## Project Of Systems-on-Chip (CS-309)

# Lab 2: Thermal Camera Interface

ESER Can
MAROUF Imad Eddine

Supervisor : Pr. BEUCHAT René
*École Polytechnique Fédéral de Lausanne, Switzerland*

***Abstract*—In this report, we present the details and steps used to design a thermal camera interface. The goal of this laboratory is to learn mechanics used in the camera acquisation systems in order to get a frame from thermal camera and store it in an image file. As well, during the second part of the laboratory we learned about the system integration and how to assemble all interface components together. We described how we completed the design of the interface, a simulation of it and the system integration process. This report is in the context of "Projet de Systems-on-Chip CS-309" course supervised by Pr. BEUCHAT René.**

### INTRODUCTION

The objective of this second laboratory is to complete the design of thermal camera interface to save frame on a host computer from Lipton IR camera sensor by coding the computation module and communication between different modules within the interface. In order to do this, we had to interconnect many modules with in the interface going from acquisition by Flir lepton camera into saving it as pgm format. More precisely, we designed the module to do the statistics computation and level adjustments.

Camera we used is FLIR lepton which captures infrared radiation input in its nominal response wavelength band (from 8 to 14 microns) and outputs a uniform thermal image. It has 9 Hz refresh cycle, it belongs to serial-data cameras as it uses I2C for configuration and SPI for data acquisition. This thermal camera has a master clock frequency of 25 MHz but the export compliant frame rate is less than 9 Hz due to military restrictions to avoid terrorist attacks. For exemple, it can be used to attack planes by rockets. With 9 Hz of refreshing cycle, it will be difficult to detect the plane.

Moreover, we realised the importance of infra-red camera during this difficult situation when the entire world is affected by the coronavirus. IR camera are widely used in airports and transport stations to detect affected people by using them with color map in order to measure high temperatures and fevers.

Figure 1: Flir Lepton Thermal Camera

### I. MASTER COMPONENTS AND AVALON BUS

The NIOS processor can communicate with surrounding components via an Avalon bus which is an interface that indicates port connections between the master and slave components. Many bus designs exist in the industry but our FPGA of Intel use the bus called the Avalon bus. The operation of the Avalon bus is quite simple. The Avalon bus uses memory-mapped I/O to ensure the link between the

master and the slave, so the same address bus is used to access memory and I/O devices. It means that the master can look for the slave at specific addresses in the master's address space. The diagram below allows us to understand our full system. It's composed of a NIOS II Processor which is the master, SRAM, an Avalon Bus and our three programmable interfaces: MCP3204 used to convert the analog signal coming from the joystick to digital signal. This interface contains also the Serial Peripheral Interface, Pulse Width Modulation Generator allows to generate two PWM signals depending on the ADC value in order to control both servos, becayse Flir camera is attached on pantilt controlled by two servos to move it right-left, and buttom-up. Lepton Controller used to take a picture with the thermal camera and store it in host pc.

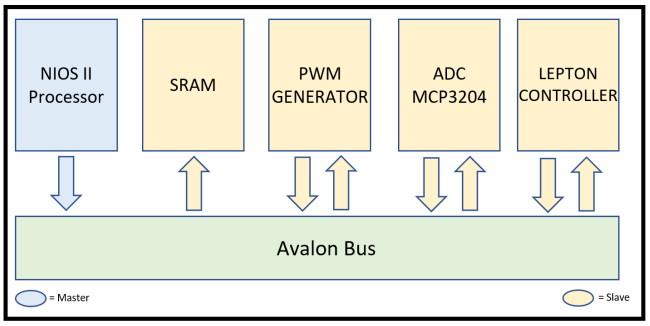


Figure 2: Diagram of Full System

## II. Camera Acquisition System Design

Thermal camera interface is composed of many modules connected together in order to capture and save the image on the computer, described below:

- *SPI Controller*: is needed to read data serially and forwards to Lepton Controller.
- *Lepton Controller* : is used to filter out the discard packets of data and reconstruct 14-bit pixel values from the incoming data stream.
- *RAM* : is 8192*16-bit RAM used to store the frame coming from the camera, in order to apply interpolation on the frame and send it to Avalon-MM slave interface.
- *Statistics computation* : is used to compute the minimum, maximum and average pixel values in an image. The computation of average requires a resource-heavy hardware divider which is better to avoid. Instead, we compute the sum of all pixel values and leave the division to be done by the host processor which most probably has a very efficient hardware divider.
- *Level adjustements* : standard scene images taken with a thermal camera often produce very dark images due to the low temperature difference among the various entities in the scene. This component is used to interpolate the frame's pixel intensities to obtain a much more visible image.

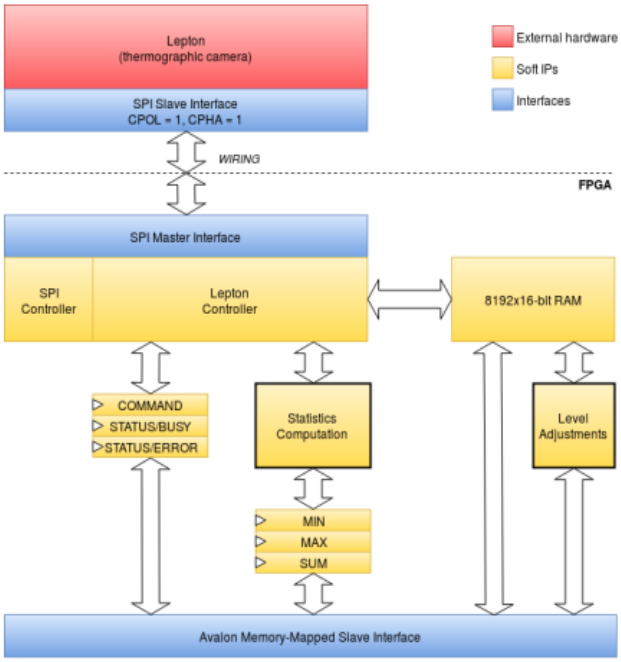


Figure 3: Camera Acquisition System.

## III. Design of the Architecture

### A. Statistics Computation

Thermal cameras are able to capture scences within a temperature range therefore the values of pixels is based on temperature variation within the image captured. To make, an image visible we have to interpolate values within the frame captured. A illustrated in Figure 3, it has a statistics computation module which is responsible for calculating minimim, maximum and average pixel values in a frame in O(1) time by the hardware since all pixels are flowing through.We had complete Lepton_stat.vhd file to the computation, based on the timing diagram of figure 4, as pixels of the frame start flowing each rising edge of system clock, which is a vector of 14 bits when we get the start signal named "pix_sof". We have also to check in each reading the pix_data signal that it has active logic on pix_data signal. simultaneously we are calcuting the minimum, maximum and average signals. We notice end of the frame by high logic of pix_end signal, therefore we returns stat_valid signal as high as well the statistics signals (minimum, maximum, and average signals), the code is provided in Annexe.

### B. Level Adjustments

In order to see a visible thermal image, we have to the interpolation operation due to the fact that the acquisition process in based on variation within the frame as desribed above in the section I. As we know the division operation is tough process to be implemented in FPGA, but it cannot be avoided here, we used the module lpm_divier.vhd provided

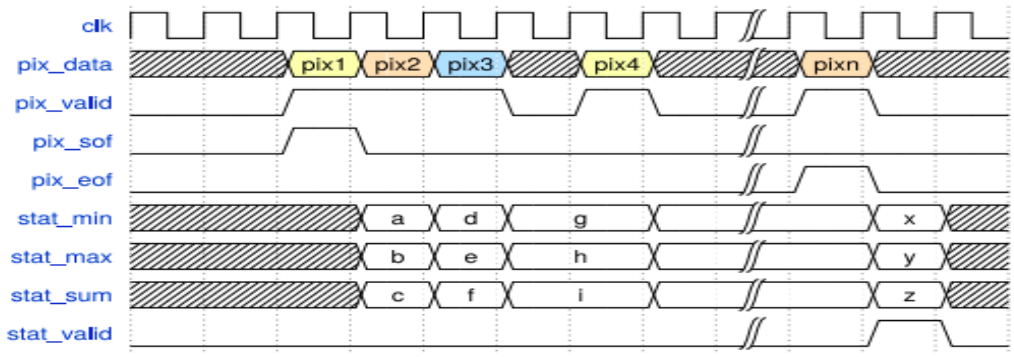


Figure 4: Lepton stats port timing diagram

in the system to do the interpolation operation. Taking into consideration as described in the documentation that the denominator is 14 bits and numerator 28 bits, so we had to resize the input signals.

$$adjusted\_pixel = (pixel - min)/(max - min)$$

### C. Software

After our programmable interface is designed, compiled and launched on our FPGA, we used NIOS II System Architect Design to program the NIOS II Processor with completing a C program.

To do this, we used a IO Header file lepton_regs.h. This header file, defines the core's registers map. So it allows us to use macros to access low-level hardware instead of hard-coding various numbers to make the code much easier to read and to debug when necessary. We should also add system.h to access to base addresses of global architecture as defined by Qsys and io.h in order to read and write to system peripherals which allow us to use simple instruction in our code.

The table below shows the unit's register map. The unit's registers are read-only and write-only. Each register is 16 bits wide, and there are 2 views over the data captured (RAW_BUFFER & ADJUDSTED_BUFFER).

| Byte OFFSET | Register | Name | R/W |
|---|---|---|---|
| 0x0000 – 0x0001 | 0 | COMMAND | WO |
| 0x0002 – 0x0003 | 1 | STATUS | RO |
| 0x0004 – 0x0005 | 2 | MIN | RO |
| 0x0006 – 0x0007 | 3 | MAX | RO |
| 0x0008 – 0x0009 | 4 | SUM_LSB | RO |
| 0x000A – 0x000B | 5 | SUM_MSB | RO |
| 0x000C – 0x000D | 6 | ROW_IDX | RO |
| 0x000E – 0x000F | 7 | RESERVED | - |
| 0x0010 – 0x258F | 8 – 4807 | RAW_BUFFER | RO |
| 0x2590 – 0x3FFF | 4808 – 8191 | RESERVED | - |
| 0x4000 – 0x657F | 8192 – 12991 | ADJUSTED_BUFFER | RO |
| 0x6580 – 0x7FFF | 12992 – 16383 | RESERVED | - |

Table I: Lepton Controller register map

- First, we had to allow NIOS II software to be able to write an image file on the host computer by following the steps as described in the lab manual.

- Secondly, we had to complete lepton_start_capture() function, which in charge of initiating the start of capturing frame, by writing 1 to the start bit (bit 0) of the COMMAND register (0x0004). We used IOWR_16DIRECT(BASE, OFFSET, DATA): write a 16-bit of the value DATA at the location with address BASE+OFFSET.

```
void lepton_start_capture(lepton_dev *dev) {
 IOWR_16DIRECT(dev->base, LEPTON_REGS_COMMAND_OFST
     , LEPTON_COMMAND_START);
}
```

- Thirdly, we had to complete lepton_error_check() which checks if the ERROR bit (bit 1) in STATUS register is equal to '1', then we restart the capturing process.

```
bool lepton_error_check(lepton_dev *dev) {
    uint16_t reg_status = IORD_16DIRECT(dev->base,
     LEPTON_REGS_STATUS_OFST);
    uint16_t error_flag = reg_status &
    LEPTON_STATUS_ERROR_MASK;
    return error_flag != 0;
}
```

- Finally, we had to complete lepton_wait_until_eof() which is in charge of detecting the end of frame capturing. If the BUSY bit (bit 0) in STATUS register (0x0002) is equal to '0'.

```
void lepton_wait_until_eof(lepton_dev *dev) {
  uint16_t reg_status = 0;
  uint16_t capture_flag = 0;
    do {reg_status = IORD_16DIRECT(dev->base,
    LEPTON_REGS_STATUS_OFST);
        capture_flag = reg_status &
    LEPTON_STATUS_CAPTURE_IN_PROGRESS_MASK;
    }while (capture_flag != 0);
}
```

All functions described above were initiated in the main "app.c" file to save the captured frame as "output.pgm" in the host computer and print a message on the screen to see the if the process was successful. Complete version of the code is provided Annexe.

## IV. Full System Integration

During the second part of the laboratory, we were asked to assemble all the components of the system together, because during previous laboratories our task was just to edit different components of the provided system. In order to assemble different modules of our interface, we have to use a system intergration tool which is Qsys included in Quartus Software package provided by Intel.

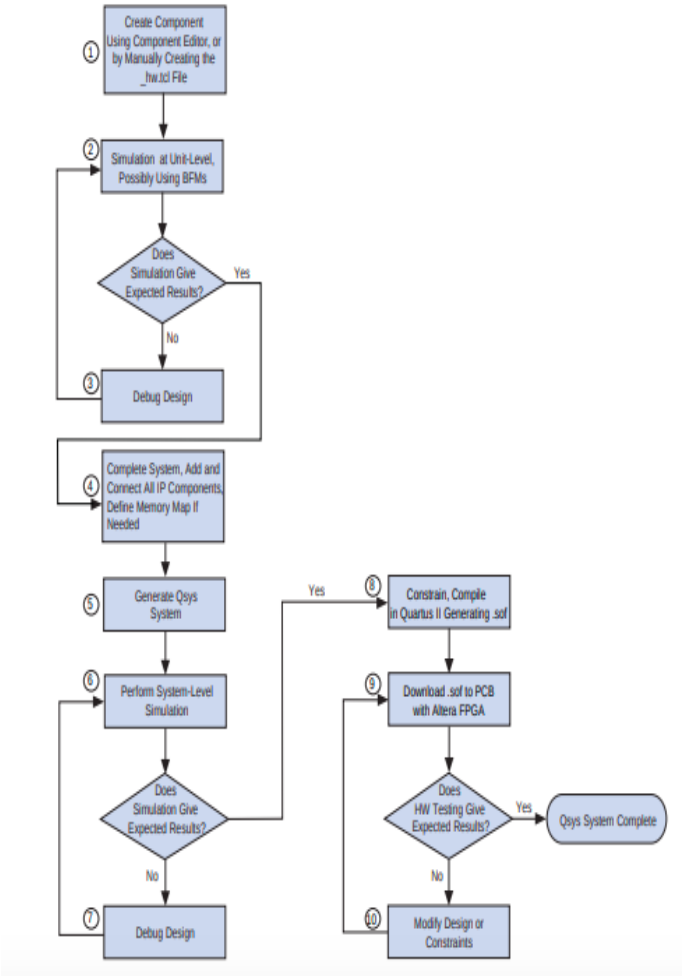


Figure 5: QSys Design Flow

Qsys automatically creates interconnect logic from the high-level connectivity you specify, thereby eliminating the error-prone and time-consuming task of writing HDL to specify system-level connections due to the fact that in many projects components does not have the same interface which makes the connection between them difficult, integration tools makes that possible by insering adapters to convert between the different bus sizes between components.

According to figure 6, showing our full Qsys system composed of different components :

- *clk_0*: which is the master clock component.
- *nios2_gen2_0* : is the new generation of nios2 processor component. This component is composed of data_master used for reading and writing in the memory or peripherals and instruction_master which is looking for interruption in the memory.
- *jtag_uart* : is needed for serial communication with our system.
- *pwm* : is our pwm component used to generate two PWM signals depending on the ADC value in order to control both servos.
- *mcp3204* : is our component used to convert the analog signal coming from the joystick to digital signal.
- *lepton* : is our component used to take a picture with the FLIR Lepton thermal camera and store it in host PC.

## V. Results

Once our programmable interface is designed, we simulated it on Modelsim to check its operation before programming on NIOS and running it on the FPGA. As illustrated in Figure 7, we see that the full design functions correctly, where we only want to write data by setting Write, SLCK, Csn and MISO to high level.

After validating the simulation of our programmable interface Lepton Controller, others programmable interfaces were tested and verified during previous labs, we obtain the picture on Figure 8 which presents a cup of tea took with the thermal camera. The frame is saved in host PC in PGM format. A simple online converter allows to obtain our picture in PNG format. Here there isn't information about color because our system was just designed to receive the intensity for every pixel of infrared energy and store them in a table of value without a computation of which intensity level is assigned to which color in RGB format.

Figure 8: Image obtained with Thermal Camera

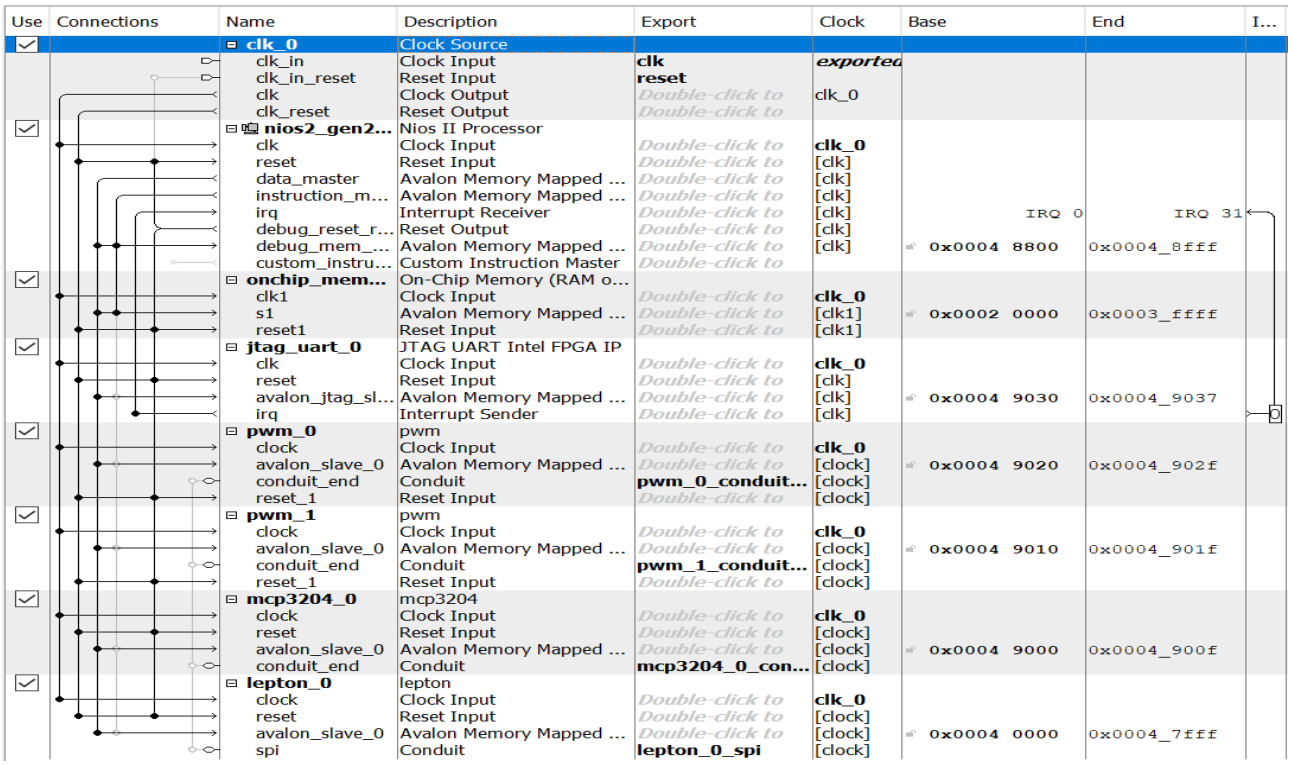


Figure 6: Full Qsys System

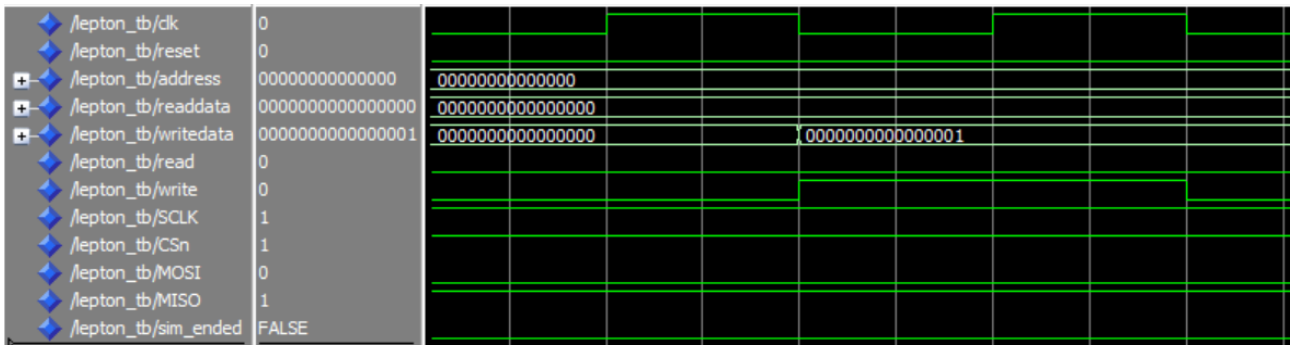


Figure 7: Lepton Controller Simulation

## Conclusion

The Lab 2 of ”Projet de Systems-on-Chip CS-309” allowed us to learn about the mechanics involved in camera acquisition systems, and complete the design of the thermal interface to capture frames and save them in host computer. Moreover, in the second part we learnt about the process of integrating different components of our system using Qsys software to have more rebust, easy to improve, and easy to debug systems in the future.

Additionally, the crucial role of the documentation and design tools manuals to learn more about the tools which can make the production process more easy and very effective using RTFM technique to be a competent engineer.

## References

**Lab Handouts :**

[1] R. Beuchat, Thermal Camera Interface.

[2] R. Beuchat, Full System Integration.

**Theoretical course :**

[3] R. Beuchat, Camera Infra Red.

**Datasheet :**

[4] FLIR, FLIR LEPTON LWIR Datasheet.

## ANNEXE : VHDL AND C CODES

```vhdl
-- Lepton_stats.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lepton_stats is
    port(
        clk : in std_logic;
        reset : in std_logic;
        pix_data : in std_logic_vector(13 downto 0);
        pix_valid : in std_logic;
        pix_sof : in std_logic;
        pix_eof : in std_logic;
        stat_min : out std_logic_vector(13 downto 0);
        stat_max : out std_logic_vector(13 downto 0);
        stat_sum : out std_logic_vector(26 downto 0);
        stat_valid : out std_logic);
end lepton_stats;

architecture rtl of lepton_stats is

signal min : unsigned(13 downto 0) <= "11111111111111";
signal max : unsigned(13 downto 0) <= "00000000000000";
signal sum : unsigned(13 downto 0) <= "00000000000000";
signal next_sum : unsigned(26 downto 0);
signal valid: std_logic;
signal u_pix_data: unsigned(13 downto 0);
signal u_resized_pix_data: unsigned(stat_sum'range);

begin
 u_pix_data <= unsigned(pix_data);
 u_resized_pix_data <= resize(u_pix_data, sum'length);
 next_sum <= u_resized_pix_data + sum;

 stats : process(clk, reset)
   begin
       if reset ='1' then
            min <= (others => '1');
            max <= (others => '0');
            sum <= (others => '0');

       elsif rising_edge(clk) then
         if (pix_sof ='1' and pix_valid = '1') then
          min <= u_pix_data;
          max <= u_pix_data;
          sum <= u_resized_pix_data;
          elsif pix_valid ='1' then
                if min >= u_pix_data then
                    min <= u_pix_data;
                end if;
                if max < u_pix_data then
                    max <= u_pix_data;
                end if;
                    sum  <= next_sum;
       elsif (pix_eof = '1' and pix_valid = '1') then
            valid <= '1';
          end if;
       end if;
    end process stats;
    stat_min <= std_logic_vector(min);
    stat_max <= std_logic_vector(max);
    stat_sum <= std_logic_vector(next_sum);
    stat_valid <= valid;
end rtl;
```

```vhdl
-- level_adjuster.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity level_adjuster is
    port(
        clk             : in  std_logic;
        raw_pixel       : in  std_logic_vector(13 downto 0);
        raw_max         : in  std_logic_vector(13 downto 0);
        raw_min         : in  std_logic_vector(13 downto 0);
        raw_sum         : in  std_logic_vector(26 downto 0);
        adjusted_pixel : out std_logic_vector(13 downto 0));
end level_adjuster;

architecture rtl of level_adjuster is
    component lpm_divider
        port(
            clock    : in  std_logic;
            denom    : in  std_logic_vector(13 downto 0);
            numer    : in  std_logic_vector(27 downto 0);
            quotient : out std_logic_vector(27 downto 0);
            remain   : out std_logic_vector(13 downto 0));
    end component;

    signal denom :std_logic_vector(13 downto 0);
    signal numer :std_logic_vector(27 downto 0);
    signal quotient :std_logic_vector(27 downto 0);
    signal remain   : std_logic_vector(13 downto 0);
    -- output = ( pixel - min ) / (max - min )
    begin
        divider : lpm_divider
            port map(
            clock    => clk,
            denom    => denom,
            numer    => numer,
            quotient => quotient,
            remain   => remain); --unused
        -- Why there is raw_sum input signal called sum although we do not need it in this file ?
        numer <= std_logic_vector((unsigned(raw_pixel) - unsigned(raw_min)) * resize(X"3fff", raw_pixel'length));
        denom <= std_logic_vector(unsigned(raw_max) - unsigned(raw_min));
        adjusted_pixel <= quotient(13 downto 0);
end rtl;
```

```
-- app.c

#include <io.h>
#include <stdbool.h>
#include <stdlib.h>
#include <stdio.h>

#include "lepton/lepton.h"
#include "system.h"

int main(void) {
    // Hardware control structures
    lepton_dev lepton = lepton_inst((void *)
     LEPTON_0_BASE);

    // Initialize hardware
    lepton_init(&lepton);

    bool error = false;
    do {
      lepton_start_capture(&lepton);
      lepton_wait_until_eof(&lepton);
      error = lepton_error_check(&lepton);
    }while (error);

    // Save the adjusted (rescaled) buffer
    to a file.
    print("Done Capturing, Saving Image in
    '/mnt/host/output.pgm' \n");
    lepton_save_capture(&lepton, true, "/mnt
    /host/output.pgm");
    printf("Your image is written to host\n
    ");

    return EXIT_SUCCESS;
}
```