Laboratory 4 report

Cédric Hölzl Antoine Brunner

May 2020

1 Introduction

In this laboratory, we had to create our own project using the DE0-nano-SoC. We chose to build upon thermal camera from the second lab and create a server capable of streaming the image from the thermal camera to a client through the Ethernet interface available on the board.

The server would run on the Linux OS on the hard-core ARM processor and would access the thermal camera through the interface that was created on the FPGA part of the board. This means that we had to find a way to access data captured by the thermal camera on the FPGA part from the Linux.

2 Thermal camera interface

We have reused the lepton camera interface programmed in VHDL from lab 2. It already suited our needs, so we didn't need to change anything to it. However, it is still important to remember how the interface worked. The interface allows us to control the camera through memory mapped registers. For example, there are a *COMMAND* and a *STATUS* registers that allow to instruct the camera to start capturing a frame and check its error status. There are also two memory regions that allow to read the raw image, as it was captured, or the image adjusted between the minimum and maximum temperature. This means that if we have access to that memory, we can control the camera by just reading and writing to certain addresses. The challenge was to be able to access that memory from the ARM processor, and it will be discussed in the next sections.

3 Linux OS

Since we needed to create a server, we needed to use the Ethernet interface. It would be possible to develop a baremetal application that uses the Ethernet interface, but it would be harder to implement. Instead, Linux provides us with drivers that allow to access it transparently through existing libraries. As we are programmers, we are lazy and don't want to reinvent the wheel, so that's why we chose to use Linux, instead of a baremetal application. In the previous lab, we have been able to boot Linux on our DE0-nano-SoC so we could reuse that configuration for this lab. As was said above, the challenge was to access the lepton camera interface from the Linux part.

4 Accessing the thermal camera from Linux

The fact that there is a Linux system running on the ARM processor means that we cannot access the memory directly anymore. OS's virtualize the memory so that several processes can run simultaneously. While this architecture provides a great abstraction over the hardware, it also means that we cannot directly access the physical memory. But fortunately, there is a way to map physical memory onto virtual memory, through the *mmap* function.

mmap allows to map a given file descriptor in a certain range of the virtual memory. On Linux systems, there is a special file */dev/mem* that represents the physical memory. Combining the two functionalities, we can first open that special file, and map it into the virtual memory. This gives us the ability to access arbitrary ranges of the physical memory from a Linux process. This approach has one drawback: opening */dev/mem* requires elevated privileges, so it means that our server needs to be run with *sudo*. This is not such a big problem for us since we are the administrators of the Linux system, but it could cause some problems if a normal user wants to run our server.

5 Server-client protocol

Now that we have a way to access the thermal camera images from a Linux process, we need to send it to the client. To do so, we can use the Linux socket programming libraries to set up a working server. We chose to send the image over UDP with one packet per line, along some metadata. The motivation behind that choice is that UDP hass less overhead than TCP (no handshakes,...) and that packet loss is not a major issue. Indeed if a packet containing a certain line is lost, we can simply display the line from the previous frame, and it is not going to be noticeable (at most it might have an effect similar to screen tearing).

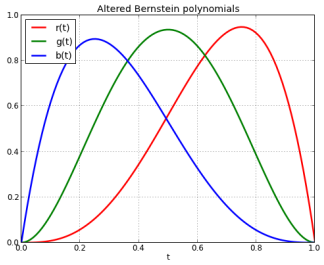
We will now explain our very simple server-client protocol. Once the server has created and bound a UDP socket, it then waits for a client to arrive. To start the communication to the server, a client has to send an empty (zero-length) packet to the server. Once the server received that packet, it starts sending the thermal image with one packet per line. Each packet first contains the index of the line, encoded in four bytes (for alignment purposes), and then contains the pixels of the line that take two bytes each. The client can send again a zero-length packet to the server to indicate that he wants to stop the connection. In that case, the server stops sending packets to that client and waits for the next client to connect. The server can also be stopped at any time by pressing on 'q'.

6 Clients

Since both of us were familiar with different libraries for rendering images to the screen, we have decided to create two clients to visualize the thermal image. Cedric wanted to experiment with a library called CACA, while Antoine already had some experience with a library called SFML.

6.1 Refresh Logic

While trying to reduce the performance impact of our software, we had some choices to make. Initialy, we refreshed the display for every line received. This however proved rather inefficient, updating the display 60 times for every frame, at 9 frames per seconds. Ideally, we would want to refresh the image every 60 lines, but with packet loss it might end up out of sync with the frames.



(a) The red, blue and green curves as the temperature varies

(b) The resulting color range

Figure 1: The Bernstein color mapping

We could add a frame counter to the packets but it will have the drawback that there is no way to detect missing packets. We made the choice in the end to refresh every 60 lines, resetting the counter when we receive the last line of any frame. This prevents the line counter from getting out of sync and guarantees that even with out of order packets and packet loss we still get a relatively good image.

6.2 Greyscale to RGB

The values that are read from the thermal camera are encoded on 14 bits and represent the temperature between the minimum and the maximum (we sample the adjusted buffer). To visualize them, we could simply map them to a grey scale color. While it would be easy to implement, the result is not really intuitive, or at least not visually pleasant. To improve the image visualization we have used a slightly more complex mapping using Bernstein polynomials. Those polynomials are not related in any way to color mapping, but it turns out that they yield a perfect color mapping for temperature visualization, so we chose to use them. Figure 1 shows the red green and blue curves that the Bernstein polynomials give, as well as visualization of the resulting color range.

6.3 CACA client

CACA stands for **C**olor-**A**S**C**II-**A**rt. It is a library that is aimed to transform media such as images or video into colored ASCII frames in terminal environments. It uses NCurses, is relativly lightweight, and has options for image processing and more, making it an interesting tool for this project. We used standard C sockets to communicate with the server. We managed to optimize performance making our program use at most 1.5% CPU while running.

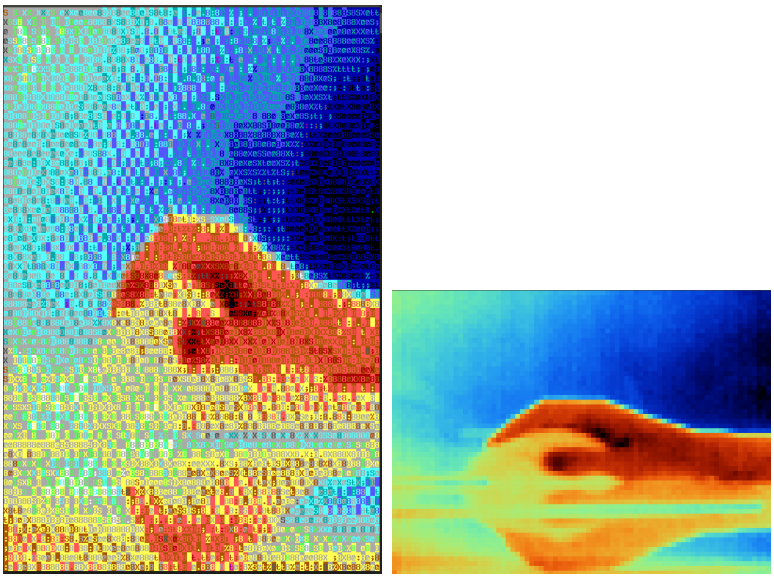
6.4 SFML client

SFML stands for **S**imple and **F**ast **M**ultimedia **L**ibrary. It is a cross-platform library that allows to build simple window applications and that also provides access to sockets. We have initially chosen to use asynchronous socket receiving in order not to block the interface while waiting for packets to arrive. While this was meant to make the application more responsive, it had the opposite effect. We noticed that the client was using 6 threads and almost 200% of the CPU (two cores). We believe that the threads come from how SFML was implemented, and in particular

from the asynchronous sockets. To try and optimize this, we switched back to synchronous socket receiving (meaning that the process can sleep while waiting for packets).We also tried to refresh the screen less often, which had a big effect on the performance. After those optimizations, the usage dropped from 200% (2 cores) to around 2%.

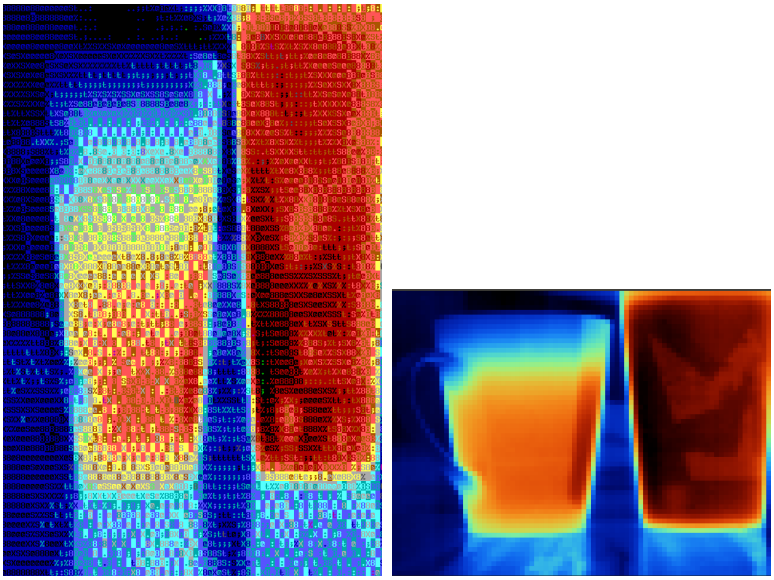
7 Gallery

In this section, we show some images that we captured using the thermal camera and the two clients. Note that the images captured using the CACA client are stretched, because ASCII characters are not perfectly square in terminals, but are usually twice as high as they are wide.



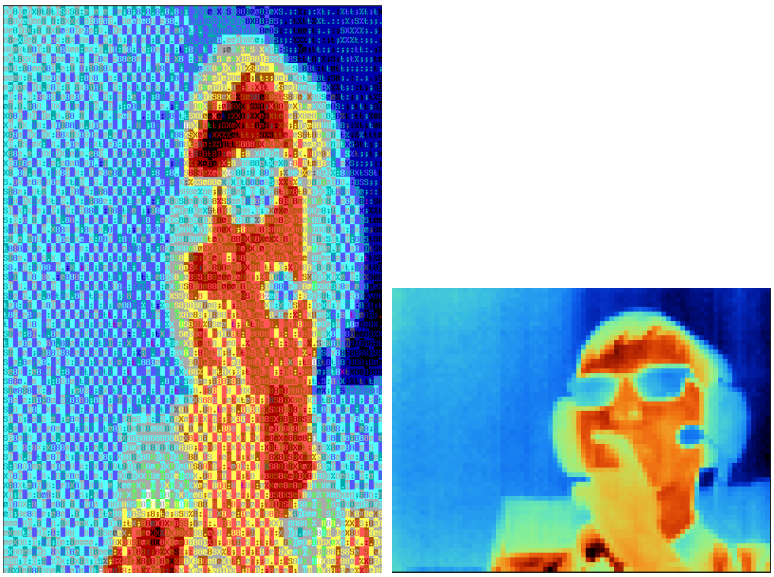
(a) CACA (b) SFML

Figure 2: A hand holding a mouse



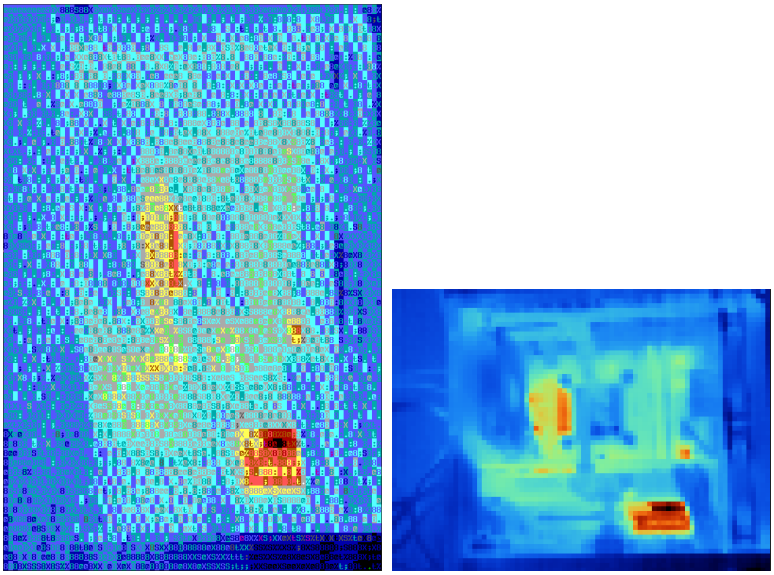
(a) CACA (b) SFML

Figure 3: A mug and plastic cup holding boiling water



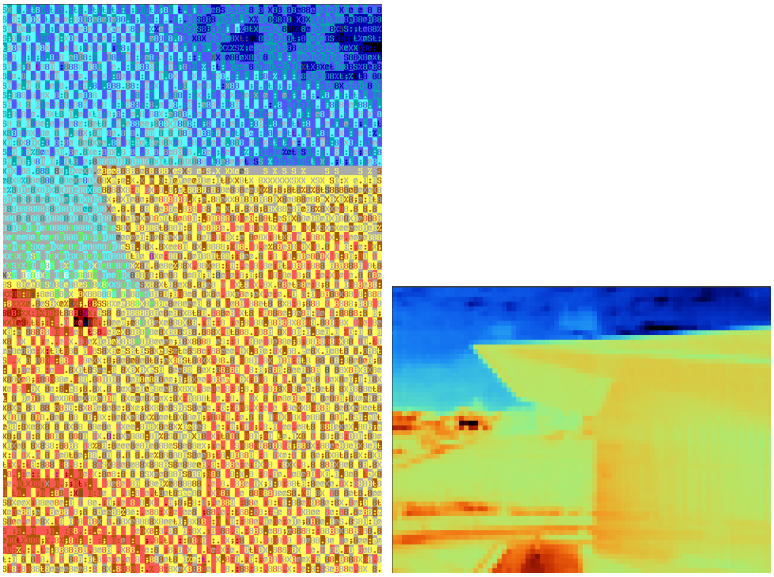
(a) CACA (b) SFML

Figure 4: Cedric's face, with his glasses and microphone



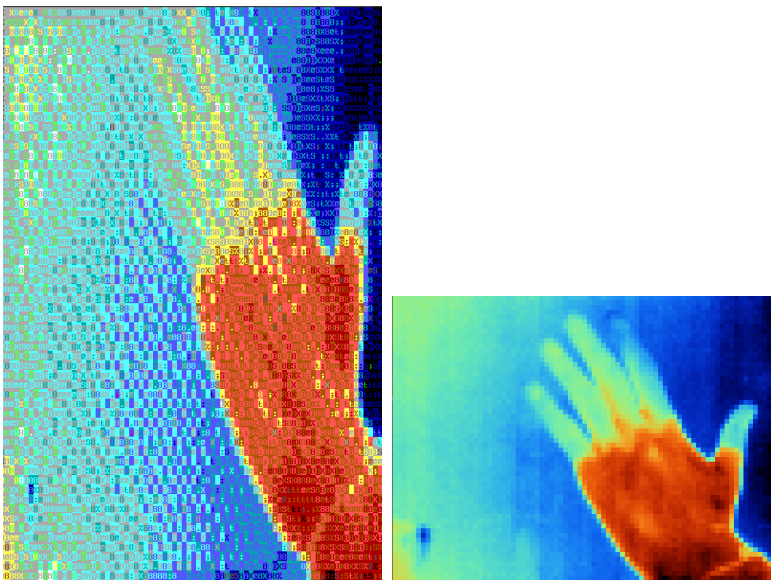
(a) CACA (b) SFML

Figure 5: Inside a water-cooled computer (top-left cpu, bottom right SSD)



(a) CACA (b) SFML

Figure 6: The Swiss Tech Convention Center



(a) CACA (b) SFML

Figure 7: Hand with wet fingers

8 Conclusion

It was very interesting to work on a project were we have to write a system from the hardware to the software. It is not often that we get to program hardware interfaces, sockets, and window applications in the same project. Since a lot of the work had already been done in the previous labs, we didn't struggle too much with the hardware part, and we had a lot of time left to program the server and the clients.

We also have multiple ideas of additions to extend our project. One would be to make use of the servos to control the infrared camera's pitch and yaw from the client. An other idea would be to transmit the real min/max temperatures so that the client knows what temperature the colors translate to. A final idea that we had would be to allow the server to send the stream to multiple clients, either using multicast or by keeping a list of connected clients. Of course, if we had more time, we would certainly tinker a bit more with that project, but unfortunately all things have an end.

9 Appendix: archive structure

Since we have used a lot of files for this project, we have decided not to include them in the report, so that it doesn't get absurdly long. We have included the files in which we implemented the server and the clients in the archive using the structure described just below. Note that we do not include VHDL files in the archive because we have taken the files from the second laboratory without modifying them. There are three main folders in the archive, where you can find the server and the two clients respectively. Note that in the server, we have mainly modified the files *server.h*, *server.c* and *app.c*, but we include the other files for completeness.

```
lab4.zip
├── CedricHoelzl_AntoineBrunner_lab4_EmbeddedLinuxMiniProject.pdf
├── client-caca
│   └── main.cpp
├── client-sfml
│   └── main.cpp
└── server
    ├── lepton
    │   ├── lepton.h
    │   ├── lepton.c
    │   └── lepton_regs.h
    ├── hps_soc_system.h
    ├── iorw.h
    ├── server.h
    ├── server.c
    └── app.c
```