Lab 4.0 – Embedded Linux Systems

# Lab 3 – Hybrid Systems (recap)

The goal of lab 3 was to explore how hybrid systems involving both hard and soft components interact together. To this end, you re-implemented the FPGA-only thermal camera acquisition system you had previously developed in lab 2, by replacing the embedded Nios II processor controlling the system by the general-purpose ARM processor.

To simplify the transition and to keep the code changes minimal, you used the ARM processor to run bare-metal code, similarly to how you were using the Nios II processor.

# Lab 4.0 – Embedded Linux Systems

## Bare-metal limitations

Running bare-metal code on the ARM processor is essentially similar to having a high-frequency Nios II processor, but there are many downsides:

1. The ARM processor available on Cyclone V SoC devices is a dual-core CPU. However, the preloader does not wake CPU1 up from reset, so your code is running on only one of the cores. You cannot use the second core unless you write the code needed for waking it up from reset.
2. Interfacing with the SD card is impossible unless you write the code needed for interacting with the controller of the SD card and for handling the filesystem used on your card.
3. Interfacing with the Ethernet port is impossible unless you write the code needed to implement a complete TCP/IP stack.
4. …

As you can see, the phrase “unless you write the code needed for <xyz>” comes up often in the previous list. Actually, the situation is much worse than in the Nios II systems you have built until now. Indeed, whenever you create a software project with the Nios II software toolchain, you see that 2 projects are always created:

* Application project
* Board Support Package (BSP) project

The BSP project contains all the information relative to the system on which the Nios II processor is instantiated, but it also contains code needed for the processor to interact with its environment (standard I/O, file I/O, networking …). This “bridge” code is called the Hardware Abstraction Layer (HAL). All systems implement a HAL in some form in order to ease user programming related to interfacing with hardware devices. Whenever you compile your Nios II application code, the software toolchain also automatically compiles and links its associated HAL into a single binary.

However, when writing bare-metal programs for the HPS, you don’t have access to a HAL unless you explicitly include it yourself[[1]](#footnote-1). Even if we include this HAL, you still haven’t solved the numerous problems listed previously, namely you still have to write all the code needed to access system interfaces, even standard ones that exist on all machines (multi-core CPUs, filesystems, networking stack …).

This situation arises all the time, and hence has an expression coined specifically to describe it. We usually call this re-inventing the wheel.

In this lab, we will explore one way of getting around this issue: by installing an operating system (OS).

## Getting Started

The operating system we will install in this lab is Linux. In order to gain access to more than what the Linux kernel alone provides, we will further customize our system by installing the Ubuntu Core root filesystem.

In lab 3, you saw that getting the HPS up and running is a much more involved process compared to the Nios II processor, even for a simple bare-metal application. As you can expect, the steps needed to get a complete Linux system up and running are even longer…

As for the last lab, we would normally tell you to RTFM, but given that no single document exists that explains how to build such a Linux system from scratch, we have written a step-by-step tutorial that explains how this is done for the Cyclone V SoC-based devices. You can follow the tutorial by reading the [SoC-FPGA Design Guide](http://moodle.epfl.ch/mod/resource/view.php?id=912609).

The tutorial was written for the DE1-SoC board, which is a much larger Cyclone V SoC-based device compared to the DE0-Nano-SoC that we are using for this course. However, the steps needed to get a Linux system running are very similar for both devices, and you can apply what you’ll learn about the DE1-SoC to the DE0-Nano-SoC with only minor changes.

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 that might help you are the following:

* 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 DE1-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 13: Using the Cyclone V – HPS – ARM – Linux

As in lab 3, if you understand how the system is built and how the different components interact together, you will see that installing Linux is not that complicated. All the information you need can be found in the tutorial, but whenever in doubt, don’t forget that we have a [Q&A Forum](http://moodle.epfl.ch/mod/forum/view.php?id=917574) for this course where you can ask questions!

## Your Task

Ok! Let’s get started. The first step is to get you ready for the fun by making your custom SD card with the kernel installed on it. Since we are brave, we are going to make it from scratch. It will help you understand how things actually work. Finding the details of how to do these steps is up to you. Here is the roadmap of what you are supposed to do:

1. Compile the kernel U-Boot. You can find it on the official kernel git[[2]](#footnote-2) or FTP[[3]](#footnote-3) server.
2. Compile the device tree of the DE0-Nano-SoC board.
3. Set up the preloader and the bootloader, i.e. U-Boot.
4. Set up a root file system.
5. Partition and format your SD card.

On the other hand, we will try to (re)view a little bit of background together.

### Background: SD card partitioning

You probably heard of hard disk[[4]](#footnote-4) partitioning at some point. It was probably while reinstalling an OS on your computer where you had to choose between “Use the default partitioning scheme” and “Use a personalized partitioning scheme”. To make it short for those of you who never choose the second option: partitioning a disk is the process by which we divide it into multiple regions. For instance, you might want the first GBs of your disk to be used to store your operating system image and the rest be reserved to maintain a fancy file system (NTFS on Windows or ext4 on Linux) to store the pictures of your last vacations. Do not confuse partitioning and formatting a disk even if they generally happen together. Formatting is the procedure by which you install a file system into a partition.

Do you ever wondered why we partition hard drives? Why don’t we put everything in a single big chunk of memory? There are multiple advantages for partitioning drives. For instance, you might back up just the part of your drive that stores your file. The OS might reserve portion of the disk where it knows it can do whatever it wants, say swapping memory frames, without hurting user files. Another less known fact is that, if you have a lot of space, your file system’s structures might become crazily large slowing down file lookup in which case you may want to split you file systems into many file systems stored on separate smaller partitions.

That’s it for the why! Let’s discuss the how: how is hard disk partitioning generally performed? The classical way in which it is done is via a master boot record, more commonly called an MBR. The MBR is a 512-bytes structure classically stored at the beginning of your partitioned drive. It contains three sections: the first 446 bytes are for bootstrap code, then 4 16-bytes partition entry forms what is referred to as the partition table, finally the last two bytes are used to store 0x55AA, the boot signature that tells the machine that this is indeed an MBR. The bootstrap code might responsible for reading the partition table and jumps to the one containing your OS for instance. If you have a free week-end, we encourage you to try to write one once ;).

### Background: Typical Boot Flow of Cyclone V SoC

In the section above, we discussed the partitioning of hard drive and bootstrap codes but we did not discuss it generally. Let’s review how boot typically happens on the HPS of Cyclone V SoC devices as depicted on Figure 1.

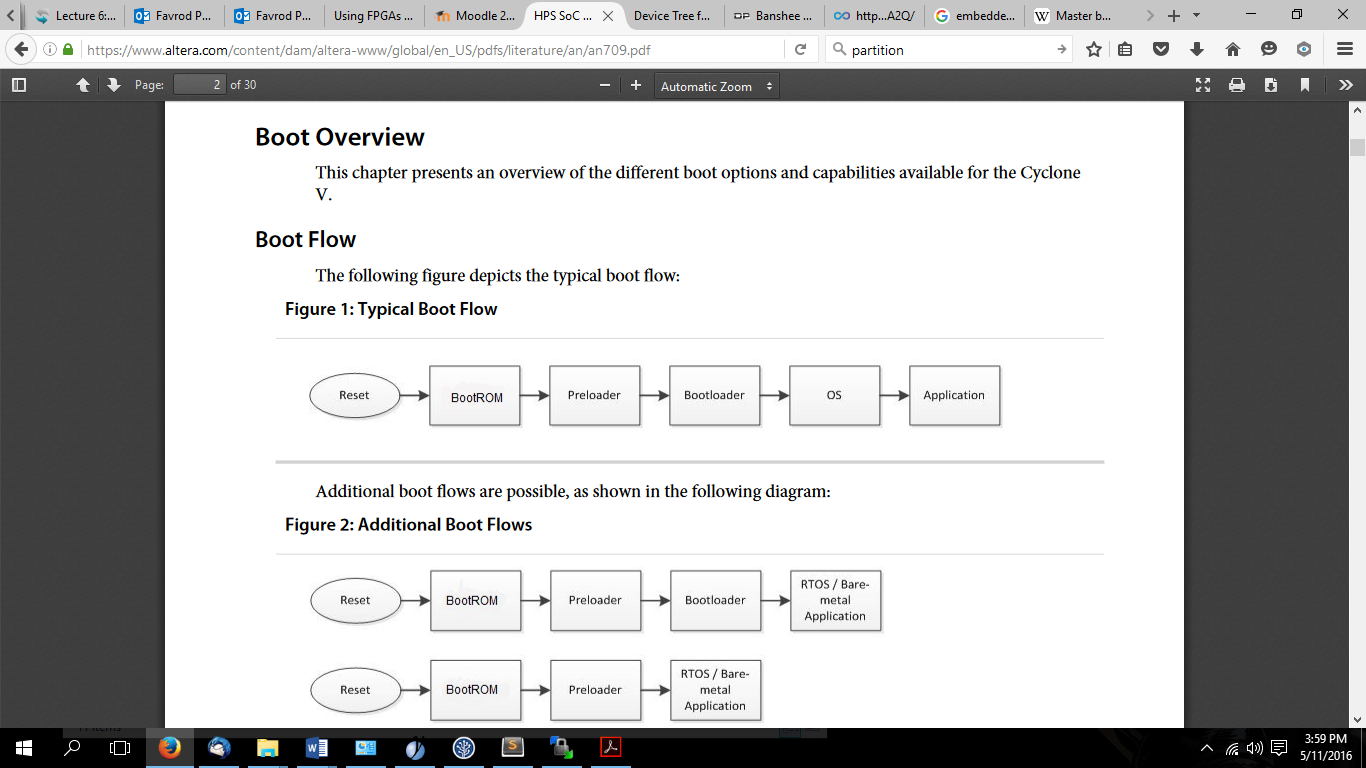


Figure : Cyclone V typical boot flow. source: <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an709.pdf>

At reset, the HPS executes the code stored in the *BootROM* which is responsible to initialize the system a bit, to detect the boot source and load the next boot stage from it. Typically, the next boot stage is the Preloader. The preloader is a bunch of code that is loaded to the on-chip RAM (OCRAM) of the Cyclone V SoC device. It cannot do much because it is limited by the size of this memory and therefore has for main task to configure the SDRAM controller. It can then load the bootloader, U-Boot[[5]](#footnote-5) in our case, which has the plenty of room to invite our Linux friend.

When the BootROM loads the preloader from the SD card, it expects one of the following two partitioning scheme depicted in Figure 2. Actually, as you might have seen, the Raw Mode is not a partitioning scheme since it does not use partitions.

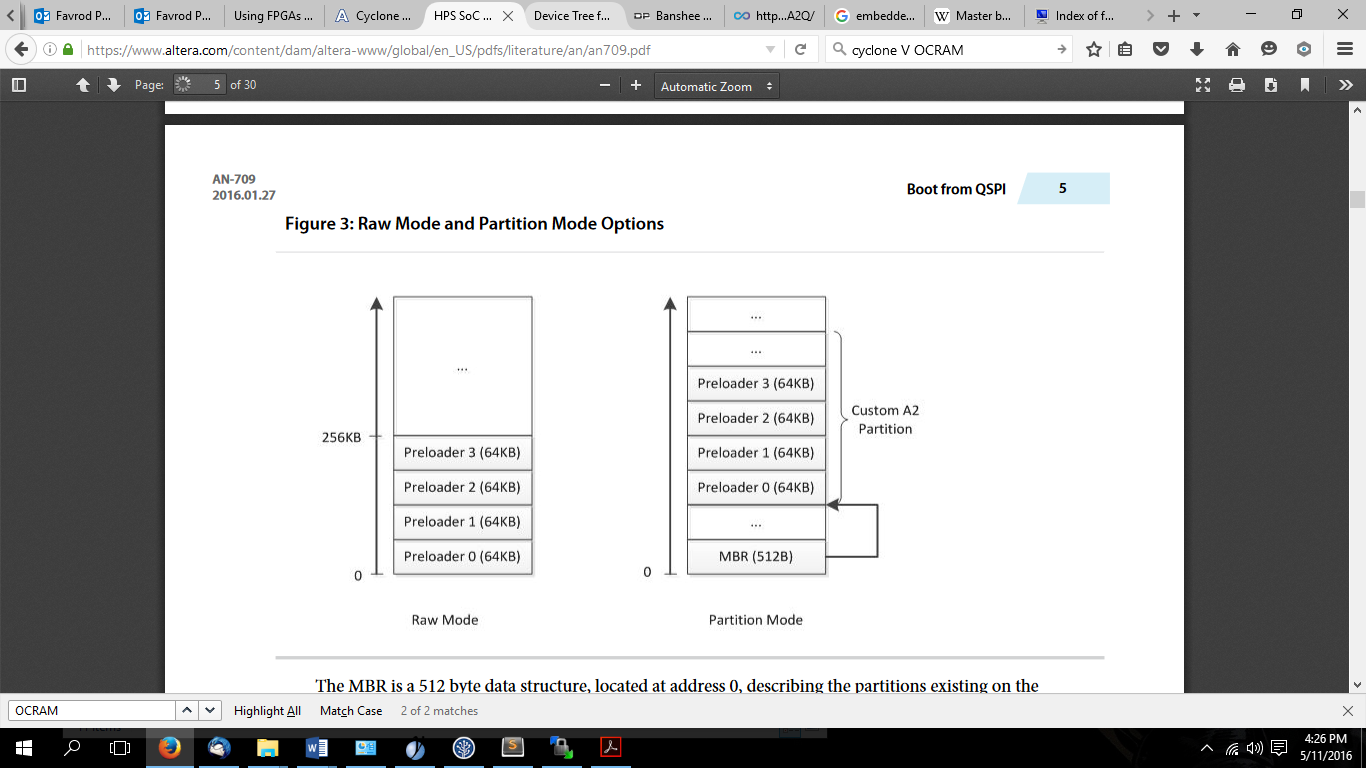


Figure : Supported partitioning scheme. Source: <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an709.pdf>

### Background: the components of an embedded Linux

The Linux kernel can be thought as a packaging of OS services (scheduler, memory manager, and file systems) and a bunch of device driver. At boot time, the kernel must know which of these device drivers need to be loaded. Back in the old days, it was done through platform-specific code that described each board. This is still the way it is done in some architecture, but not for ARM.

Indeed, this lacked scalability: the kernel code became polluted by board-specific code file. Therefore, another approach was taken: the device tree structure (DTS) file. It’s a text file that describes the board configuration. It is then compiled and placed on the primary partition[[6]](#footnote-6) along with the kernel image. This is depicted in Figure 3. The bootloader is responsible for loading it into memory and passing a pointer to it to the Linux kernel. The Linux kernel then parses it and loads the appropriate device drivers. You won’t have to touch the DTS file as the mainline kernel provides one for the Terasic DE0-Nano-SoC.

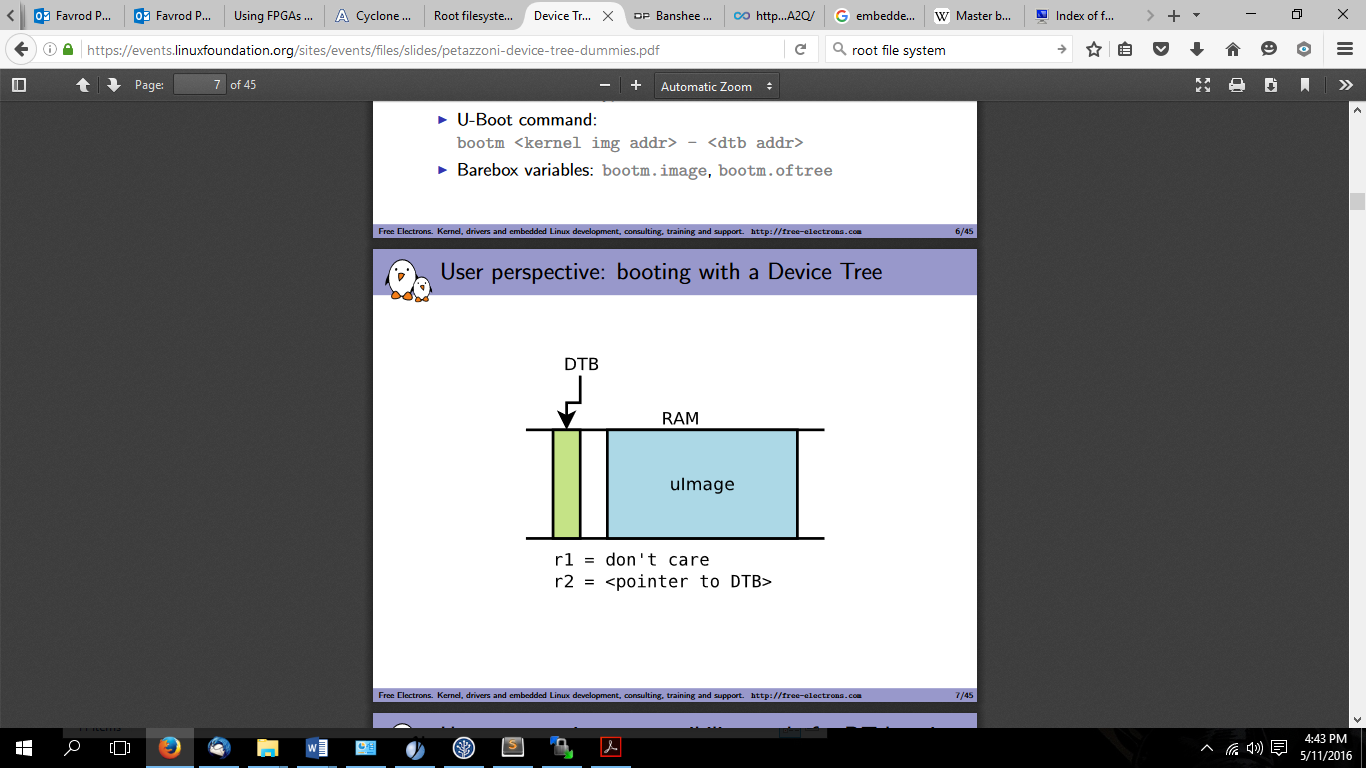


Figure : The DTB is loaded in memory along with the kernel by the bootloader. A pointer to it is passed to the kernel in the r2 register on ARM. Source: <https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf>

Another key component of a Linux system is the root file system. That’s the file system where the root is located. The root is the directory referred to as “/” in your command line.

1. Altera provides a minimal HAL under the name HWLIB. It is mentioned in the [SoC-FPGA Design Guide](http://moodle.epfl.ch/mod/resource/view.php?id=912609) for reference. [↑](#footnote-ref-1)
2. <https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/> [↑](#footnote-ref-2)
3. <ftp://ftp.kernel.org/pub/linux/kernel/v4.x/> [↑](#footnote-ref-3)
4. In this section, we are going to use the term hard drive to denote any kind of secondary storage. It might be an SD card. It just makes the discussion easier. [↑](#footnote-ref-4)
5. The preloader might also be from U-Boot, i.e. U-Boot SPL. [↑](#footnote-ref-5)
6. The primary partition is a FAT32-formatted partition containing the kernel image that is used by the bootloader. [↑](#footnote-ref-6)