Project Of Systems-On-Chip (CS-309)

Lab 4: Mini-Project

MAROUF Imad Eddine
ESER Can

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

***Abstract*—In this report, we present the details and steps used to assemble all created interfaces during the previous labs, in order to create an application on an embedded Linux system. In this laboratory, we targeted creating a mini-project which takes an image captured by an Flir Lepton camera which can be moved by joysticks and display captured image on an LCD display. 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 fourth laboratory is to assemble what we did in the previous labs during this course. To provide a software for an embedded Linux system that uses various peripherals seen like PWM, Lepton Camera, Joystick Interface. As well using provided interfaces like frame-buffer manager, and VGA sequencer to interface the Cyclone-V with LCD display, which can a backbone system for creating interesting projects.

In summary, the goal of this laboratory can be divided into :

- *1- Enabling Linux's frame-buffer support.*
- *2- Compiling a custom frame-buffer driver for our frame-buffer manager and VGA sequencer.*
- *3- Loading the custom frame-buffer into the Linux kernel.*
- *4- Provide an application which takes usage of the interfaces developed previously.*

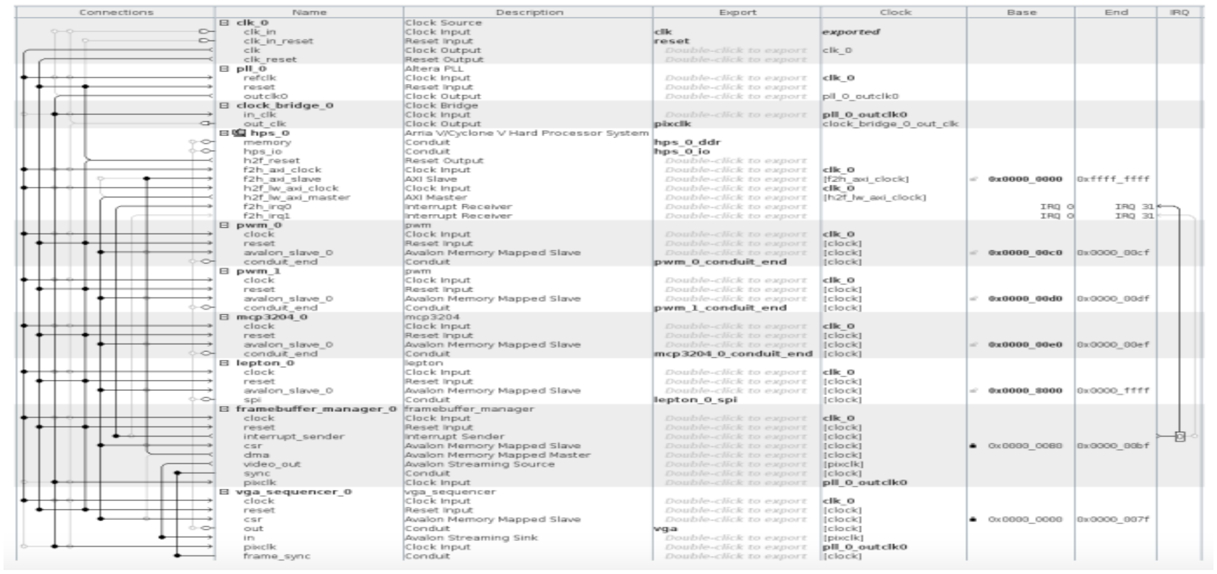
Therefore, the application we would like to develop in this laboratory is make use of the provided Flir lepton thermal camera which showed significant importance especially during this recent months in order to detect high temperature variations and fever of sick persons in airports and transport stations throughout covid-19 pandemic. This camera is attached to a pan-tilt which can be moved left-right, and bottom-up in order to have a wide angle of view using joysticks.

A key advantage that we learnt in this laboratory is the importance of using Linux's built-in drivers that support a vast array of peripherals, including many of them found on the DE0-Nano-SoC board. As well, the Top-Down design process. In order to create an interesting project, it is more efficient if we can divide the overall architecture into small components that communicate with each other, and develop each one separately which facilitates the development and debugging process in the future.

The group is composed of two members, and the tasks follow a progressive approach. The COVID-19 pandemic situation this year requires remote work. For this reason, we have organized ourselves to work together through Zoom, Overleaf and WhatsApp. Weekly meetings and workflow are organized through Zoom's shared screen.

I. System Architecture

A general block diagram of the used peripherals in our project is illustrated in Fig 2. Our aim during this laboratory



Name	Description	Export	Clock	Base	End	IRQ
⊟ clk_0	Clock Source					
clk_in	Clock Input	clk	exported			
clk_in_reset	Reset Input	reset				
clk	Clock Output	Double-click to export	clk_0			
clk_reset	Reset Output	Double-click to export				
⊟ pll_0	Altera PLL					
refclk	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export				
outclk0	Clock Output	Double-click to export	pll_0_outclk0			
⊟ clock_bridge_0	Clock Bridge					
in_clk	Clock Input	Double-click to export	pll_0_outclk0			
out_clk	Clock Output	pixclk	clock_bridge_0_out_clk			
⊟ hps_0	Arria V/Cyclone V Hard Processor System					
memory	Conduit	hps_0_ddr				
hps_io	Conduit	hps_0_io				
h2f_reset	Reset Output	Double-click to export				
f2h_axi_clock	Clock Input	Double-click to export	clk_0			
f2h_axi_slave	AXI Slave	Double-click to export	[f2h_axi_clock]	0x0000_0000	0xffff_ffff	
h2f_lw_axi_clock	Clock Input	Double-click to export	clk_0			
h2f_lw_axi_master	AXI Master	Double-click to export	[h2f_lw_axi_clock]			
f2h_irq0	Interrupt Receiver	Double-click to export		IRQ 0	IRQ 31	
f2h_irq1	Interrupt Receiver	Double-click to export		IRQ 0	IRQ 31	
⊟ pwm_0	pwm					
clock	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clock]			
avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_00c0	0x0000_00cf	
conduit_end	Conduit	pwm_0_conduit_end	[clock]			
⊟ pwm_1	pwm					
clock	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clock]			
avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_00d0	0x0000_00df	
conduit_end	Conduit	pwm_1_conduit_end	[clock]			
⊟ mcp3204_0	mcp3204					
clock	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clock]			
avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_00e0	0x0000_00ef	
conduit_end	Conduit	mcp3204_0_conduit_end	[clock]			
⊟ lepton_0	lepton					
clock	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clock]			
avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_8000	0x0000_ffff	
spi	Conduit	lepton_0_spi	[clock]			
⊟ framebuffer_manager_0	framebuffer_manager					
clock	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clock]			
interrupt_sender	Interrupt Sender	Double-click to export	[clock]			0
csr	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_0080	0x0000_00bf	
dma	Avalon Memory Mapped Master	Double-click to export	[clock]			
video_out	Avalon Streaming Source	Double-click to export	[pixclk]			
sync	Conduit	Double-click to export	[clock]			
pixclk	Clock Input	Double-click to export	pll_0_outclk0			
⊟ vga_sequencer_0	vga_sequencer					
clock	Clock Input	Double-click to export	clk_0			
reset	Reset Input	Double-click to export	[clock]			
csr	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0000_0000	0x0000_007f	
out	Conduit	vga	[clock]			
in	Avalon Streaming Sink	Double-click to export	[pixclk]			
pixclk	Clock Input	Double-click to export	pll_0_outclk0			
frame_sync	Conduit	Double-click to export	[clock]			

Figure 1: QSYS system with LCD Interface

is to used the already developed modules in order to display frames captured by the thermal camera Flir LEPTON on LCD screen. Additionally, as a feature we would like to attach thermal camera with the pan-tilt which can be moved using two-servo motors controlled by PlayStation joysticks in order to have a full view of the surroundings.

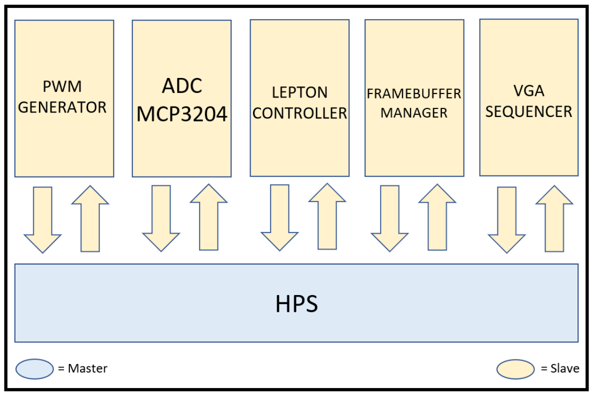


Figure 2: Diagram of full system

A. PWM

Pulse Width Modulation, this module is responsible for generating signals with different duty cycles in order to rotate two servo-motors of pan-tilt attached to the thermal Lepton camera.

B. LEPTON

This module is responsible for interconnecting Cyclone V with Thermal camera Flir. Therefore, it will be used in this laboratory to display its output on LCD screen rather then just saving it on the embedded linux filesystem.

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

- *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.

C. MCP3204

MCP3204 is a module used to convert analog signals into digital signals. It uses Serial Peripheral interface (SPI), this later communication protocol is based on connection between Slave and Masters components, which are in our project ADC and FPGA respectively. In order to establish, the communication between them, we did design a VHDL module in the laboratory 1.0 that can supply commands to the ADC and can read the converted values back through the SPI bus.

This module is used to control a pan-tilt composed of two servo motors and a camera at the top with joysticks.

D. *Frame-buffer manager*

Frame-buffer manager is a general Direct Memory Access (DMA) used to move data stored in memory by ARM processor which is responsible for processing frame captured by Flir camera in efficient way, to LCD through the heavyweight FPGA2HPS bridge which is used to control HPS from FPGA.

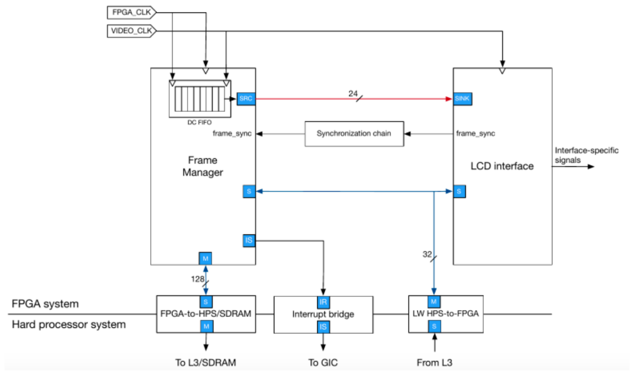


Figure 3: Video DMA + LCD Interface

Due to the fact that CPUs are very slow when it comes to addressing I/O pins. Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work resulting to what is know as ”Lag” phenomenon. With DMA, the ARM CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt signal from the DMA controller (DMAC) upon completion of a data transfer.

E. *VGA-sequencer*

This module is responsible for handling conflicts of resolution between Lepton and LCD, since the resolution of Lepton is (80x60) as well its frames per second rate is 9 FPS is quite slow comparing to LCD which can display frames at higher rates with larger resolution (480x272).

LCD Screen used in this laboratory should be clocked with 9MHz. Thus, Phase-Locked Loop (PLL) is required to avoid the issue of synchronisation between camera and LCD interface which takes as input the standard FPGA 50 MHz clock and creates a 9 MHz output clock for the LCD interface, the operation is illustrated in the Figure 3.

II. Software Architecture

A. *Prerequisites*

As stated, in the previous section. This laboratory comes as an assembly of all developed interfaces to create an application. But, first of all we have to prepare a working operating system on an SD card as we done in laboratory 3 using a provided shell script, SD card should contains at least:

- A boot partition that contains the kernel image and the compiled device tree.
- A partition containing the root filesystem, i.e. the filesystem automatically mounted by the kernel that contains the root directory “/”.
- A raw preloader partition which contains the initialization code that is executed at boot time.
- Created modules for peripherals in previous laboratories like PANTILT, MCP3204, LEPTON.

After successfully, running the operating system, we have to enable Linux’s Frame-buffer support and re-compile the kernel

B. *Main Steps*

As requested in the lab manual we can divide the roadmap to install the linux operating system into seven main steps explained below:

- *1- Compile the preloader: The preloader is used to sets up the HPS. We used the preloader generator on Quartus to generate and compile the preolader and copie its binary to the sdcard target directory.*

- *2- Compile the bootloader: The bootloader allows to load the linux kernel. We downloaded the official U-Boot sources online and cross-compiled it.*

- *3- Compile mainline linux kernel: We downloaded the latest version of Linux kernel and cross-compiled it.*

- *4- Compile the device tree of DE0-Board: In this step, we need to compile the device tree blob for the DEO-Nano-SoC and copy the linux zImage binary and the linux device tree blob to the sdcard target directory.*

- *5- Set up a root file system: After having a fully-working linux machine, then we installed a linux distribution in order to have more tools and functionality. We needed to install Ubuntu Core on our DE0-Nano-SoC which is is the minimal root filesystem (rootfs) needed to run Ubuntu. It consists of a very basic command-line version of the distribution, and can be customized to eventually ressemble the desktop version of Ubuntu. Most importantly, it comes with a package manager. The final step was to configure the rootfs directly on the DE0-Nano-SoC when it boots the operating system* *for the first time. For this, we used a provided shell script “config_system.sh” on our host machine and place it in the extracted rootfs and configure the rootfs to launch our script when it boots for the first time.*

- *6- Partition and format SD card: After having all the files needed to create the final sdcard, we mounted the sdcard partitions and wrote the preloader to the ”a2” partition, FPGA .rbf file, U-Boot .img file, U-Boot .scr file, linux zImage file, and linux .dtb file to the FAT32 partition and the customized Ubuntu Core root filesystem to the EXT3 partition.*

- *7- Boot the Linux system: Finally, to boot the Linux system, we wired up the DE0-Nano-SoC by connecting the Power cable, USB-Blaster cable, Ethernet cable and UART cable, then, we pluged in the microSD card. At this step, we should be sure to set the MSEL switch on the top side of the DE0-Nano-SoC to ”00000”. By using minicom, we should disable the hardware flow control, and we can Power-on the DE0-Nano-SoC. Finally, once we are logged in, we should call the post-installation configuration script to install the required tools from the package manager. We need to be sure to be connected to the internet and share properly the host pc’s internet connection with the DE0-Nano-SoC trough the Ethernet cable.*

The tutorial [2] provided, explains in much more details each step above in order to complete the setup.

III. Linux’s framebuffer support

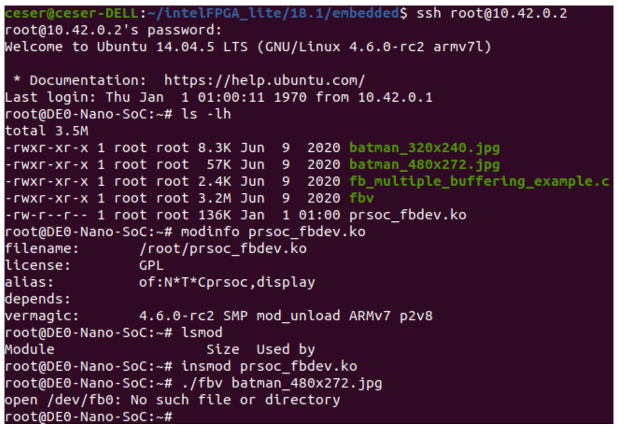
It’s possible for us to write a C code which generates a frame and programs the DMA unit in order to transfer the frame taken with the FLIR Camera to the LCD. But we need to know that programming the DMA unit isn’t a non-trivial task because a user space application has no way of determining the physical address of a page or of instructing Linux to allocate the memory contiguously in physical memory. Linux assigns virtual addresses for all elements in a user space application. We know that all operating systems supporting a graphical user interface have some abstraction for a display. APIs describes standard operations supported by all displays with the advantage of controlling the display with knowing nothing about the specifics of the display hardware. In Linux, the memory region used to hold an image to be displayed on a screen is the framebuffer. So we need firstly to enable Linux’s framebuffer abstraction by configuring the kernel. To do this, we need to activate manually framebuffer support with the ”Linux/arm Kernel Configuration” and to compile the kernel before copying it to the sdcard.

Drivers are independent of board models in order to be used for many boards. Therefore, a compiled structure called the Device Tree is loaded in the kernel’s memory by the bootloader. The kernel makes this structure available to its drivers in order to read their configuration parameters. In this case, a new board requires only a device tree to be written.

The Device Tree contains information relative to the hardware design of the framebuffer manager and VGA sequencer which will be read by the framebuffer driver in order to control the different devices.

The last last step is to compile the framebuffer driver provided. In order to launch the output binary file by using ssh.

After successfully, running the OS on SD card. We had to compile the customized device tree which describes board’s configuration. We were not able to generate the device tree file with ”DTB” extension labeled as step 10 in the laboratory handout [4]. Which forbids establishing connection with LCD display in order to display the captured IR image. As illustrated in the Figure 6 below stating that the driver fb0 has not been created into drivers folder in the Linux system. Although, we did try to change the parameters in framebuffer file which corresponds to our small LCD display regarding height and width.

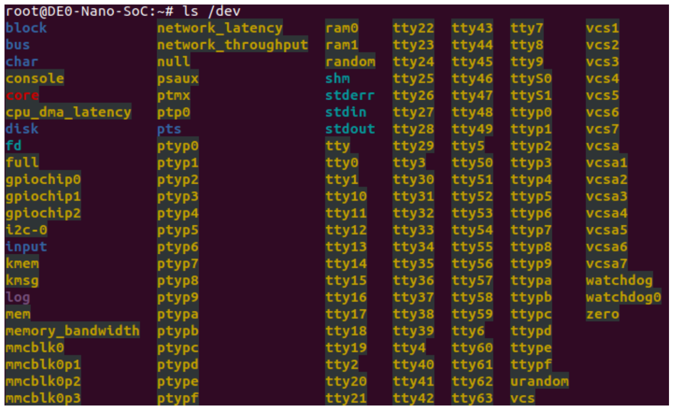


```
ceser@ceser-DELL:~/intelFPGA_lite/18.1/embedded$ ssh root@10.42.0.2
root@10.42.0.2's password:
Welcome to Ubuntu 14.04.5 LTS (GNU/Linux 4.6.0-rc2 armv7l)

 * Documentation:  https://help.ubuntu.com/
Last login: Thu Jan  1 01:00:11 1970 from 10.42.0.1
root@DE0-Nano-SoC:~# ls -lh
total 3.5M
-rwxr-xr-x 1 root root 8.3K Jun  9  2020 batman_320x240.jpg
-rwxr-xr-x 1 root root  57K Jun  9  2020 batman_480x272.jpg
-rwxr-xr-x 1 root root 2.4K Jun  9  2020 fb_multiple_buffering_example.c
-rwxr-xr-x 1 root root 3.2M Jun  9  2020 fbv
-rw-r--r-- 1 root root 136K Jan  1 01:00 prsoc_fbdev.ko
root@DE0-Nano-SoC:~# modinfo prsoc_fbdev.ko
filename:       /root/prsoc_fbdev.ko
license:        GPL
alias:          of:N*T*Cprsoc,display
depends:
vermagic:       4.6.0-rc2 SMP mod_unload ARMv7 p2v8
root@DE0-Nano-SoC:~# lsmod
Module                  Size  Used by
root@DE0-Nano-SoC:~# insmod prsoc_fbdev.ko
root@DE0-Nano-SoC:~# ./fbv batman_480x272.jpg
open /dev/fb0: No such file or directory
root@DE0-Nano-SoC:~#
```

Figure 4: Error when testing the driver

Because of this error, we were not able to use the framebuffer driver and to integrate the LCD on our project. It’s why we decided to make an application by using only the Thermal camera and the pantilt controlled by both joysticks. Our new goal was to take a picture with one joystick and to control the pantilt with the other one.



```
root@DE0-Nano-SoC:~# ls /dev
block             network_latency     ram0    tty22  tty43  tty7     vcs1
bus               network_throughput  ram1    tty23  tty44  tty8     vcs2
char              null                random  tty24  tty45  tty9     vcs3
console           psaux               shm     tty25  tty46  ttyS0    vcs4
core              ptmx                stderr  tty26  tty47  ttyS1    vcs5
cpu_dma_latency   ptp0                stdin   tty27  tty48  ttyp0    vcs6
disk              pts                 stdout  tty28  tty49  ttyp1    vcs7
fd                ptyp0               tty     tty29  tty5   ttyp2    vcsa
full              ptyp1               tty0    tty3   tty50  ttyp3    vcsa1
gpiochip0         ptyp2               tty1    tty30  tty51  ttyp4    vcsa2
gpiochip1         ptyp3               tty10   tty31  tty52  ttyp5    vcsa3
gpiochip2         ptyp4               tty11   tty32  tty53  ttyp6    vcsa4
i2c-0             ptyp5               tty12   tty33  tty54  ttyp7    vcsa5
input             ptyp6               tty13   tty34  tty55  ttyp8    vcsa6
kmem              ptyp7               tty14   tty35  tty56  ttyp9    vcsa7
kmsg              ptyp8               tty15   tty36  tty57  ttypa    watchdog
log               ptyp9               tty16   tty37  tty58  ttypb    watchdog0
mem               ptypa               tty17   tty38  tty59  ttypc    zero
memory_bandwidth  ptypb               tty18   tty39  tty6   ttypd
mmcblk0           ptypc               tty19   tty4   tty60  ttype
mmcblk0p1         ptypd               tty2    tty40  tty61  ttypf
mmcblk0p2         ptype               tty20   tty41  tty62  urandom
mmcblk0p3         ptypf               tty21   tty42  tty63  vcs
```

Figure 5: Missing of fb0 directory in the booted Linux

IV. Software

We used the ARM DS-5 software development solution for Linux-based and bare-metal embedded systems which includes an Eclipse-based IDE, compilation tools, a fully featured, graphical debugger and Arm Compiler 5 and Arm Compiler 6 toolchains that enable us to build embedded Linux code. The Nios II-specific I/O macros that we used in our drivers for the PWM, MCP3204 and LEPTON are not useable on the HPS because the ARM processor can't decode the Nios II assembly. So the first thing we need to do is to replace all the Nios II I/O macros with their equivalent HPS functions in our drivers. To do so, we have to create a header file with architecture-independent I/O macros (”io_custom.h”). This header file contains all Nios II-specific I/O macros with their equivalents. As an example, ”ioc_write_8(base, ofst, data)” will replace ”IOWR_8DIRECT((base), (ofst), (data))”. Finally, we need to include this header file on our app.c code. As compiler, we use GCC for Nios II and ARM versions of the driver. So the header file will look which architecture-specific symbols defined in the compiler and defines I/O macros which are usable both by HPS and Nios II code so we don't have to hard-code any architecture-specific instructions anywhere.

The main advantage of the Cyclone V is the ability to have the HPS and FPGA communicate with each other easily. But contrary to bare-metal application, this is not possible while the HPS is running Linux, as user code doesn't have the right to access hardware directly. We can write a device driver for each target peripheral we want to access with our user code and package it in a loadable Linux kernel module but we need to know how to write a device driver for this. A simpler technique often used in embedded Linux environments is to leverage the virtual memory system in order to access any MEMORY-MAPPED peripherals. This last method doesn't require any kernel code to be written but we need to connect with root privileges on DS-5. The tutorial [2], explains in much more details this part and provide a zip ”DE0_Nano_SoC_demo.zip” with a source file which contains needed functions for this step.

Firstly, we need to be able to acces to HPS peripherals addresses in order to interact with them. A process can have access to another virtual memory region by using the ”mmap() function which maps another memory region into the running process virtual address space. It's why we have to mmap() the HPS peripherals memory regions into our address space. With Linux, all devices are represented as a file. ”/dev/mem” is a character device file that is an image of the main memory of the computer. Byte addresses in ”/dev/mem” are interpreted as physical memory addresses. In order to memory-mapping this file, we need to open it before. The function **”open_physical_memory_device()”** allows to map the address space related to the FPGA peripherals into user space so we can interact with them.

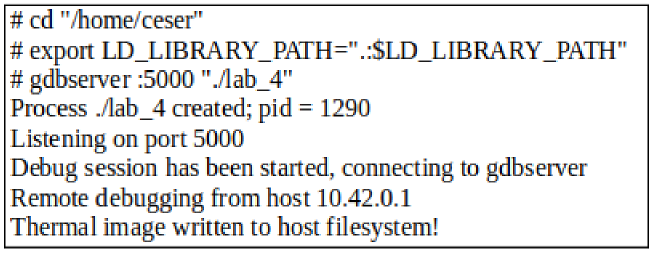
After opening the physical memory file, we need to memory-map a subset of it into our process virtual address space. The FPGA peripherals are connected to the h2f_lw_axi_master, so their base addresses is calculated from that of the h2f_lw_axi_master. The function **”*mmap_h2f_lw_axi_master”** allows us to memory-map the FPGA peripherals.

Finaly, after accessing FPGA peripherals, we have to remove any unneeded memory mapipings and to close the physical memory file descriptor before our application terminates. The function **”munmap_h2f_lw_axi_master”** allows to unmap our peripheral's memory-mapping and the function **”close_physical_memory_device”** allows to close the physical memory file descriptor.

We have complete app.c file such that moving the right joystick to the bottom past a certain threshold triggers a frame capture with the Lepton controller and to be able to control the pantilt with the left joystick. The function **”handle_pantilt”** is used to interpolate the left joystick position the servo's minimum and maximum duty_cycle and to configure servos with interpolated joysticks values. **”handle_lepton”** allows to take a picture with Lepton when the right joystick is higher than an horizontal threshold value. Finally, the thermal image is written to embedded Linux filesystem.

V. Results

Once our system is fully configured, we can boot the Linux system and access to it through SSH as Root directly by creating an SSH Remote Connection on DS-5. After setting a Debug Configuration, we can build and test our project. As the J14 and J15 pins of our extension board provided are not functional, we are not able to see if the control of the pantilt with the left joystick is working properly. Therefore, when the right joystick is oriented towards bottom, the Lepton thermal camera takes a picture and this picture is written to embedded Linux filesystem on our sdcard. If the framebuffer driver was working on our Linux, we could have display this thermal image on the LCD.



```
# cd "/home/ceser"
# export LD_LIBRARY_PATH=".:$LD_LIBRARY_PATH"
# gdbserver :5000 "./lab_4"
Process ./lab_4 created; pid = 1290
Listening on port 5000
Debug session has been started, connecting to gdbserver
Remote debugging from host 10.42.0.1
Thermal image written to host filesystem!
```

Figure 6: DS-5 Debugging App Console

Conclusion

The Lab 4 of ”Projet de Systems-on-Chip CS-309” allowed us to review all what we have seen during this course, as well the methodology required to have a practical system, by developing many interfaces and combine them in order to have robust and easy to debug system. Additionally, the main skill required is to take full usage of HPS and FPGA, each with its corresponding good utilisation whether reading/writing on I/O pins, fetching from memory or processing.

Although, we did not manage successfully to display the output frame captured by lepton camera on LCD screen due to the problem of the frame-buffer. But, we could say we did learn how to set setup an application into embedded Linux system, as well the overall working process of our desired project. We will try to figure out the root of the error and find a solution to it in the coming weeks, which is a crucial skill in the learning process in order to develop good systems and be a competent engineer.

References

Lab Handouts :

[1] R. Beuchat, Lab 4: Mini-Project.

Tutorials:

[2] SoC-FPGA Design Guide [DE0-Nano-SoC Edition].
[3] Cyclone V SoC Examples: SD Operating system.
[4] Using LCDs