# Lab 2.0
# Thermal Camera Interface

## Lab 1.0 + 1.1 + 1.2 - Camera directional-stand (recap)

The goal of the lab 1 series was to use a PS2 joystick to control the movement of a pan-tilt module. To this end, you implemented the hardware and software interfaces needed to control a PWM generator and an ADC.

- The ADC was used to sample data from the analog PS2 joystick.
- The digital samples were used to control the duty cycle of 2 PWM signal generators and to therefore move the pan-tilt module.

We now have a system on which a camera can be mounted, so let's move on and shift our focus towards camera systems themselves.

## Lab 2.0 – Camera acquisition interface

The goal of this lab is to learn about the machinery involved in camera acquisition systems (i.e. how to get a frame from the camera sensor onto an image file stored on your computer).

### Theory

#### Background - Camera hardware interfaces

There are many different interfaces used among the various cameras available on the market, but they can all be separated into 2 categories:

- *Parallel*-data cameras can output a full pixel value on each clock cycle (subject to horizontal and vertical signaling). Interfacing with such cameras is costly as one needs to use a chip with at least as many free pins as the pixel depth of the sensor. Therefore, many systems may just not have enough pins available to interface with a high pixel depth camera.
- *Serial*-data cameras output a pixel bit-by-bit over multiple clock cycles (subject to horizontal and vertical signaling). Interfacing with such cameras is very affordable, as only 1 pin is required for communication (note though that you may need 2 pins depending on the electrical signaling used).



René Beuchat, Philémon Favrod, Sahand Kashani

In this lab, we will examine *serial*-data cameras. Such cameras can further be divided into 2 sub-categories:

- Cameras that use a *custom* serial communication protocol require specific controllers to be built to interface with them.
- Cameras that use a *standard* serial communication protocol ($I^2C$, SPI, UART …) are more flexible, as most microcontrollers provide such communication interfaces.

## Chosen camera - FLIR Lepton

We are going to use a *standard*-protocol *serial*-data camera. To make this even more interesting, we will use a *thermal* camera, namely the FLIR LEPTON, shown in Figure 1. Its characteristics are summarized in Table 1.

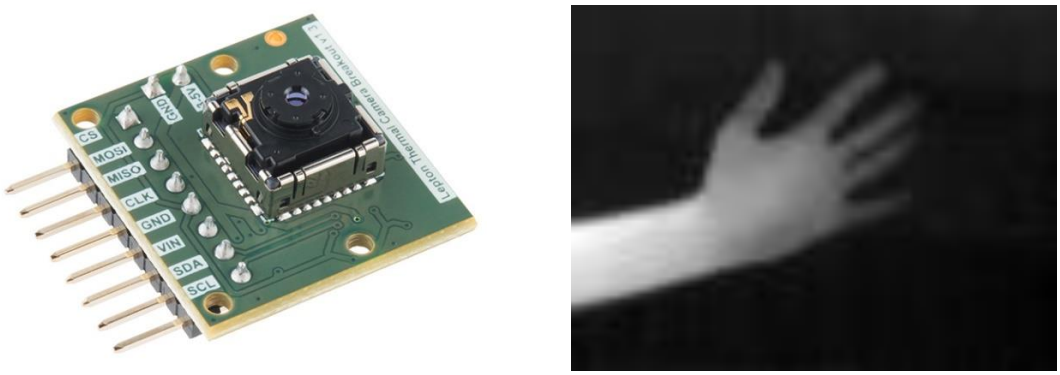


FIGURE 1. FLIR LEPTON

| | |
|---|---|
| **Array width** | 80 |
| **Array height** | 60 |
| **Effective frame rate** | 8.6 Hz |
| **Output format** | 14-bit |
| **Data interface** | Video over SPI (VoSPI) |
| **Control interface** | $I^2C$ |

TABLE 1. LEPTON SPECIFICATIONS

The lepton is easy to interface with, as it provides an SPI *data* interface and an $I^2C$ *control* interface. This makes it simple for most microcontrollers to use the device, as having an SPI and an $I^2C$ controller would be enough to communicate with it. However, since we are going to use an FPGA to interface the device, we will design the complete frame acquisition system ourselves in order to add some cool extra features (or else what would you be doing in this course? ☺).

## General note about thermal cameras

Thermal cameras are able to capture scenes with a wide temperature range. Therefore, if you take a photo of a standard scene with such a camera, you will obtain a *very dark image* with almost nothing visible. This is normal as there is not much temperature variation in standard scenes.

To make the temperature differences more visible, you need to interpolate the scene's pixel values to the minimum and maximum supported by the image format. As an example, Figure 1 contains an interpolated image in the range (0, 16383).

## Camera acquisition system design

Figure 2 shows the block diagram of our camera acquisition system. We will now discuss the various components involved.

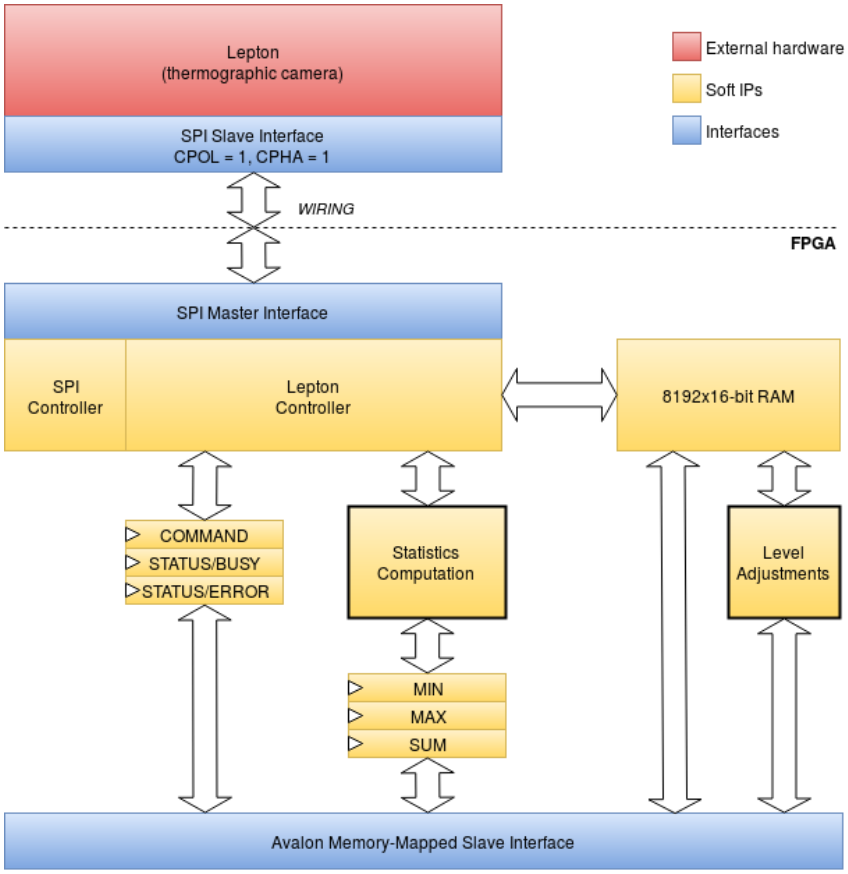


**FIGURE 2. LEPTON ACQUISITION SYSTEM**

### SPI controller

The lepton outputs its data over an SPI bus, so we need an SPI interface to capture the data. This component is a generic SPI controller that reads data serially and forwards 8-bit chunks to the LEPTON CONTROLLER.

### Lepton controller

The lepton outputs data in the form of two 160-byte VoSPI packets: *video* and *discard*. These packets are received in 8-bit chunks from the SPI CONTROLLER. The LEPTON CONTROLLER is in charge of 2 tasks: It filters out the *discard* packets, and reconstructs 14-bit pixel values from the incoming data stream.

### Statistics computation

This component is in charge of computing the *minimum*, *maximum* and *average* pixel values in an image, which are useful values for numerous image processing algorithms.

These values can be computed in *O(W x H)* time on a processor (by iterating over all pixel values in a frame), but can be computed in *O(1)* time by the hardware capturing the frame (all pixels are already flowing through it, so the hardware can update the values as it sees new pixels).

The *minimum* and *maximum* can easily be computed while data is flowing through the LEPTON CONTROLLER. The *average*, however, requires a resource-heavy hardware divider, which we would like to avoid in our design. Instead, we chose to 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 adjustments

As previously stated, 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. The LEVEL ADJUSTMENTS component is responsible for interpolating the frame's pixel intensities to obtain a much more visible image. It uses the *minimum* and *maximum* values computed by the STATISTICS COMPUTATION unit to this end.

### 8192x16-bit RAM

We chose to store the incoming frame in an internal on-chip memory instead of in system memory. This is done mainly in order to provide 2 alternative views of the frame at no extra storage and computation cost.

- Original RAW buffer view
- Adjusted (interpolated) buffer view

Note that the image is *not stored twice* in the RAM. The 2nd view is computed on-the-fly as data is being read from the RAM, and is then outputted on the Avalon-MM Slave interface (see Figure 1.)

Due to the small frame size of the lepton, the on-chip memory space is not too prohibitive. Pixels are 14 bits wide, so we store them in 16-bit memory words. Storing a full frame therefore requires 80 x 60 = 4800 words. Since memory sizes need to be powers of 2, we resort to using an 8192-word memory.

### Avalon-MM Slave Interface

Table 2 shows the register map of the lepton interface. Notice the 2 views over the data.

| Byte offset (from base) | Register Number | Name | Access |
|---|---|---|---|
| 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 2. AVALON-MM SLAVE REGISTER MAP

### *COMMAND register*

Table 3 shows the details of the COMMAND register.

| **Bit** | 31 .. 1 | 0 |
|---|---|---|
| **Name** | RESERVED | START |

**TABLE 3. COMMAND REGISTER**

Writing 1 to the START bit will instruct the unit to start capturing a new frame and resets the ERROR bit of the STATUS register.

### *STATUS register*

Table 4 shows the details of the STATUS register.

| **Bit** | 31 .. 2 | 1 | 0 |
|---|---|---|---|
| **Name** | RESERVED | ERROR | BUSY |

**TABLE 4. STATUS REGISTER**

- The BUSY bit reads as 1 when the unit is busy, and 0 when idle.
- The ERROR bit reads as 1 when an error is detected, and 0 if no error was detected.

### *MIN, MAX, SUM_LSB, SUM_MSB registers*

These registers contain the minimum, maximum, and sum over all pixels in the frame. Note that the sum requires 27 bits to be fully represented, so it is split into 2 16-bit registers.

### *ROW_IDX*

This register contains the number of the current line being captured ($1 \leq$ ROW_IDX $\leq 60$). It is useful for debugging purposes only.

### *RAW_BUFFER, ADJUSTED_BUFFER*

These address zones point to the RAW and to the adjusted frame address ranges. Remember that only the RAW frame is stored in the internal on-chip memory, and that the adjusted view is computed live when requested for each pixel.

## Practice

### Task 1 – HW – Statistics computation

Implement the STATISTICS COMPUTATION component in `hw/hdl/lepton/hdl/lepton_stats.vhd`.

Figure 3 shows the timing diagram that the component must satisfy. The `pix_sof` and `pix_eof` signals inform you about the start and the end of a frame. Remember to generate a pulse on the `stat_valid` signal when you have valid data in the `stat_min`, `stat_max`, and `stat_sum` registers.

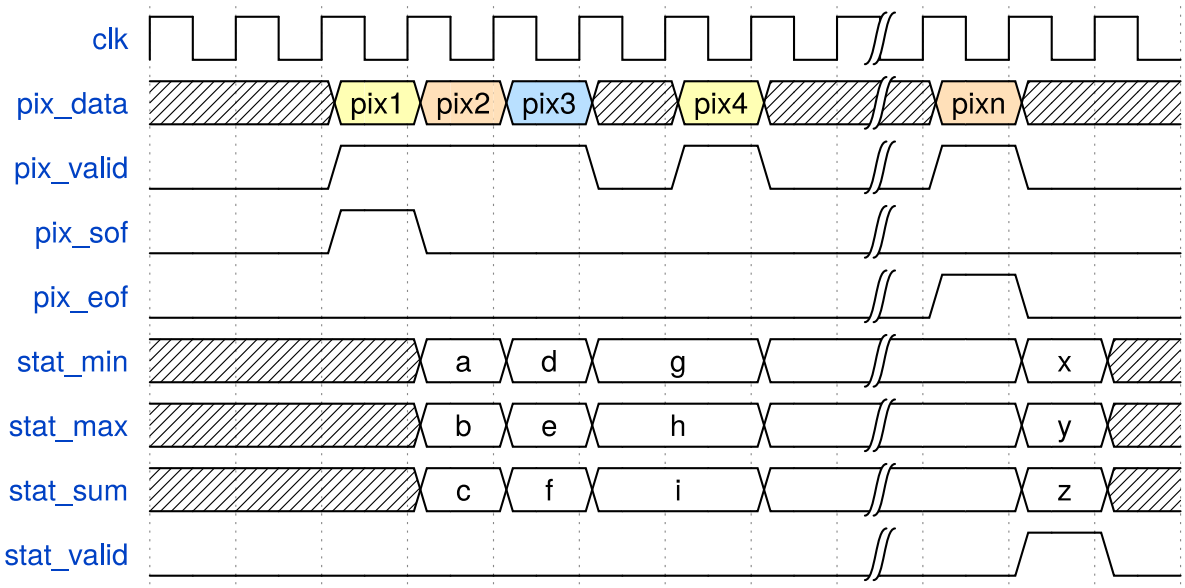


FIGURE 3. LEPTON_STATS PORT TIMING DIAGRAM

## Task 2 – HW – Level adjustments

Implement the LEVEL ADJUSTMENTS component in `hw/hdl/lepton/hdl/level_adjuster.vhd`.

We said earlier that hardware dividers are expensive and that they should be avoided when possible. We were able to avoid inserting one in the STATISTICS COMPUTATION component, but it cannot be avoided in LEVEL ADJUSTMENTS, as there is no way to interpolate the pixel values without one.

We provide you with the component declaration for one such divider in `lepton_stats.vhd`.

## Task 3 – SW – C code completions

### Nios II SBT project setup

We want the Nios II processor to be able to write a frame to an image file located on your *host* computer. To do this, we need to enable a specific software package in the *BSP Editor*. After creating your Nios II SBT project, follow the steps below:

1. Right-click on the BSP project > Nios II > BSP Editor …
2. In the *Software Packages* tab, enable the *altera_hostfs* package (Figure 4).
3. Save the configuration
4. Press the *Generate* button.
5. Close the dialog.

You can then continue with the standard software workflow you have used for the previous labs.

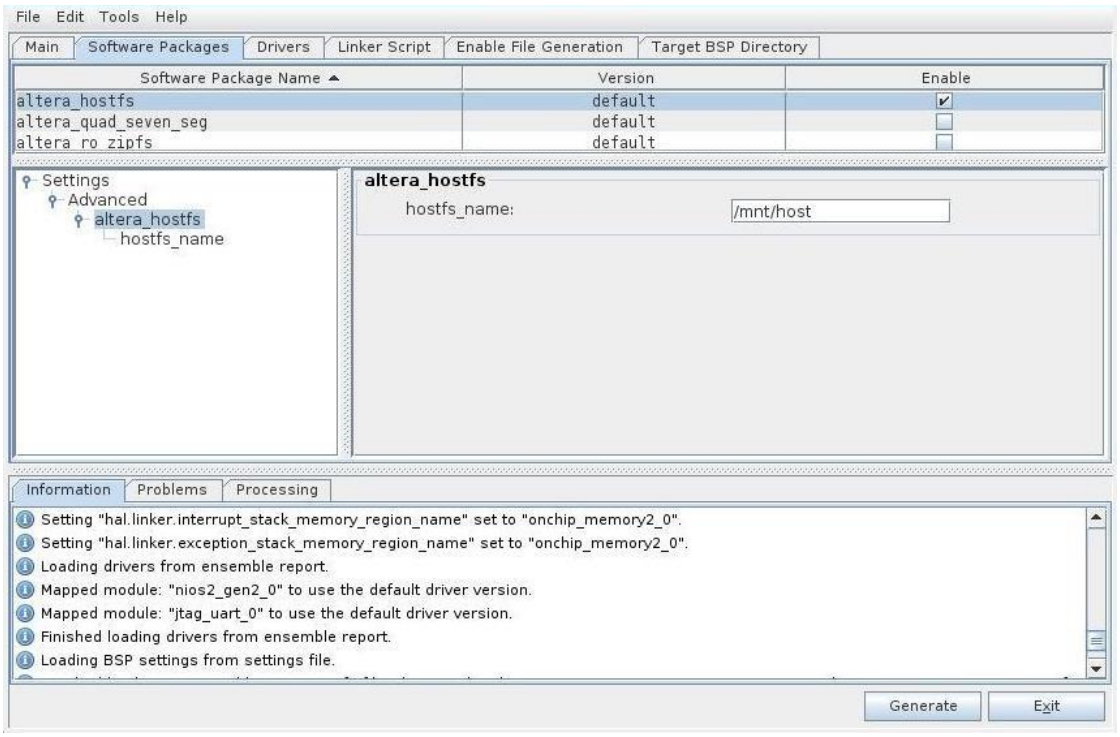


**FIGURE 4. ALTERA HOSTFS SOFTWARE PACKAGE**

## Lepton library

Implement the following 3 functions in `lepton.c`:

```
void lepton_start_capture(lepton_dev *dev);
bool lepton_error_check(lepton_dev *dev);
void lepton_wait_until_eof(lepton_dev *dev);
```

## Procedure for capturing a frame

Complete the `main(void)` function in `app.c` to capture and write a frame to a PGM image file (use the functions implemented in `lepton.c`).

Recall the procedure needed to capture a frame with this hardware design:

1. Write 1 to the START bit of the COMMAND register.
2. Wait until the BUSY bit of the STATUS register is 0.
3. If the ERROR bit of the STATUS register reads 1, then restart from point 1. Otherwise the frame is available for reading at the RAW_BUFFER and ADJUSTED_BUFFER offsets within the lepton's register map.

## Viewing the results

Once all code segments have been filled, and that you successfully execute the main function, you can find the resulting image file at `sw/nios/application/output.pgm`.

*Enjoy the thermal camera ☺*