Lab 3.1
Embedded Linux Systems

Lab 3.0 – Hybrid Systems (recap)

The goal of lab 3.0 was to explore how hybrid systems involving both *hard* and *soft* components interact together. To this end, you re-implemented the FPGA-only system you had previously developed in labs 1 & 2, but where you replaced 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 3.1 – Embedded Linux Systems

Bare-metal limitations

Running *bare-metal* code on the ARM processor is essentially similar to having a high-frequency (975 MHz!) 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 SD card controller and for handling the filesystems used on your card.
3. Interfacing with the Ethernet port is impossible unless *you write the code* needed to implement a 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

René Beuchat, Philémon Favrod, Sahand Kashani

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]. 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, DMAs, 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 3.0, 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 the amount of online documentation one needs to read in order to know 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.

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. 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

[1] Altera provides a minimal HAL under the name HWLIB. It is mentioned in the SoC-FPGA Design Guide for reference.

  - 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

Normally you have already read the relevant parts of chapters 7, 8, 9, and 11 in lab 3.0, so the only new chapter to read is chapter 13, which is focused on building and programming Linux systems.

As in lab 3.0, 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!

Prerequisites

Pay attention to the fact that you need a *Linux machine* in order to perform the steps mentioned in the tutorial! The specific version of Linux is not really important so long as you have the correct version of the `fdisk` command installed (check the tutorial to be sure of the version).

If you work on your personal machine and are not running a Linux operating system, we provide a virtual machine you can use for all future labs with all the tools installed. If you work on the machines in lab room, the virtual machine is already installed and you can directly launch it by using VirtualBox.

Background

In this section, we present some general systems background that could be of use when reading the SoC-FPGA Design Guide so we are all on the same page.

SD card partitioning

You've probably heard of hard disk[2] partitioning at some point. It was probably while reinstalling an OS on your computer, when you had to choose between "Use the default partitioning scheme" and "Use a personalized partitioning scheme". To summarize the term for those of you who never chose the second option: *partitioning* a disk is the process by which we divide it into multiple regions. For instance, you might want the first few GBs of your disk to be used to store your operating system image and the rest to be reserved to maintain a fancy file system (NTFS on Windows or ext4 on Linux) to store the pictures of your latest vacations.

Do not confuse partitioning and formatting a disk (although they generally happen together). *Formatting* is the procedure by which you install a filesystem onto a partition.

Have you ever wondered why we partition hard disks? Why don't we just put everything in a single big chunk of (non-volatile) memory? There are multiple advantages to partitioning disks. For instance, you might backup

[2] In this section, we are going to use the term *hard disk* to denote any kind of secondary storage. It generally refers to hard drives or SSDs, but in our case it will be an SD card. It just makes the discussion easier.

just the part of your disk that stores your user files. The OS might reserve a portion of the disk where it knows it can do whatever it wants, say swapping memory frames, without damaging user files. Another less known fact is that, if you have a lot of space, your filesystem's structures might become crazily large, slowing down file lookup. In this case you may want to split your filesystems into many filesystems 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-byte 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-byte partition entries form 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 be responsible for reading the partition table and jumping to the one containing your OS for instance. If you have a free week-end, we encourage you to try to write one once ☺.

Typical boot flow of the Cyclone V SoC

Let's review how the HPS typically boots in Cyclone V SoC devices. The boot process is depicted in Figure 1.

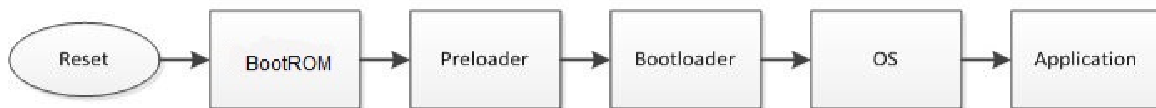

FIGURE 1: CYCLONE V TYPICAL BOOT FLOW.
SOURCE: HTTPS://WWW.ALTERA.COM/CONTENT/DAM/ALTERA-WWW/GLOBAL/EN_US/PDFS/LITERATURE/AN/AN709.PDF

At reset, the HPS executes code stored in the *Boot ROM* which is responsible for initializing the system a bit, detecting the boot source, and loading the next boot stage from it. Typically, the next boot stage is the preloader. The *preloader* is a chunk of code that is loaded into the on-chip RAM (OCRAM) of the Cyclone V SoC's HPS. It cannot do much as it is limited by the size of this memory. Therefore, its main task is to configure the SDRAM controller, and to load the bootloader (U-Boot[3] in our case) which then has plenty of space (1 GB!) to invite our Linux friend.

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

[3] The preloader might also be from U-Boot, i.e. U-Boot SPL.

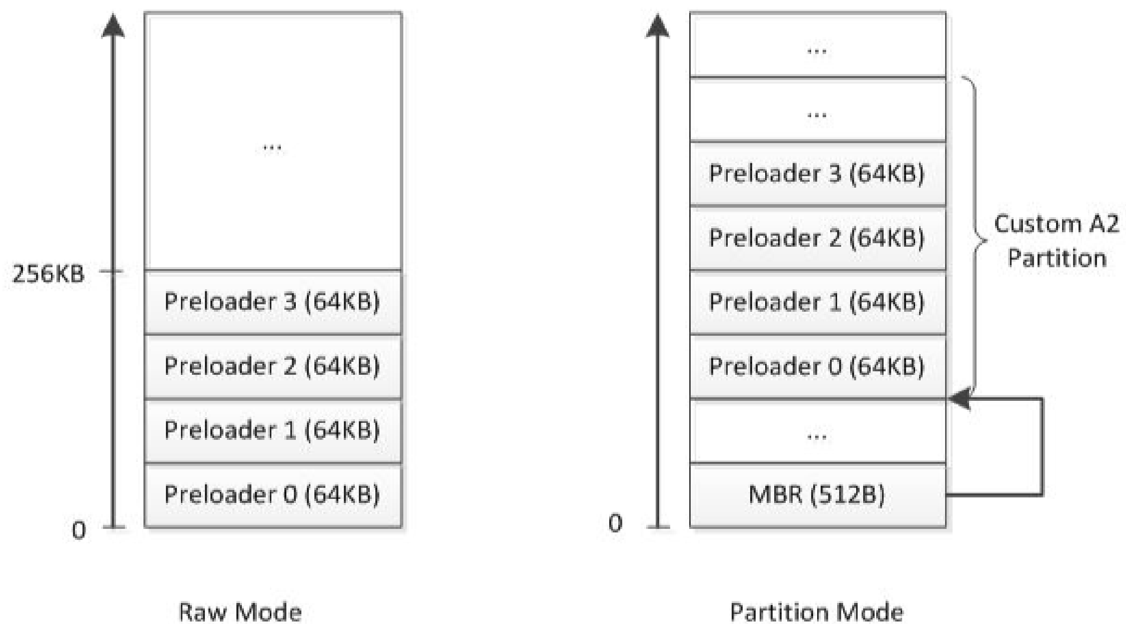

FIGURE 2: SUPPORTED PARTITIONING SCHEMES.
SOURCE: HTTPS://WWW.ALTERA.COM/CONTENT/DAM/ALTERA-WWW/GLOBAL/EN_US/PDFS/LITERATURE/AN/AN709.PDF

Components of an embedded Linux system

The Linux kernel can be thought as a packaging of OS services (scheduler, memory manager, filesystem support) and *a lot* of device drivers. 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 architectures, but not for ARM.

Indeed, this lacked scalability: the kernel became polluted by board-specific code. Therefore, another approach was taken and the *device tree source (DTS)* file was created. It's a text file that describes a board's configuration. It is then compiled into a *device tree binary (DTB)* file and placed on the primary partition[4] along with the kernel image. This is depicted in Figure 3. The bootloader is responsible for loading the DTB 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.

[4] The primary partition is a FAT32-formatted partition containing the kernel image that is used by the bootloader.

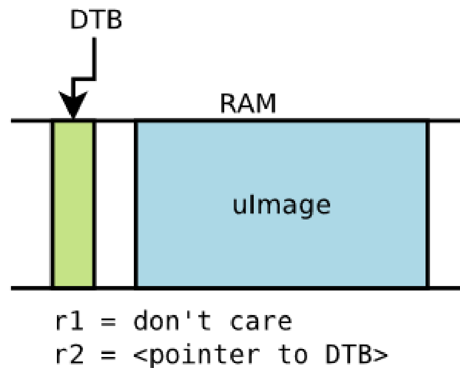

FIGURE 3: THE DTB IS LOADED IN MEMORY ALONG WITH THE KERNEL BY THE BOOTLOADER. A POINTER TO IT IS PASSED TO THE KERNEL IN THE R2 REGISTER ON ARM PROCESSORS.
SOURCE: HTTPS://EVENTS.LINUXFOUNDATION.ORG/SITES/EVENTS/FILES/SLIDES/PETAZZONI-DEVICE-TREE-DUMMIES.PDF

Another key component of a Linux system is the root file system, which is the filesystem where the Linux root is located. The root is the directory referred to as "/" in your command line.

Your Task

Ok! Let's get started. The goal of this lab is to get you ready for the fun that awaits you next week for your mini project. The lab session will be about making your custom SD card with the embedded Linux system installed on it.

Since you are brave, you are going to make this SD card from scratch. It will help you understand how things actually work. Finding the details of how to do these steps is up to you (with some help in the tutorial ☺).

Here is the roadmap of what you are supposed to do:

1. Compile the preloader.
2. Compile the bootloader, i.e. U-Boot.
3. Compile the mainline Linux kernel.
4. Compile the device tree of the DE0-Nano-SoC board.
5. Set up a root file system (Ubuntu Core 14.04).
6. Partition and format your SD card.
7. Boot the Linux system.