# Lab 3.0
# Hybrid Systems

## Lab 2.0 + 2.1 – Thermal Camera & Full-System Integration

The goal of the lab 2 series was to design a specialized component in order to interface with the Lepton thermal camera. You built 2 pieces of this interface, then created an *instantiable component* that represents it within a *system integration tool* (Qsys). You then used the tool to compose a full system that captures images from the Lepton and saves the output to a file on your host machine.

As such, you create a system that runs *exclusively* on the programmed *FPGA fabric*.

## Lab 3.0 – Hybrid Systems

### Introduction

The Cyclone V SoC architecture is a hybrid design: it does not solely consist of an FPGA, but rather of a single chip containing both a HPS[1] *and* an FPGA. The HPS consists of an ARM-A9 MPCore, which is a general-purpose *application processor*. In this lab, we will explore using the HPS instead of the softcore Nios II processor you have used until now.

Although the HPS was designed to run an operating system, you still lack familiarity with the tools needed to create such a system. We will come back to this next week in lab 3.1. For the moment, you are instead going to use the HPS to run bare-metal code, as the setup is much shorter and gives you more time to get acquainted with the tools.

### Getting Started

Getting the HPS up and running is a much more involved process compared to the *embedded* Nios II processor, which is to be expected since it is a *general-purpose* processor.

We would normally tell you to **RTFM**, but given the huge search space of documentation available, we have written a step-by-step tutorial for getting up to speed with development for Cyclone V SoC-based devices. You can follow the tutorial by reading the SoC-FPGA Design Guide.

[1] *Hard* Processor System (a.k.a not synthesized on the FPGA fabric, but a "real" processor)

René Beuchat, Philémon Favrod, Sahand Kashani

The tutorial was written for the “base” DE0-Nano-SoC board (without the PrSoC extension board), however the steps needed to get an application running are 99.9999...% (you get the idea ☺) similar for both devices, so you should have no problem adapting the steps to suit the PrSoC extension board.

The tutorial is quite long, but you don’t need to read all of it, as it includes material that we cover in future labs. The chapters you should read for this lab are the following:

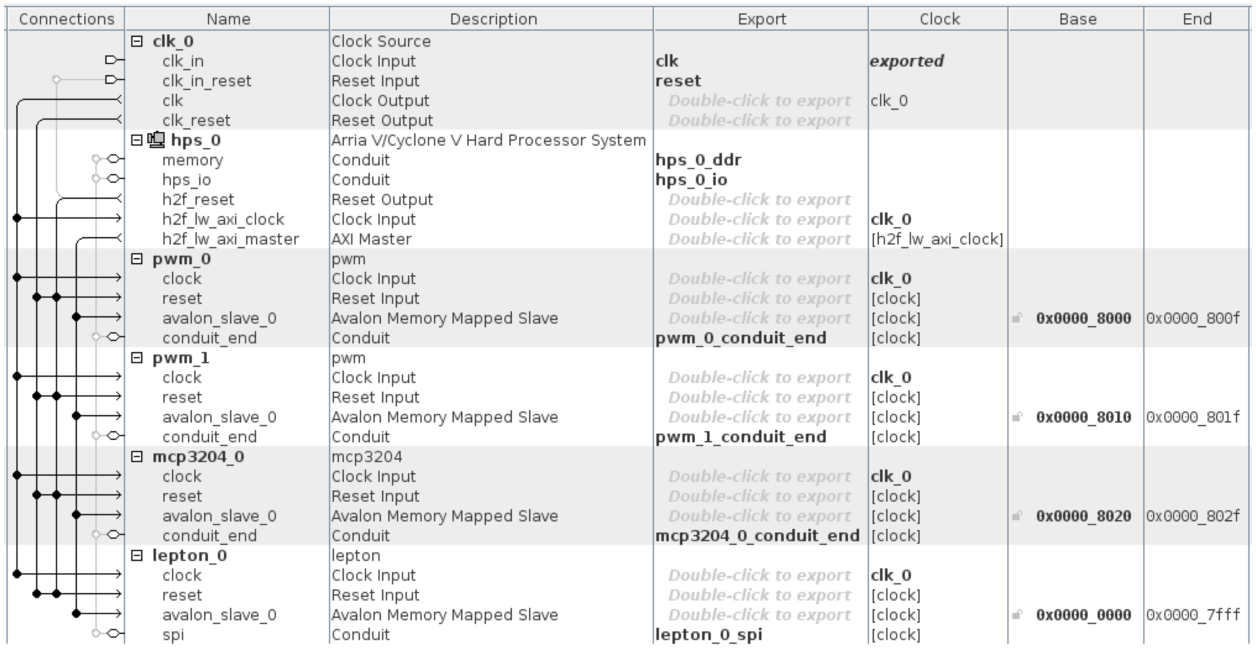
- Chapter 4: Prerequisites
- Chapter 7: Cyclone V Overview
    - 7.2: Features of the HPS
    - 7.4: HPS-FPGA Interfaces
    - 7.5: HPS Address Map
    - 7.6: HPS Booting and FPGA Configuration
- Chapter 8: Using the Cyclone V – General Information
- Chapter 9: Using the Cyclone V – Hardware
    - 9.3: System Design with Qsys – HPS
    - 9.4: Generating the Qsys System
    - 9.5: Instantiating the Qsys System
    - 9.6: HPS DDR3 Pin Assignments
    - 9.7: Wiring the DE0-Nano-SoC
    - 9.8: Programming the FPGA
- Chapter 11: Using the Cyclone V – HPS – ARM – General
    - 11.2: Generating a Header File for HPS Peripherals
    - 11.3: HPS Programming Theory
- Chapter 12: Using the Cyclone V – HPS – ARM – Bare-metal

If you understand how the system is built and how the different components interact together, you will see that there is not much code to write for this lab. All the information you need can be found in the tutorial, but whenever in doubt, don’t hesitate to ask questions!

## Your Task

### Hardware

The goal is to create a HPS version of the system you have built until now during the labs. The system you have to implement is shown in Figure 1.



| Connections | Name | Description | Export | Clock | Base | End |
|---|---|---|---|---|---|---|
| | ⊟ **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* | | | |
| | ⊟ **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* | | | |
| | 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] | | |
| | ⊟ **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_8000** | 0x0000_800f |
| | 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_8010** | 0x0000_801f |
| | 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_8020** | 0x0000_802f |
| | 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_0000** | 0x0000_7fff |
| | spi | Conduit | **lepton_0_spi** | [clock] | | |

FIGURE 1. HYBRID QSYS SYSTEM TO IMPLEMENT

As you can see, the system no longer contains a Nios II processor, on-chip memory, or JTAG UART. The 3 former *soft* components have all been replaced with their equivalent *hard* equivalents. Indeed, since the HPS is a *hard* processor, it has a *hard memory controller* connected to dedicated *off*-chip DDR3 memory, and also has a dedicated *hard* UART.

To spare you the burden of entering the HPS' DDR3 timings by hand, we provide you a template where the "`SDRAM`" tab of the "`Arria V/Cyclone V Hard Processor System`" in Qsys is already filled in. Note though that you still need to configure all the other tabs!

### Software

#### Porting the FPGA peripheral drivers to the HPS

As you read through the tutorial, you will learn that the Nios II-specific I/O macros (`IOxx_xxDIRECT`) you have been using until now in your drivers for the PWM, MCP3204, and the Lepton are not useable on the HPS. This is normal, as they translate to Nios II assembly, and hence the HPS (an *ARM* processor) cannot decode them.

However, we spent considerable time developing these drivers, so we want to use them in our HPS system. Therefore, the first thing you need to address is to replace all the Nios II I/O macros with their equivalent HPS functions in your drivers. The tutorial contains the names of the functions you are looking for ☺.

There are 2 ways to achieve this. As usual, the first is for *slackers* who want a fast and easy way out, whereas the second is geared towards those who want to do things correctly (I'm *sure* you want to be part of the 2nd category):

1. *Copy* all your Nios II peripheral drivers into a separate folder dedicated to containing HPS versions of the peripheral drivers. For all driver files in the HPS folder, replace all Nios II-specific I/O macros (`IOWR_xxDIRECT` & `IORD_xxDIRECT`) with their HPS equivalents.

2. The previous solution suffers from a *maintainability* problem. If you ever detect a bug in your driver code (who knows?), you now have to apply fixes to multiple copies of the driver source code. This can be described in two words: **CODING HORROR**. There must be a way to avoid copying the driver code as all versions are, in fact, *identical* except for the I/O instructions.

   In computer science, it is well-known that introducing a level of indirection often helps solve problems. In our case, a clean solution to the duplicated driver code issue would be to use a header file such as the one shown in Figure 2.

```
#ifndef __IO_CUSTOM__
#define __IO_CUSTOM__

#ifdef __nios2_arch__
    #include <io.h>

    #define ioc_write_8(base, ofst, data)  (IOWR_8DIRECT((base), (ofst), (data)))
    #define ioc_write_16(base, ofst, data) (IOWR_16DIRECT((base), (ofst), (data)))
    #define ioc_write_32(base, ofst, data) (IOWR_32DIRECT((base), (ofst), (data)))
    #define ioc_read_8(base, ofst)         (IORD_8DIRECT((base), (ofst)))
    #define ioc_read_16(base, ofst)        (IORD_16DIRECT((base), (ofst)))
    #define ioc_read_32(base, ofst)        (IORD_32DIRECT((base), (ofst)))
#else
    #include <socal/socal.h>

    #define ioc_write_8(base, ofst, data)  (alt_write_byte((uintptr_t) (base) + (ofst), (data)))
    #define ioc_write_16(base, ofst, data) (alt_write_hword((uintptr_t) (base) + (ofst), (data)))
    #define ioc_write_32(base, ofst, data) (alt_write_word((uintptr_t) (base) + (ofst), (data)))
    #define ioc_read_8(base, ofst)         (alt_read_byte((uintptr_t) (base) + (ofst)))
    #define ioc_read_16(base, ofst)        (alt_read_hword((uintptr_t) (base) + (ofst)))
    #define ioc_read_32(base, ofst)        (alt_read_word((uintptr_t) (base) + (ofst)))
#endif

#endif /* __IO_CUSTOM__ */
```

FIGURE 2. CUSTOM HEADER FILE WITH ARCHITECTURE-INDEPENDENT I/O MACROS (FOR NIOS II AND ARM ONLY)

We use GCC as the compiler both for the Nios II and ARM versions of the driver, so the header file takes advantage of this information to lookup which architecture-specific symbols are defined in the compiler (`__nios2_arch__`), therefore allowing us to know what its target architecture is. Once the target architecture is known, we can then generate architecture-*in*dependent wrapper macros around the architecture-specific I/O instructions. Essentially, this header file defines I/O macros which are useable both by HPS and Nios II code so we don't have to hard-code any architecture-specific instructions anywhere. Maintainability problem solved ☺.

## lepton.c

Unlike in lab 2.0 where we wrote the captured image to a *file*, we don't have access to any host filesystem when running bare-metal code on the HPS. In order to see the output of the Lepton, you must instead *print* the captured to the serial console. All data outputted on the serial console can then be saved on your host machine by *manually* copy-pasting the output from the serial console into a file (the ASCII nature of the PGM format we use to represent the image ensures that this simple copy-pasting procedure is adequate). We recommend you write the following function in `lepton.c` for this purpose:

```
/**
 * lepton_print_capture
 *
 * Prints the captured frame to STDOUT. The frame will be printed in PGM format.
 *
 * @param dev lepton device structure.
 * @param adjusted Setting this parameter to false will cause RAW sensor data to
 *                 be written to the file.
 *                 Setting this parameter to true will cause a preprocessed image
 *                 (with a stretched dynamic range) to be saved to the file.
 */
void lepton_print_capture(lepton_dev *dev, bool adjusted) {
    /* TODO : complete this function */
}
```

## app.c

The code in `app.c` contains the same functionality as you implemented in lab 1.2, i.e. the *left* joystick is used to control the pan-tilt module. We now wish to merge the lepton application code developed in lab 2.0 to our main program.

Complete `app.c` such that moving the *right* joystick to the *right* past a certain *threshold* triggers a frame capture with the Lepton controller.