd[1] = trace;
    cmd[2] = visible;
    SerialPort_instance()->sendFrameCommand(cmd, sizeof(cmd), NULL, 0);
}

static void    GUICom_setTime     (GUITrace trace, double time)
{
    uint8_t cmd[] = {CMD_SET_TIME, 0};
    cmd[1] = trace;
    SerialPort_instance()->sendFrameCommand(cmd, sizeof(cmd), (uint8_t *)&time, sizeof(double));
}

static void    GUICom_setStepTime(GUITrace trace, double time)
{
    uint8_t cmd[] = {CMD_SET_STEP_TIME, 0};
    cmd[1] = trace;
    SerialPort_instance()->sendFrameCommand(cmd, sizeof(cmd), (uint8_t *)&time, sizeof(double));
}

static void    GUICom_textMsg     (uint8_t id, char * msg)
{
    uint8_t cmd[] = {CMD_TEXT_MSG, 0, 0};
    cmd[1] = id;
    cmd[2] = strlen(msg);
    SerialPort_instance()->sendFrameCommand(cmd, sizeof(cmd), (uint8_t *)msg, sizeof(char) * strlen(msg));
}

static void    GUICom_setProgressCallback(GUIProgressCallback function, void * data)
```

```
{
    progressCallback = function;
    progressCallbackData = data;
}
```

The protocol communication is a layer present in all the communicating points, in this case this layer is present in the POXI device and in the Java application in the desktop system. This based command communication must match in all the systems. The command send by a transmitting point must be understood with exactly the same meaning in all the receivers systems.

All the commands have as parameter the desired trace to affect in Java GUI, and the values of times and points are double precision numbers (64 bits). The following table enlist the commands implemented in this layer.

Table 5. GUI Communicator - Commands

Command	Hex value	Description
CMD_CLEAR	0x00	Clears the given trace
CMD_PLOT	0x01	Sends a batch of points to be traced
CMD_SET_VISIBLE	0x02	Makes visible or hides the given trace
CMD_SET_STEP_TIME	0x03	Sets the time interval of points of the given trace
CMD_SET_TIME	0x04	Sets the initial time of a given trace
CMD_TEXT_MSG	0x05	Sends a text message

The following image shows the traces in the Java GUI application. The commands are useful for clearing the traces, set the initial time and the interval times, plot the points and for sending text messages displayed at the bottom of the screen.

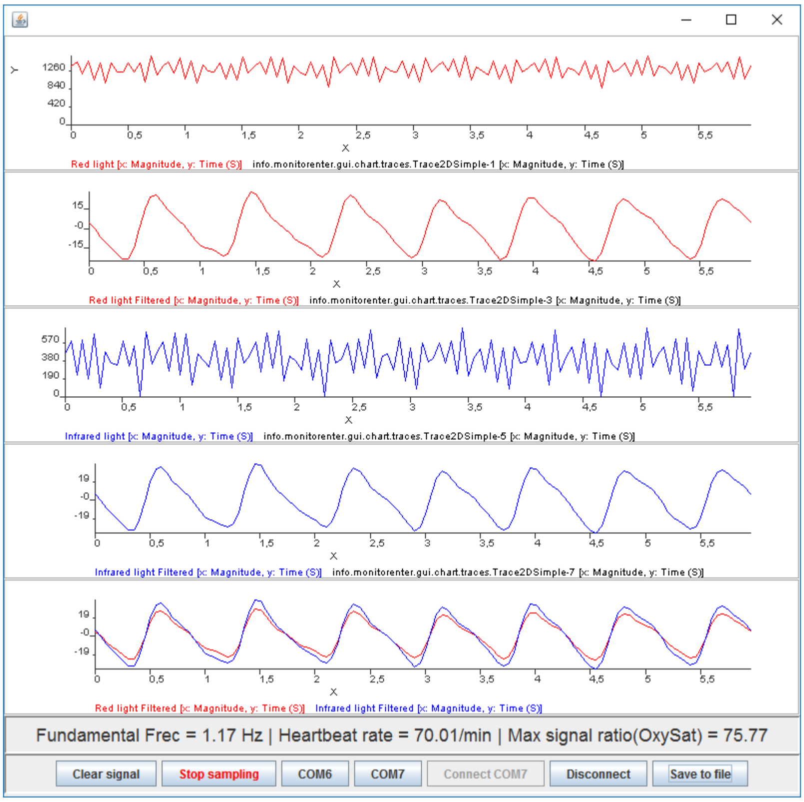


Figure 41. Java GUI – Traces

The next code illustrates the usage of the GUI communicator layer, it reduces the effort of the application by performing the tasks of command makeup and communication.

```
#define NUM_SAMPLES 3

static GUICom * Poxi_JavaGUICom = GUICom_instance();
static double GUISampleBuffer[NUM_SAMPLES];
static char    msg[120];
```

```
GUISampleBuffer[0] = 3.14159265;
GUISampleBuffer[1] = 0.0;
GUISampleBuffer[2] = -3.14159265;

Poxi_JavaGUICom->clearTrace(TRACE_ALL);
Poxi_JavaGUICom->setStepTime(TRACE_ALL, DSP_SAMPLE_TIME);
Poxi_JavaGUICom->plotSamples(TRACE_0, GUISampleBuffer, NUM_SAMPLES);

sprintf(msg," Heartbeat rate = %.2f bpm ", fundFrec * 60);
Poxi_JavaGUICom->textMsg(0,msg);
```

2.2.7 Poxi application

The POXI application is the main firmware solution for the POXI device, it is the top software layer, and it assembles many pieces from the application framework described in the previous paragraphs. The main tasks of the POXI application are described in the next list.

Data acquisition tasks.

- Sets up the timer interruption as the DSP sample time
- Takes sample of light absorbance in light sequence (off, red, off, infrared)
- Determines voltage levels for red, infrared lights, finger clip sensor bias voltage and DC level for ambient light subtraction.

Signal processing tasks.

- Instantiates composed band-pass filter for red light absorbance samples
- Instantiates composed band-pass filter for infrared light absorbance samples
- Takes statistics of the composed filters to obtain fundamental frequency and any other property

GUI tasks.

- Collects processed information and sends it to the TFT display and Java GUI

The POXI application is implemented in C++ style, its class is defined in poxi.h.

```
/*
 * poxi.h
 *
 *  Created on: 11 de feb. de 2017
 *      Author: Yarib Nevárez
 */

#ifndef SRC_POXI_H_
#define SRC_POXI_H_
#include "xil_types.h"

typedef struct
{
    uint8_t  (* initialize)(void);
    uint8_t  (* run)(void);
    void     (* dispose)(void);
} Poxi;

Poxi * Poxi_instance(void);
#endif /* SRC_POXI_H_ */
```

The class interface is exposed in the manner to be able to run in a real time operating system by interfacing initialization, run, and disposal function members.

The main function runs the POXI application.

```
/*
 * main.c
 *
 *  Created on: 13 de feb. de 2017
 *      Author: Yarib Nevárez
 */

#include "poxi.h"

int main()
{
```

```
    int rc;

    Poxi * App = Poxi_instance();

    rc = App != NULL;

    if (rc)
    {
        rc = App->initialize();

        if (rc)
        {
            rc = App->run();
        }

        App->dispose();
    }

    return rc;
}
```

The following paragraphs explain the main details of the POXI application.

2.2.7.1 Application initialization

The initialization of the POXI application perform the following tasks.

- Interrupt handlers
- Device instances and device start up.
- Graphics set up.
- Creation of UI content instances.
- Creation and setup of the signal processing blocks.

```
static uint8_t Poxi_initialize(void)
{
    uint8_t rc;

    init_platform();

    rc = Poxi_initInterruptHandlers();
    if (rc) rc = Poxi_initDeviceInstances();
    if (rc) rc = Poxi_initGraphics();
    if (rc) rc = Poxi_initUIContent();
    if (rc) rc = Poxi_initDSP();

    return rc;
}
```

The initialization of the device instances is exhibited below.

```
static uint8_t Poxi_initDeviceInstances()
{
        uint8_t rc;

        Poxi_JavaGUICom  = GUICom_instance();
        Poxi_DAC         = AD5624R_instance();
        Poxi_PGA         = MCP6S2X_instance();
        Poxi_ADC         = AD7887_instance();
        Poxi_ZYBO        = ZYBO_instance();
        Poxi_clip        = LightProbe_instance();
        Poxi_graphics    = TFTGraphics_instance();

        rc = Poxi_JavaGUICom != NULL
          && Poxi_DAC        != NULL
          && Poxi_PGA        != NULL
          && Poxi_ADC        != NULL
          && Poxi_ZYBO       != NULL
          && Poxi_clip       != NULL
          && Poxi_graphics   != NULL;

        if (rc)
        {
                Poxi_ADC->set_reference(REF_ENABLED);
                Poxi_ADC->set_channel_mode(SINGLE_CHANNEL);
                Poxi_ADC->set_power_mode(MODE2);

                Poxi_DAC->reset();
                Poxi_DAC->LDAC_setup(0);
                Poxi_DAC->power_mode(NORMAL_OPERATION, 0xF);
```

```
            Poxi_DAC->internal_reference(REFERENCE_ON);

            Poxi_DAQ.ampGain      = GAIN_32;
            Poxi_DAQ.enviantLight = 0x08FF;
            Poxi_DAQ.redLight     = 0x0FFF;
            Poxi_DAQ.iredLight    = 0x0FFF;
            Poxi_DAQ.clipSensor   = 0x0FFF;

            Poxi_JavaGUICom->setProgressCallback(Poxi_JavaGUI_progressCallback, NULL);
        }

        return rc;
}
```

The initialization of the signal processing block is exposed in the following code.

```
static uint8_t Poxi_initDSP(void)
{
        uint8_t rc;
        ComposedFilter * filterRed  = NULL;
        ComposedFilter * filterIred = NULL;

        memset(&Poxi_DSP, 0x00, sizeof(Poxi_DSP));

        filterRed = ComposedFilter_new(2,
                        Filter_new(&high_pass_filter_parameters),
                        Filter_new(&low_pass_filter_parameters));

        Poxi_DSP.red = filterRed;
        rc = filterRed != NULL;
        if (rc)
        {
                filterRed->setSampleTime(filterRed, DSP_SAMPLE_TIME);
                filterRed->setupStatistics(filterRed,
                                FUNDAMENTAL_FREC | MAX_OUTPUT | MAX_INPUT | MIN_INPUT);


                filterIred = ComposedFilter_new(2,
                                Filter_new(&high_pass_filter_parameters),
                                Filter_new(&low_pass_filter_parameters));

                Poxi_DSP.ired = filterIred;
                rc = filterIred != NULL;
                if (rc)
                {
                        filterIred->setSampleTime(filterIred, DSP_SAMPLE_TIME);
                        filterIred->setupStatistics(filterIred,
                                        FUNDAMENTAL_FREC | MAX_OUTPUT | MAX_INPUT | MIN_INPUT);
                }
        }

        return rc;
}
```

2.2.7.2 Application execution

The execution of the POXI application is made basically by sending the processed signal to the GUI in TFT display and the Java application. The processing of the signal is performed in the interruption handler, and the resulting signals and their statistics are employed to refresh the info in the user interface.

The following code is self-explanatory and perform the main application loop that is interrupted when turning off the switch number 3 (last switch in ZYBO board), by turning on the first switch the information is sent to the Java GUI.

```
static uint8_t Poxi_run(void)
{
        Poxi_UI.framePanel->draw((Widget *)Poxi_UI.framePanel);

        Poxi_interruptDutyEnable = 1;
        do
        {
                Poxi_refreshUI();
                if (Poxi_ZYBO->switch_(0))
                {
                        Poxi_JavaGUI_report();
                }
        } while (Poxi_ZYBO->switch_(3));
        Poxi_interruptDutyEnable = 0;

        Poxi_clip->light(LIGHTPROBE_OFF);
        Poxi_ZYBO->leds(0);

        Poxi_UI.statusLabel->set_text(Poxi_UI.statusLabel,"* STOP *");

        return 0;
}
```

2.2.7.3 Application disposal

The disposal function member performs all the shutdown processes of the POXI application. Below it is listed the disposal tasks.

- Stops timer, disable interrupt handling and cache memories.
- Turns off finger clip LEDs.
- Turns off ZYBO LEDs
- Shuts down ADC, DAC, and PGA devices
- Releases composed filters dynamic memory.
- Releases UI components dynamic memory.

The code is exhibited below.

```
static void Poxi_dispose(void)
{
    Poxi_stopInterruptHandlers();

    cleanup_platform();

    Poxi_clip->light(LIGHTPROBE_OFF);
    Poxi_ZYBO->leds(0);

    Poxi_ADC->set_power_mode(MODE1);
    Poxi_DAC->power_mode(POWER_DOWN_THREE_STATE, 0xF);
    Poxi_PGA->shutdown();

    if (Poxi_DSP.red != NULL)
        Poxi_DSP.red->delete(&Poxi_DSP.red);

    if (Poxi_DSP.ired != NULL)
        Poxi_DSP.ired->delete(&Poxi_DSP.ired);

    if (Poxi_UI.framePanel != NULL)
        Poxi_UI.framePanel->delete((Widget **) &Poxi_UI.framePanel);
}
```

2.2.7.4 Interruption handling and data acquisition

The data acquisition is accomplished in the interruption handling call-back function. The interruption handling function has an internal state machine of four cyclic states, the interruption takes place four times in a millisecond, any internal state take place every millisecond which is the sample time for the DSP design.

In the next lines of code, it is exhibited the interruption handling function. It has four states, in these states the samples are taken and processed by the discrete filter, the resulting signal is stored in the corresponding buffer.

```
static void Poxi_timerInterruptHandler(void * data)
{
    static double primitiveSignal;
    static double filteredSignal;
    static uint32_t adjustCount = 0;

    XScuTimer_ClearInterruptStatus((XScuTimer *) data);

    if (Poxi_interruptDutyEnable)
    switch (Poxi_interruptState)
    {
    case REDLIGHT_ON:
        Poxi_clip->light(LIGHTPROBE_RED);
        Poxi_interruptState = SAMPLERED_OFF;
        break;

    case SAMPLERED_OFF:
        primitiveSignal = Poxi_ADC->read_analog();
        Poxi_clip->light(LIGHTPROBE_OFF);

        filteredSignal = Poxi_DSP.red->process(Poxi_DSP.red, primitiveSignal);

        Poxi_sampleBuffer[Poxi_sampleSection][RED_LIGHT][PRIMITIVE_SIGNAL][Poxi_sampleCount] = primitiveSignal;
        Poxi_sampleBuffer[Poxi_sampleSection][RED_LIGHT][FILTERED_SIGNAL][Poxi_sampleCount] = filteredSignal;

        Poxi_interruptState = IREDLIGHT_ON;
        break;

    case IREDLIGHT_ON:
        Poxi_clip->light(LIGHTPROBE_INFRARED);
        Poxi_interruptState = SAMPLEIRED_OFF;
        break;

    case SAMPLEIRED_OFF:
        primitiveSignal = Poxi_ADC->read_analog();
```

```
        Poxi_clip->light(LIGHTPROBE_OFF);

        filteredSignal = Poxi_DSP.ired->process(Poxi_DSP.ired, primitiveSignal);

        Poxi_sampleBuffer[Poxi_sampleSection][IRED_LIGHT][PRIMITIVE_SIGNAL][Poxi_sampleCount] = primitiveSignal;
        Poxi_sampleBuffer[Poxi_sampleSection][IRED_LIGHT][FILTERED_SIGNAL][Poxi_sampleCount] = filteredSignal;

        if (Poxi_maxSampleQty <= ++Poxi_sampleCount)
        {
            Poxi_sampleCount = 0;
            if (Poxi_sampleSection == SECTION_A)
            {
                Poxi_sampleStatus = READY_SECTION_A;
                Poxi_sampleSection = SECTION_B;
            }
            else
            {
                Poxi_sampleStatus = READY_SECTION_B;
                Poxi_sampleSection = SECTION_A;
            }
        }

        if (AUTOADJUST_TIME < ++adjustCount)
        {
            Poxi_adjust();
            adjustCount = 0;
        }

        Poxi_interruptState = REDLIGHT_ON;
        break;

    default:;
    }
}
```

2.2.8 Java GUI Application

The Java desktop application serves merely as a graphic user interface, it receives commands from the POXI device trough the serial port, these commands control the data displayed in the GUI traces in the Java app. Some of the details are explained in previous paragraphs.

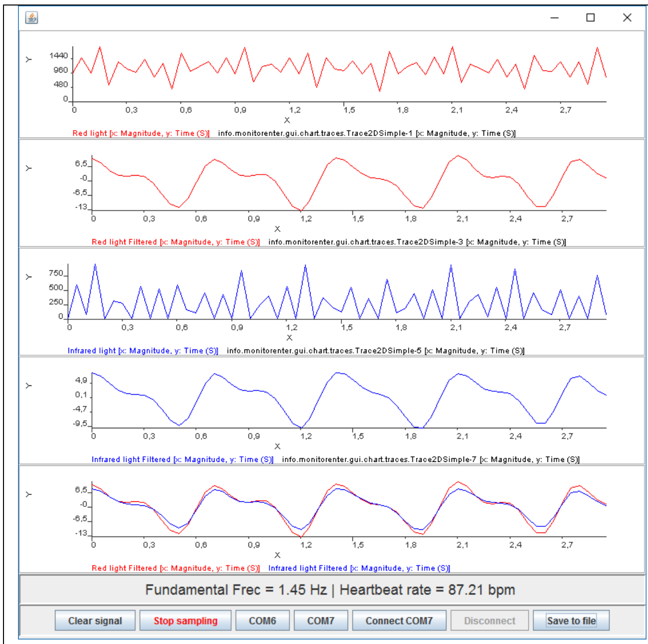


Figure 42. Java GUI Application

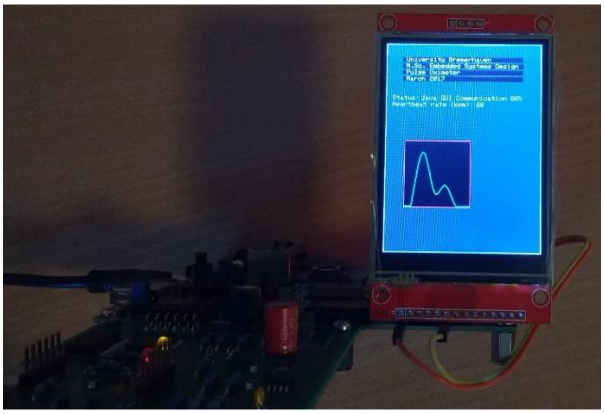


Figure 43. POXI TFT Display

The Java GUI application has very few option which are very self-explanatory, it can be cleared the current signal, start or stop the sampling from the POXI device, connect to any available port, and save to a file the current trace.

2.2.8.1 Command handler

The mechanism of the command handler is the most important section of the Java application, it must receive, process and recover the original command and perform its particular job.

The following diagram displays the mechanism to recover command message.

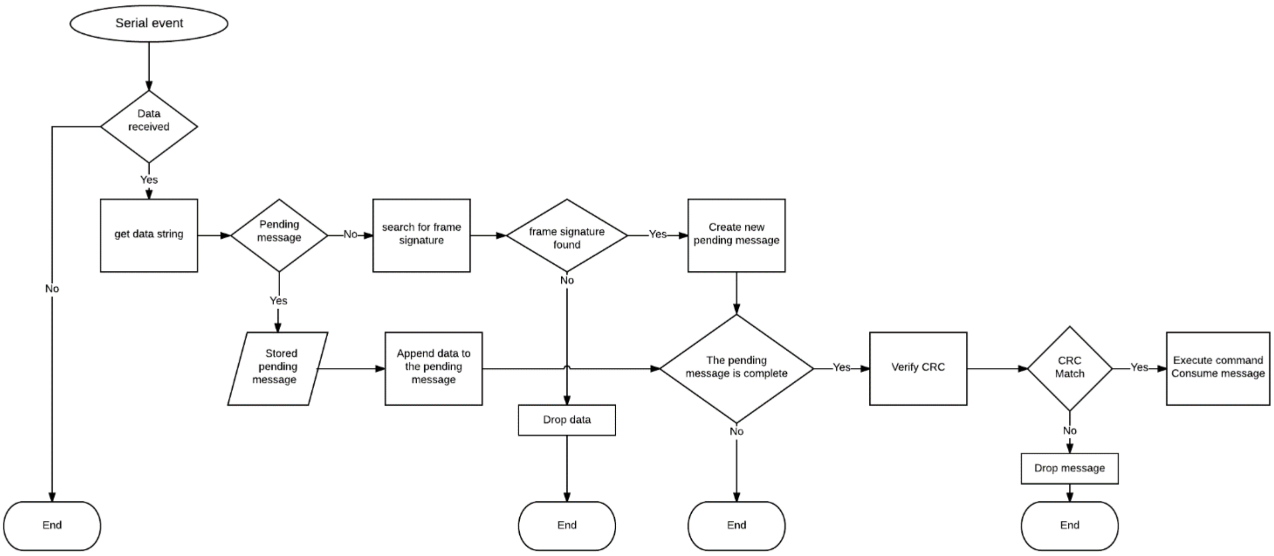


Figure 44. Command recovery process

2.2.8.1.1 Clear

The clear command is in charged to remove all the points from the given trace. If the trace parameter is 0xFF all traces are cleared.

0x5A	SIZE	CMD_CLEAR	TRACE	CRC

Figure 45. Frame command - Clear

```
    case CMD_CLEAR:
        System.out.printf("SClear %d\n", serial_buffer[3]);
        if (plot_flag) clearChart(serial_buffer[3]);
        break;
```

2.2.8.1.2 Plot

The plot command receives an array of double format numbers (64 bits) every double number is represented in 8 bytes, from POXI device it can be sent a command frame containing from 1 double number (8 bytes) to 16 double numbers (128 bytes). Individually the bytes are arranged and shifted in a 64 bits register and then converted into a double number by a function member of the class Double (Double.longBitsToDouble). The trace parameter defines the trace to plot the points.

0x5A	SIZE	CMD_PLOT	TRACE	double array[]	CRC

Figure 46. Frame command - Plot

```
case CMD_PLOT:
        System.out.printf("SPlot %d\n", serial_buffer[3]);
        if (plot_flag)
        if (13 <= serial_buffer.length)
        {
            int k = 0;
            long data_long = 0;
            int array_length = serial_buffer[4];

            if (array_length <= serial_DBSize)
            while(k < array_length)
            {
                data_long = 0;
                for (int j = 0; j < 8; j++)
                {
                    data_long |= (long)((0xFF&((long)serial_buffer[(5+j) + 8*k])) << (8*j));
                }
                value = Double.longBitsToDouble(data_long);

                if (selTrace == 0xFF)
                {
                    for (int t = 0; t < (numOfCharts * numOfTracesPerChart); t ++)
                    {
```

```
                        trace[t].addPoint(serial_TimeTrace[t], value);
                        serial_TimeTrace[t]+=serial_STimeTrace[t];
                    }
                }
                else
                {
                    trace[selTrace].addPoint(serial_TimeTrace[selTrace], value);
                    serial_TimeTrace[selTrace]+=serial_STimeTrace[selTrace];
                }
                k ++;
            }
        }
        break;
```

2.2.8.1.3 Set visible

The set visible command is intended to hide or make visible a particular trace. It has one boolean parameter in the command frame that can be either zero or different than zero. If the trace parameter is 0xFF, it will hide or make visible all the traces.

0x5A	SIZE	CMD_SET_VISIBLE	TRACE	Boolean	CRC

Figure 47. Frame command - Set visible

```
    case CMD_SET_VISIBLE:
        System.out.printf("SVisible %d = %d\n", serial_buffer[3], serial_buffer[4]);
        if (serial_buffer.length == 6)
        {
            if (selTrace == 0xFF)
            {
                for (int t = 0; t < (numOfCharts * numOfTracesPerChart); t ++)
                {
                    trace[t].setVisible(serial_buffer[4]>0);
                }
            }
            else
            {
                trace[selTrace].setVisible(serial_buffer[4]>0);
            }
        }
        break;
```

2.2.8.1.4 Set step time

The set step time command is intended to define the time interval or time delta between the trace points, in other words the magnitude of increment in horizontal axes. The time is represented in double precision (8 bytes). If the trace parameter is 0xFF, it will set the step time to all the traces.

0x5A	SIZE	CMD_SET_STEP_TIME	TRACE	double	CRC

Figure 48. Frame command - Set step time

```
    case CMD_SET_STEP_TIME:
        System.out.printf("SSTime %d", serial_buffer[3]);
        if (serial_buffer.length == 13)
        {
            long data_long = 0;
            for (int j = 0; j < 8; j++)
            {
                data_long  |= (long)((0xFF&((long)serial_buffer[4+j])) << (8*j));
            }
            value = Double.longBitsToDouble(data_long);

            if (selTrace == 0xFF)
            {
                for (int t = 0; t < (numOfCharts * numOfTracesPerChart); t ++)
                {
                    serial_STimeTrace[t] = value;
                }
            }
            else
            {
                serial_STimeTrace[selTrace] = value;
            }

            System.out.printf(" %f\n",value);
        }
```

```
        break;
```

2.2.8.1.5 Set time

The set time command is intended to establish the initial time of the trace points, in other words the initial place in horizontal axes. The time is represented in double precision (8 bytes). If the trace parameter is 0xFF, it will set the time to all the traces.

0x5A	SIZE	CMD_SET_TIME	TRACE	double	CRC

Figure 49. Frame command - Set time

```
    case CMD_SET_TIME:
        System.out.printf("STime %d", serial_buffer[3]);
        if (serial_buffer.length == 13)
        {
            long data_long = 0;
            for (int j = 0; j < 8; j++)
            {
                data_long |= (long)((0xFF&((long)serial_buffer[4+j])) << (8*j));
            }
            value = Double.longBitsToDouble(data_long);

            if (selTrace == 0xFF)
            {
                for (int t = 0; t < (numOfCharts * numOfTracesPerChart); t ++)
                {
                    serial_TimeTrace[t] = value;
                }
            }
            else
            {
                serial_TimeTrace[selTrace] = value;
            }

            System.out.printf(" %f\n",value);
        }
        break;
```

2.2.8.1.6 Text message

The text message command is intended to transmit only text, it has an ID parameter that can be used to indicate a particular target of the text (label or text box), The Java GUI application has only one text label for displaying this text, therefore the ID parameter is disregarded in this implementation.

0x5A	SIZE	CMD_TEXT_MSG	ID	String length	char array[]	CRC

Figure 50. Frame command - Text message

```
    case CMD_TEXT_MSG:
        System.out.printf("STextMsg id = %d", serial_buffer[3]);
        if (serial_buffer[4] < serial_buffer.length)
        {
            char [] msg = new char [serial_buffer[4]];

            for (int i = 0; i < serial_buffer[4]; i ++)
                msg[i] =(char) serial_buffer[5+i];

            infoLabel.setText(String.copyValueOf(msg));

            System.out.printf("Msg = %s\n", msg);
        }
        break;
```

3 CONCLUSIONS

This work covered the development of a configurable enhanced SPI hardware, software libraries for DSP, device drivers for ADC, DAC, PGA, and TFT colour display, graphics, UI widget components, and a command communication link layer.

The enhance SPI hardware was used to establish communication with the peripheral devices: ADC, DAC, PGA and TFT colour display. All these devices have different SPI setup for clock polarity, clock phase, data length and even settle time for the TFT display to achieve maximum communication speed.

The DSP software library was developed in the manner to have an instance of a light or composed filter, the composed filter features multi filtering and input output signal statistics. The filter design is made in MATLAB script for modelling step and frequency responses and simulation of the legacy code blocks of the DSP in Simulink.

The device drivers for ADC, DAC, PGA, and ZYBO make bottom software layer easy and intuitive to use, these libraries encapsulate the dirty task of low level hardware handling and configuration. This layer presents a high level software interface of the hardware.

The graphics library contains the low level driver code for the TFT display, and the middleware library for drawing pixels, lines, rectangles, circles, and text. This layer serves as the foundation for the UI widget components.

The UI widget components library is a high level software implementation, it presents a design for a graphical user interface library. At the moment, this library has three classes: 2D plot, Text label and Frame panel. The 2D plot displays the graphic trace of given integer values, the Text label gives capabilities for resizing, moving, and refreshing graphic text, and the Frame panel holds up all the widgets in a tree data structure, transmit the draw and refresh interface messages to the entire tree, and shows border and back ground colours.

The command communication link layer library is intended to have a reliable communication, it offers a data verification method, For now, this layer has commands to transmit double numbers for time and magnitude, text, and flags. This library is extensible to include device ID in the command arguments, and establish communication in any network topology. The foundation layer of this library based on UART protocol, nonetheless it can be exchanged to any other communication protocol.

The dynamic memory allocation approach worked in a safe manner, there were no crash neither leakage of memory. However, for some critical applications with software compliance, it would be needed to implement a memory management unit having a pool taken from the bottom of the memory stack during software compilation.

The Java GUI application is intended to display data by the execution of commands sent from the POXI device, it serves merely as a display driven by the command communication link layer.

The pulse oximeter application is accomplished by the usage of hardware, device drivers, digital filters, graphical user interface and serial communication link-layer offered by the implemented hardware software base libraries.

References

Basic understanding of the pulse oximeter. (n.d.). https://windward.hawaii.edu/facstaff/miliefsky-m/ZOOL%20142L/aboutPulseOximetry.pdf.

Beer-Lambert Law. (n.d.). *http://life.nthu.edu.tw/~labcjw/BioPhyChem/Spectroscopy/beerslaw.htm*.

Digilent. (2016). *ZYBO™ FPGA Board Reference Manual.*

History of Pulse Oximetry. (n.d.). *https://www.amperordirect.com/pc/help-pulse-oximeter/z-pulse-oximeter-history.html*.

ILITEK. (2016). *a-Si TFT LCD Single Chip Driver, 240RGBx320 Resolution and 262K color.* http://www.ilitek.com.