INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Linux Based Mobile Operating Systems

DIOGO SÉRGIO ESTEVES CARDOSO

Licenciado

Trabalho de projecto para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutor Manuel Martins Barata
Mestre Pedro Miguel Fernandes Sampaio

Júri:

Presidente: Doutor Fernando Manuel Gomes de Sousa

Vogais: Doutor José Manuel Matos Ribeiro Fonseca
Doutor Manuel Martins Barata

Julho, 2015

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Linux Based Mobile Operating Systems

DIOGO SÉRGIO ESTEVES CARDOSO

Licenciado

Trabalho de projecto para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutor Manuel Martins Barata
Mestre Pedro Miguel Fernandes Sampaio

Júri:

Presidente: Doutor Fernando Manuel Gomes de Sousa

Vogais: Doutor José Manuel Matos Ribeiro Fonseca
Doutor Manuel Martins Barata

Julho, 2015

For Helena and Sérgio,
Tomás and Sofia

Acknowledgements

I would like to thank:

My parents and brother for the continuous support and being the drive force to my live.

Sofia for the patience and understanding throughout this challenging period.

Manuel Barata for all the guidance and patience.

Edmundo Azevedo, Miguel Azevedo and Ana Correia for reviewing this document.

Pedro Sampaio, for being my counselor and college, helping me on each step of the way.

Abstract

In the last fifteen years the mobile industry evolved from the Nokia 3310 that could store a hopping twenty-four phone records to an iPhone that literately can save a lifetime phone history. The mobile industry grew and thrown way most of the proprietary operating systems to converge their efforts in a selected few, such as Android, iOS and Windows Phone.

Mobile operating systems are everywhere: on our phones, watches or cars. They completely reshaped the worldwide society by having instant contact with virtually everyone everywhere. Nowadays we almost can't live without our mobile devices because we use them to work, socialize, study and consume information.

Although being the most used operating systems on the planet, the internal mechanisms, how they run and how to work them is still subject of taboo, mainly because the complexity that these systems have.

This project presents how a modern mobile operating system is organized, how to build it and how to deploy into an embedded device. To accomplish that, the necessary study was made to understand the Linux kernel, how it runs and what it contains. The full fledged operating systems Android and Tizen were dismembered to their core and analysed/studied on how to build and deploy them.

Finally, the project also describes how to deploy on a single device, multiple operating systems and how can one manage them. The proof of concept was built under an ARM board using the latest processor technology.

Keywords: Android, Embedded System, Linux Kernel, Mobile, Operating System, Tizen.

Resumo

Nos últimos quinze anos a industria móvel evolui de um Nokia 3310 que conseguia guardar vinte e quatro registos de chamadas para um iPhone que literalmente consegue salvaguardar uma vida inteira de chamadas. A industria móvel cresceu e descartou na maioria os sistema operativos proprietários, convergindo os seus esforços numa selecção de sistemas como Android, iOS e Windows Phone.

Os sistemas operativos móveis estão em todo o lado, nos nossos telefones, relógios ou carros. Estes reestruturaram completamente a sociedade oferecendo a possibilidade de contactar qualquer pessoa no mundo inteiro a qualquer hora. Hoje em dia praticamente não conseguimos viver sem os nossos dispositivos móveis porque os utilizamos para trabalhar, socializar, estudar e consumir informação.

Apesar de serem os sistemas mais utilizados no mundo, os mecanismos internos, como é que eles executam, ou como trabalhar com eles continua a ser sujeito a taboo, devido à sua complexidade.

Este projecto apresenta como é que um sistema operativo móvel moderno está organizado, como o compilar e como os executar num sistema embebido. Para o fazer foi necessário realizar um estudo para entender o kernel Linux, como é que este corre e o que contém. Os sistemas Android e Tizen foram estudados e compreendidos de forma a entender o seu processo de compilação e execução.

Finalmente, o projecto também descreve como executar num mesmo dispositivo vários sistemas operativos e como os controlar. A prova de conceito foi realizada numa placa de prototipagem ARM, usando um processador com a tecnologia mais recente.

Palavras-chave: Android, Dispositivos Móveis, Kernel Linux, Sistemas Embebidos, Sistema Operativo, Tizen.

Contents

List of Figures

List of Tables

Listings

Acronyms

API Application Program Interface.

app application.

CPU Central Processing Unit.

DRM Direct Rendering Manager.

DTB Device Tree Blob.

GBS Git Build System.

GPS Global Positioning System.

HAL Hardware Abstraction Layer.

IoT Internet of Things.

IP Internet Protocol.

IVI In-Vehicle Infotainment.

MMU Memory Management Unit.

NFC Near field communication.

OAL OEM Adaption Layer.

OS Operating System.

OSS Operating System Switch.

PC Personal Computer.

PDA Personal Digital Assistant.

PIM Personal Information Management.

RAM Random-access Memory.

ramfs RAM FileSystem.

ROM Read-only Memory.

rootfs Root FileSystem.

SCI System Call Interface.

SD Card Secure Digital Card.

SDK Software Development Kit.

SIM Subscriber identity module.

SPL Secondary Program Loader.

TCP Transmission Control Protocol.

URI Uniform Resource Identifier.

USB Universal Serial Bus.

VFS Virtual File System.

WPS Wi-Fi Positioning System.

1

Introduction

This document describes the project performed in the context of the Master's in Computer Science and Computer Engineering at Instituto Superior de Engenharia de Lisboa (ISEL). One of its goals is to study two production mobile Operating System (OS), Android and Tizen, understand their core concepts, how they work and finally how to make them run. The other goal is to find out if it's possible to have multiple systems installed on the same device and manage which of them run.

A **mobile operating system** is designed or adapted to work under mobile devices such as, smartphones or tablets. These systems are built and supported by a miscellanea of companies that make up the **mobile industry**.

Today it's nearly impossible to live without a mobile device, whether a smartphone or tablet. Also, the need to be always online, sharing and consuming information has completely changed the way we look at products, brands and information.

The first touchscreen smartphone, IBM Simon[1], was introduced on a pre-internet era (1993) and the term smartphone was first used to describe the Ericsson GS88[2] phone in 1997. Long before Android or iPhone, companies such as Nokia, Blackberry, Ericsson, Palm and Microsoft were already established businesses on the mobile world, with their Personal Digital Assistant (PDA). Before 2007, Palm OS [3], Symbian [4], Blackberry OS [5] and Windows Mobile [6] were the primary operating systems on the market. Even so, mobile devices were gadgets for a selected few, mainly enterprise workers and some early adopters. The years 2007 and 2008 were game-changing for mobile devices, mainly due to the launch of the first iPhone [7]

and later the first Android [8]. Also in 2008 the first dedicated application stores were introduced by Apple and Google, reaching out to the developer community and creating a new kind of business, so remarkable that in 2016 alone is expected to generate $46 billion dollars in sales [9]. These new players transformed the mobile business completely in a way that by 2012 operating systems like Windows Mobile, Palm OS and Symbian practically didn't exist anymore, Android and iOS completely ruled the mobile devices ecosystem. Samsung developed their own operating system named Bada [10], which never took off, as well as Microsoft's new operating system, Windows Phone, that currently is the third largest mobile operating system with 2.7% of market share [11].

Today we have three major operating systems: the Linux based **Android** that rules the mobile world; the **iOS** used only on Apple products such as iPhone and iPad; and **Windows**. Windows offers three operating systems, Windows Phone, Windows RT, used on phones and ARM tablets respectively, and Windows 10 the aggregation of the last two on one common operating system.

Besides the three main ones, there are a few other operating systems worth mentioning: **Firefox OS**[12], **Ubuntu Touch** [13], **Tizen**[14] and **SailFish OS** [15]. All of these are open-source with different target markets but that never had been able to grab a significant market share.

All these operating systems have a simple purpose being a hardware abstraction, a piece of software that virtualizes devices in order to build portable applications across different hardware configurations, illustrated in Figure 1.1.

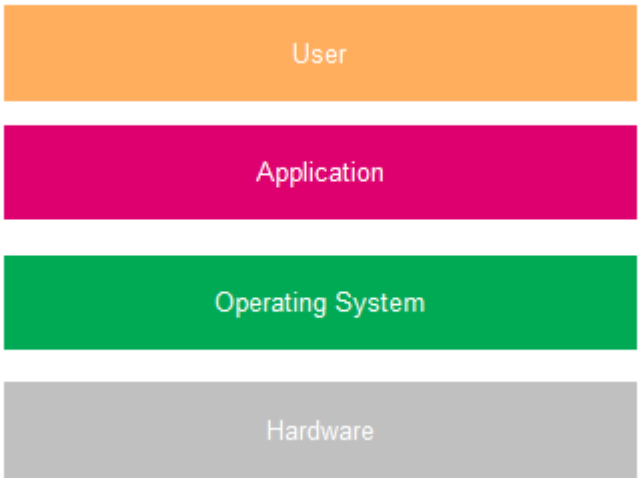

Figure 1.1: System components.

1.1 Scope and Purpose

Of the previously mentioned operating systems only two aren't open-source: iOS and Windows. All the others are fully or partially open-source and are Linux based operating systems.

This project will target Linux based operating systems because they are the majority, are open-source, have large communities and are usually well documented.

The other important aspect to define in an early stage was the target architecture: ARM[16][17] was the obvious choice. Nowadays most mobile devices have one ARM processor; also ARM architecture, processors and families are well supported under the Linux kernel.

Every day the vast majority of the world population uses some kind of operating system, and the mobile ones are becoming more common. Nevertheless, due to their complexity and size, mobile operating systems are pieces of software hard to study and understand, mainly to the overwhelming tooling and pieces that they combine together in order to work.

There are some projects that try to lower the learning curve of operating systems understanding and building, like **yocto**[18], **BuildRoot**[19] or **Linux Target Image Builder**[20], but they end up replacing one set of confusing tools with another.

The purposes of this project are: to understand both the basic and the advanced topics of operating systems; how to approach the sources; how to build them; how to deploy them into a device; and finally, how to deploy two different systems on the same device and execute them on demand.

The target audience of this project is anyone with basic operating systems knowledge with interest in embedded systems. After reading this document they should be able to understand any Linux based mobile operating system, build it and deploy it.

1.2 Methodology

Given that there are a great number of components in any operating system, the main methodology proposed is to separate and test as much as it is possible.

The components that can be executed in virtual environments should be tested in them before deploying to a real device. This is to speed up the iteration between tests, because deploying any component to a device can take a long time (minutes)

and in virtual environments like QEMU[21] or Skyeye[22] just a few seconds. Testing in virtual environments helps mastering the tools and concepts of compiling and running core operating systems components. Finally, after mastering every single possible component separately, it's time to tackle the other tooling and processes that exist when deploying to an actual device.

1.3 Hardware

Operating systems, just like applications or games, have hardware minimum requirements. Working with open-source operating systems offers the flexibility to change the system until it can run on a chosen device, by for instance, removing some functionalities. For this project it was established some minimum hardware requirements, based on the available operating systems, so that any one of them could be executed on the selected hardware, this requirements are identical to the ones from *Android Lollipop*[23]:

- *CPU* 1 GHz, ARM architecture.
- *RAM* 512MB.
- Integrated Display capability.
- Standard connectivity, *SPI*, *I2C*, *UART*.
- Network connectivity, Wi-Fi or Ethernet.
- Persistent storage, via Flash, SD Card, etc.

The first half of this project was prepared using an *i.MX53 Quick Start Board* [24]. Yet, as the project advanced it became evident that the minimum hardware requirements weren't appropriated. The issue was that modern operating systems rely directly on *Hardware Acceleration*[25] to render some or all visual components, that wasn't present in the selected hardware.

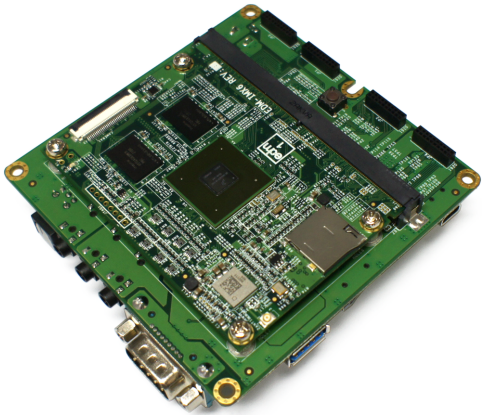

Figure 1.2: Wandboard [1].

The **Wandboard Quad**[26], as Figure 1.2 shows, meets all the defined requirements (old and new), it has an active community and a selection of operating systems already ported to it, the Table 1.1 shows the Wandboard specifications.

Processor	Freescale i.MX6 Quad
Cores	ARM Cortex-A9 Quad core 1.2 GHz
GPU	Vivante GC 2000 + Vivante GC 355 + Vivante GC 320
Memory	2GB DDR3
Display	HDMI
Network	Wi-Fi, LAN, Bluetooth

Table 1.1: Wandboard core specifications.

1.4 Outline

This report is divided in six chapters and three appendixes. The following describes briefly the remaining parts.

Chapter 2 - Mobile Ecosystem Overview: gives an overview of the current mobile world.

Chapter 3 - Linux System: explains how a generic Linux based operating system works.

Chapter 4 - Mobile Operating Systems: describes the concepts behind Linux based operating systems Android and Tizen.

[1] Taken from http://eu.mouser.com/new/Wandboard/wandboard-quad/

Chapter 5 - Boot time OS selection: describes the mechanics behind the selection of multiple operating systems on a single device.

Chapter 6 - Conclusions: expresses the final thoughts after the completion of this project, as well as possible future work.

Appendix A - Environment Setup: explains how to prepare the environment to work with the systems.

Appendix B - Download Sources: describes the necessary steps to download all the system sources.

Appendix C - Deploying to an SD Card: contains the descriptions and steps to deploy the systems to a Secure Digital Card (SD Card).

2

Mobile Ecosystem Overview

Ever since the web boom that the IT industry hasn't seen so much changes in so little time. The mobile ecosystem that almost didn't exist until 2008 surfaced and disrupted everything that was on the market, simple applications made their creators millionaires in a matter of a few months and Apple and Google took their sovereignty on their respective markets.

Mobile Operating Systems existed long before Android and iOS. Their inception is directly related to the hardware specifications and limitations that they used to run like memory, battery and computation power. What we know now as dumb and feature phones were what started it all, evolving after some time to the PDA and finally to the smartphone. Although smartphones seem to rule the ecosystem, dumb and feature phones still represent more than one third of the market[27].

Today we have smartphones whose specs almost match regular computers. The industry evolved in order to deliver faster processors and bigger memories while maintaining or lowering the power consumption. This allowed companies to invest more time and effort on how the operating system should look and feel like and what features should it contain. The change in focus allowed them to create a fast growing ecosystem that changed the way users consume information, work or even socialize.

The two main players on the mobile world are Apple with its iPad and iPhone devices running iOS, and Google with its miscellanea of devices running Android.

Google is on top regarding market share, but it only does so due to the number of

low end devices that run Android. Apple is the runner up, but only works with high end devices. The third player in the mobile world is Microsoft with their Windows Phone/Windows RT combo stretching from low to high end devices.

How the market is segmented devicewise may seem unimportant, but analyzing the application (app) revenue throughput reflects otherwise. Apple payed three times more to its developers than Google[28], representing a seven billion dollars difference. The average user don't choose which operating system is going to purchase by the market share or technical implementation, but instead it chooses it for what apps are in there and the device brand.

In the first quarter of 2015, Samsung ruled the device market[29] with 24.6% share and Apple had 18.3%. Therefore, almost 50% of the device sales are between these two manufacturers. Samsung also represents more than 60% of Androids market[30]. All these numbers may look good to Android but actually it makes it too much dependent on Samsung. Samsung has started it's own venture on Mobile Operating Systems with Tizen[1], a new OS built from scratch that is already running on cameras, televisions, smartphones and wearables.

Wearables are pieces of technology that an user can wear, comprising devices such as smartwatches, headphones or glasses. The first wearable device was made during the Qind Dinasty in the 17th century[31]. Nowadays wearables new (old) technological problems are around the battery life, simply because a wearable doesn't have the same physical space that a smartphone has to integrate with a large battery. The solution was to produce stripped operating systems that work mainly as communicators with another entity to do the processing work, turning most of the wearables on the market into input and output peripherals, with the minimum possible processing actually being done on the devices.

If wearables were the mobile trend of 2014, Internet of Things (IoT) is definitely the trend of 2015. IoT is a network of things, like embedded devices, sensors, smartphones, software, etc. The goal is to have a vast array of data producers working together in order to accomplish a defined purpose. Although not a new concept, IoT is spawning a new set of Operating Systems like Googles Brillo[32] or Microsoft Windows 10 IoT[33] designed to run on high end embedded devices like Wandboard or Raspberry Pi[34].

Operating Systems in general can be divided into two categories: the ones that

[1]Tizen is sponsored by the Linux Foundation and supported by some of the biggest enterprises on the IT world, like Fujitsu, Huawei, Intel, KT, NEC CASIO Mobile Communications, NTT DOCOMO, Orange, Panasonic Mobile Communications, Samsung, SK Telecom, Sprint and Vodafone.

require Memory Management Unit (MMU)[35] and the ones who don't. All the operating systems that were referred until this point require a MMU in order to function, but OSs such as μClinux[36], eCos[37] and FreeRTOS[38] don't. These work at the lowest level of operating systems and are reserved to low end applications on low end devices such as medium range arduinos or arm boards.

Finally there are the low level embedded systems that run without any formal operating system. They are designed from scratch for each purpose, like sensor controlling or mini automatons.

3

Linux System

An operating system is a piece of software that manages all hardware and software from a device. The advanced operating systems run on two different modes, **user mode** and **kernel mode**, the user mode is reserved to user and other low privilege applications. The kernel mode is where all the important components run such as device drivers, process management, etc.

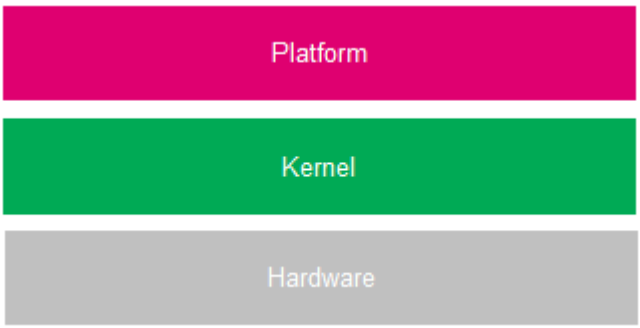

Figure 3.1: Device components.

Considering all the applications that run in user mode as a single application, one can reduce the modern device architecture into three layers, as Figure 3.1 shows: a **platform** component that aggregates all the user data, applications and user interface; a **kernel** that controls all the hardware and manages all the system; and the **hardware** where all the software runs and all data is stored. The Linux based operating systems are merely a Linux kernel with an application component over it, like Android, Tizen or Ubuntu.

3.1 Kernel Components

Linux kernel is composed by a variety of different components, some of the most important ones are shown in the Figure 3.2.

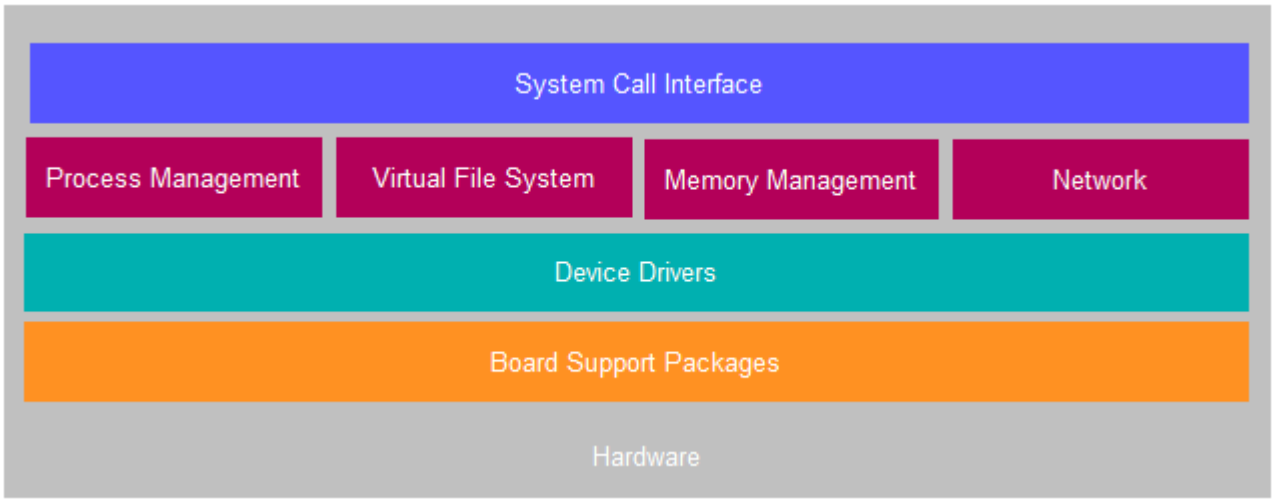

Figure 3.2: Linux major components.

To an application, the kernel can be seen as a service provider, because the kernel knows how to read or write from the disk, how to create processes and threads or if a device was connected. The applications use the **System Call Interface (SCI)** to make these operations and swap between user and kernel mode.

Threads, processes and synchronization are handled by the kernel **Process Management** component; this also handles scheduling and partition of tasks through all of the available Central Processing Unit (CPU).

The **Virtual File System (VFS)** is an API abstraction to access file systems. What Linux has is a common set of operations that were implemented for a variety of known file systems types.

Memory allocation, virtual memory and swapping between memory and hard disk are handled by the **Memory Management**.

Network Stack is a set of layers that handle network requests/responses; it also contains the common network protocols like Internet Protocol (IP) or Transmission Control Protocol (TCP).

The **Device Driver** component is where all the specific hardware interaction is made.

The **Board Support Package** consists on the specific software needed to run the

kernel on a single device. This component can be divided by *architecture*, *micro controller family* and *board*.

3.2 Folder Structure

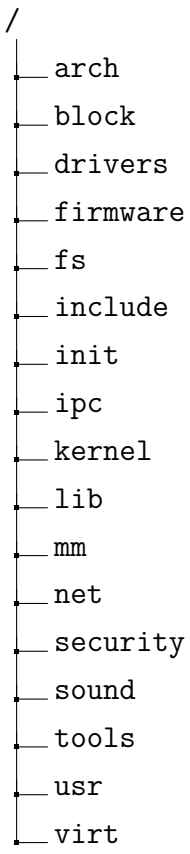

Figure 3.3: Linux root folder structure.

Figure 3.3 represents the source root folder from a regular Linux kernel (refer to Appendix B for downloading the sources). On this figure it is possible to find the location of the previous enumerated components. The **Device Driver** component is on the *drivers* folder; the **VFS** can be found in the *fs*; the **Process Management** on the *kernel*; **Network Stack** on the *net* folder; and finally the **SCI** can be found also on the *kernel* folder, but its architecture dependent source is inside the *arch* folder.

The **Board Support Package** component can be found under the *arch* folder, comprising all non generic source code as well as the device configurations. Figure 3.4 expands partially the *arch* folder.

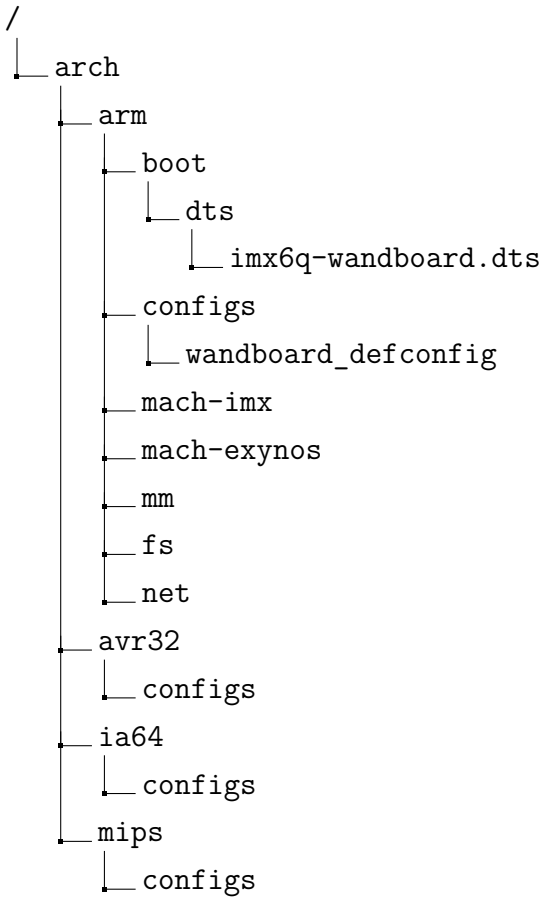

Figure 3.4: arch folder.

The direct *arch* sub-folders are architecture folders: one for each type that Linux supports, with every architecture containing specific sources for each of the Linux components (e.g. VFS or Memory Management).

The devices (or machines) sources are under a *mach-{device}*, these folders only contain non device drivers sources.

The *config* folder is present in all architecture folders and contains the configurations for each supported machine.

3.3 Building

Linux kernel uses a series of hierarquial makefiles to compile its sources. These makefiles use a known set of environment variables to allow external configuration, but only two are mandatory:

ARCH - The target architecture from the device which the kernel is going to be build for.

CROSS_COMPILE - The toolchain[39] that is going to be used to build the kernel.

The *CROSS_COMPILE* variable is used throughout all compilation and linking operations.

The *ARCH* purpose is to select which of the architectures it should search for a config file and, as Figure 3.4 shows, all architectures have a folder on its root called *config* containing all the configurations to the supported devices.

```
$ export ARCH=arm
$ export CROSS_COMPILE=arm-none-eabi-
```

Listing 3.1: Environment configuration for build Linux kernel.

After the environment initialization, shown in Listing 3.1, the next step is preparing the build to the target device. To accomplish that, a make option (Listing 3.2) from the kernel source root makefile is used:

```
$ make {device}_defconfig
$ make wandboard_defconfig
```

Listing 3.2: Configure kernel to a specfic device.

This option copies the device configuration file from its respective folder into the kernel root and does some pre-build configuration.

After all the setup the final step is to simply build the kernel image, as shown in Listing 3.3.

```
$ make
```

Listing 3.3: Build the kernel.

3.3.1 Modules

Linux kernel modules are pieces of software that the kernel can load or unload on demand depending on the current needs of the system. This modules work as a way to expand the kernel functionality without the need to recompile it, so they are standalone libraries and not part of the kernel image itself.

Each new device can add different modules to the kernel (e.g. specific device driver), so when building the kernel is also important to build the modules (Listing 3.4) to include them on the final installation.

```
$ make modules
```

Listing 3.4: Build the kernel modules.

After the modules are built, another command is executed to retrieve the deliverables, as shown in Listing 3.5.

```
$ make modules_install INSTALL_MOD_PATH=${DESIRED_TEMP_PATH_TO_MODULES}
```

Listing 3.5: Retrieve the modules deliverables.

These modules will be later installed on the target device.

3.3.2 Device Tree Blob

Linux kernel has three ways to detect the device hardware. The first consists of having hardcoded on its sources every single hardware description that a target device contains. The second is having some kind of external service (e.g. BIOS) that auto detects the hardware and communicate them with the kernel. The third is the Device Tree Blob (DTB) [40].

The DTB is the hardware description of a device, it consists on a binary file that is passed to the kernel in the boot phase. This allow to generate the kernel for multiple devices (of the same family), where the selection is made depending on the DTB file passed by the bootloader.

For instance, consider Wandboard and Sabre board [41], both have an i.MX6 processor, but their hardware is significantly different. It's possible to generate a single generic Linux kernel for both boards. At the boot phase the bootloader passes a wandboard or sabre DTB to the kernel, only then it knows the hardware that it will run upon.

The dtbs source (dts) can be found on the kernel source directory under *arch/{architecture}/boot/dts* and each device should have an associated dts.

To create the DTB for a target device the command on the Listing 3.6 must be executed.

```
$ make {device}.dts
$ make imx6q-wandboard.dts
```

Listing 3.6: Create the device dtb.

After the compilation is complete, the dtb can be found under the architecture boot folder, *arch/{architecture}/boot/*.

3.3.3 Wrap up

This section addresses the different steps to build a Linux kernel from its sources, that should generate the following items:

- Kernel image.
- Kernel modules.
- The device dtb.

3.4 Boot

When a device is turned on, there are a strict number of phases that it must successfully complete before it can start doing what it was meant to. These phases are called **booting** and their main purpose is to initialize the device hardware and software. The number of phases that the boot process contains may vary depending on what kind of device it is and on what kind of software it contains.

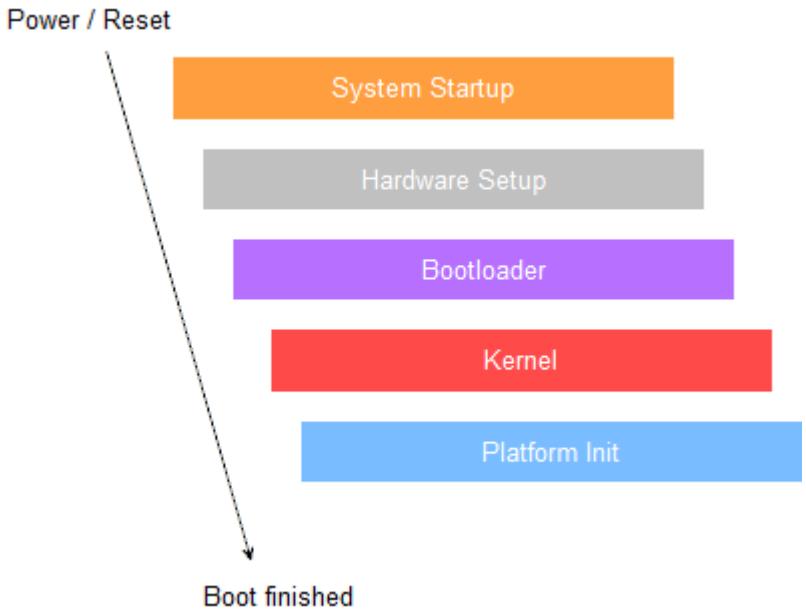

Figure 3.5: Generic Boot Phases.

Considering modern devices that run full-fledged operating systems, generically, the boot process can be divided into five phases:

System Startup - very dependent on the device and architecture, but usually this step copies a chunk of data from a fixed address on the device Read-only Memory (ROM), Flash or SD Card to the device Random-access Memory (RAM) and starts running what it just copied. Usually what is stored on that chunk of data is a bootloader.

Hardware Setup - also known as stage 1 bootloader - has the primary responsibility to configure and turn on specific device features so that the bootloader can execute correctly (e.g. turn on CPU cores). It's also used to load another bootloader when, for instance, the bootloader size is bigger than the one supported by the system startup.

Bootloader - is responsible for all the initial hardware setup and configuration. This includes memory and peripherals setup as well as loading the kernel to memory and executing it.

Kernel - initializes all the components necessary so that the operating system can work correctly like device drivers, services and the user interface.

Platform initialization - is where all the user information, applications and services are initialized and started.

For embedded systems there is a wide set of bootloaders available, RedBoot[42], BareBox[43], U-Boot[44], etc. For this project U-Boot was selected because it is the most popular, has detailed documentation readily available and it has a large user community supporting it.

3.4.1 U-Boot

U-Boot (Das U-Boot) is an universal open-source bootloader used in embedded systems; it supports a wide range of architectures like ARM, AVR32 or MIPS.

Its main goal is to prepare all the hardware to the kernel or application setup. The initialization routine is the following:

Hardware Setup initializes exception and interrupt handlers, configure the CPU and memory (MMU, stack, etc).

Board Setup configures board hardware, load device DTB and start peripherals.

Kernel Boot copies the kernel to ram, prepare the kernel parameters and call its *init* function.

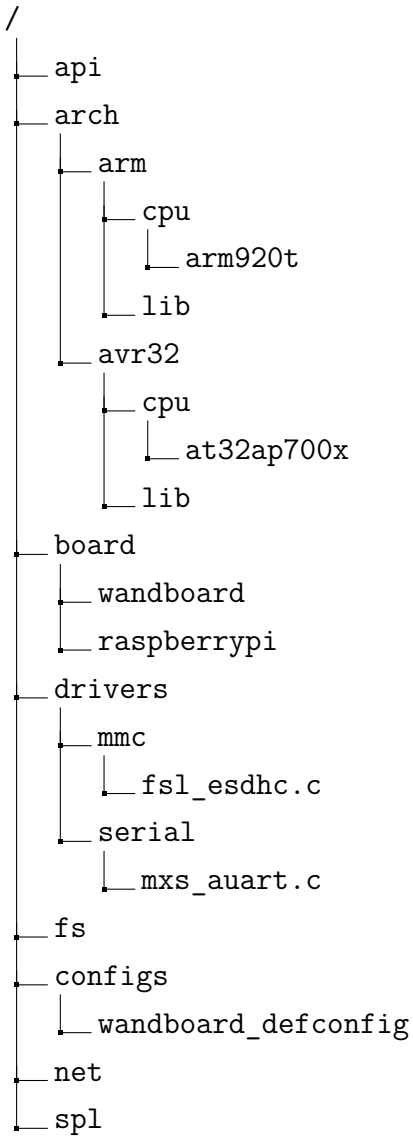

Figure 3.6: U-Boot source tree.

Figure 3.6 illustrates a part of the original U-Boot source tree. The **api** contains the core Application Program Interface (API) to be used by target applications and **fs** and **net** have the necessary source to handle different filesystems and network operations respectively.

The **spl** folder contains the framework for the *Secondary Program Loader (SPL)* a bare bootloader which purpose is to load the U-Boot into memory, the **Hardware Setup** explained on 3.4 Boot.

The **arch** folder contains the sources to support a given CPU for a specific architecture. The architecture generic source is held under the **lib** folder.

Under the **board** folder there is a subfolder for each of the supported boards, containing the board specific source.

The **drivers** folder has one subfolder for each driver type and in there one file for each device supported.

Just as Linux, U-Boot has a hierarchical makefile system which can be configured via environment variables (Listing 3.7), but in this case only the toolchain variable is mandatory. The device selection works the exact same way as Linux, their configurations can be found under the folder **configs**.

```
# Configure toolchain
export CROSS_COMPILE=arm-none-eabi-

# Device selection
make wandboard_defconfig

# Build
make
```

Listing 3.7: Configure and build U-Boot.

The build can generate more than one deliverable, if the target device requires a stage one bootloader (3.4 Boot) the build process will generate two deliverables, the U-Boot and SPL images.

3.5 Platform

The platform is the last component of an operating system. It consists of a collection of software designed to give the end users functionalities. The platform component is materialized as a set of images that can be deployed into a device, in one of which must exist a **Root FileSystem (rootfs)**[45]. The rootfs contains all the files needed to make the system work correctly, it also handles the specific system initialization.

3.6 Deployment

This chapter addresses the different components that a Linux operating system has and which deliverables are associated to them: the **bootloader** image, the **kernel** image, modules, dtbs and the **platform** images. Linux can run from a wide array of options, like hard drive, flash memory, SD Card, etc. Regardless of the physical memory used, the device should have its storage divided in at least two partitions.

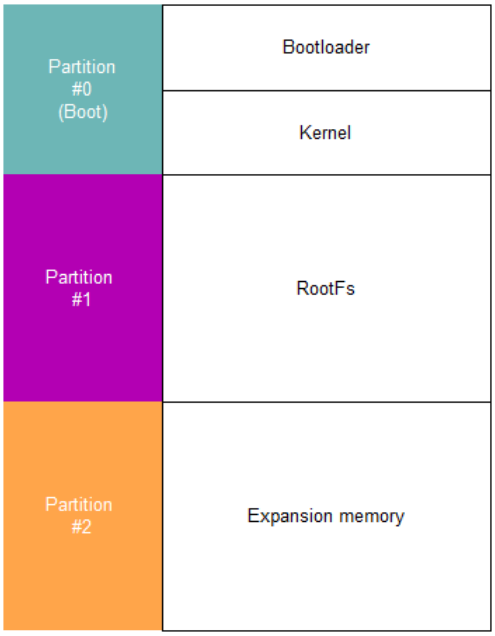

Figure 3.7: Linux device partitions.

Figure 3.7 illustrates a possible configuration of the non volatile memory containing a Linux operating system. The two mandatory partitions are the **boot** and **rootfs**.

The **boot** partition, contains the bootloader, the dtb and the kernel images.

The **rootfs** contains the main platform image and the kernel modules.

Finally, the rest of the memory can be other partitions, for instance, if the platform deliverables have more than one image or just free space to be used by the system to store data.

4

Mobile Operating Systems

Linux operating systems are usually defined as distributions, these consists on an image that contains a bootloader, a kernel and a platform. The platform is usually what names a distribution like Android, Ubuntu or Fedora.

4.1 Tizen

Tizen is a Linux based open-source operating system that supports a wide range of devices, such as smartphones, tablets, televisions, smart cameras, smart watches and In-Vehicle Infotainment (IVI)[46, Chapter 29]. Tizen is sponsored by the Linux Foundation and supported by some of the biggest enterprises on the IT world, like Fujitsu, Huawei, Intel, KT, NEC CASIO Mobile Communications, NTT DOCOMO, Orange, Panasonic Mobile Communications, Samsung, SK Telecom, Sprint and Vodafone. Tizen has a public Software Development Kit (SDK)[47], enabling third party developers to make and deploy applications into the platform. The SDK supports two kind of applications:

Native applications made using unmanaged code, they allow almost a complete control over the platform resources and features.

Web applications built using HTML, CSS and JavaScript, they allow the construction of portable applications between the Tizen platform (as well other mobile platforms). The platform integration is offered via a web API.

4.1.1 Architecture

Tizen architecture can be divided into three layers: the kernel; core; and framework. The framework component, as seen in Figure 4.1, is divided in two to support the different type of applications that the system holds.

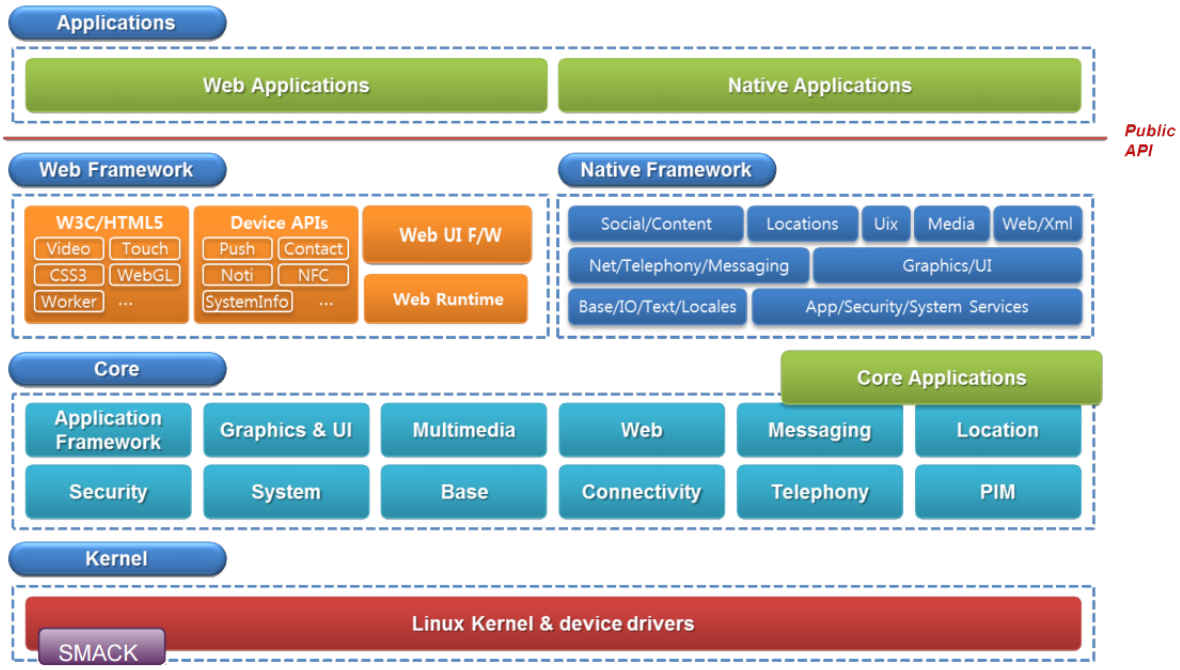

Figure 4.1: Tizen Architecture [1].

The **Web Framework** supports most of the W3C and HTML5 standard like video, audio, geolocalization, web sockets, etc. It also contains web APIs to access internal devices like Bluetooth, NFC, alarms, etc. The concept behind this kind of application is to enable developers to port existing applications or create new ones that run across platforms.

The **Native Framework** contains total access to Tizen APIs and supports third party libraries like glibc, libxml2, OpenGL, OpenAL and OpenMP. It also enables developers to port applications due to its support to many open-source libraries. This framework is specially designed to enable games, real time applications or applications that due to requirements can't be done using Web Framework (e.g. need to use some API that isn't available on web framework).

The **Core** layers contain all the system features. It consists of open-source libraries and some platform specific features:

[1] Adapted from http://www.slideshare.net/rzrfreefr/tizen-contribfosdem20140201.

Application Framework - provides mostly control operations to applications, like the launch of other applications given an Uniform Resource Identifier (URI), events to report current memory state, battery, push notifications or screen orientation changes.

Graphics & UI - supports the graphic component of all the system profiles:

Mobile (native) - consists mainly of a port of Bada OS[10].

IVI & Mobile (web) - based on WebKit[48] and WebKitEFL.

Location - provides information about geolocalization like current position, geocoding or satellite information. This information comes from many sources like Global Positioning System (GPS), Wi-Fi Positioning System (WPS), sensors, etc.

Web - Tizen platform web API to devices and sensors.

Security - responsible for the system access control, certificate verification and other security aspects.

System - an aggregation of system features, devices, sensors and energy control. It also handles hardware related events like Universal Serial Bus (USB), charger, Subscriber identity module (SIM), etc.

Base - contains the support for databases, xml parsing and internationalization.

Connectivity - where all the control for communication devices is at. Wi-Fi, 3G, Bluetooth and Near field communication (NFC) devices are handled in this module.

Personal Information Management (PIM) - contains the access to user local information, like calendar and contacts.

4.1.2 Folder Structure

Tizen source is available to the community via Git repositories (Appendix C for further detail). There is a single repository that contains all the tools, documents and source needed to the build and deploy, the Figure 4.2 displays the folder structure after a clean download of Tizen sources.

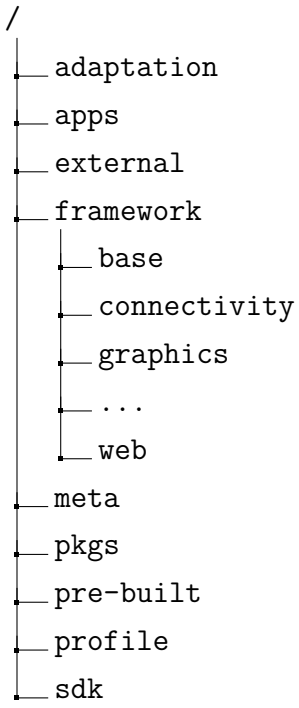

Figure 4.2: Tizen source tree.

The folder **adaptation** contains third-party projects from open-source libraries that were ported or integrated with Tizen, the folder contains projects like *xorg*[49], *OpenGl* and multi-touch handling. It's also in this folder that the sources of the Hardware Abstraction Layer (HAL) are stored.

All the source of the default **apps** (calendar, calculator, email, etc) provided with Tizen can be found on the apps folder.

In **extern**al are external tools and libraries used by the platform.

Framework contains all the Core layer source code, described earlier.

The **meta** folder contains specific build configurations to each of the platform profiles, like mobile, IVI or Personal Computer (PC).

Finally, in the **pkgs** folder, it's possible to find the source from the tool mtools, in **pre-built** there are some pre-built packages for both x86 and ARM architectures, in **profile** there are high level system configurations for specific devices, at last, in **sdk** the source of the public SDK.

4.1.3 Building

Tizen build process is divided in two phases: a **compile** phase that generates all necessary rpm packages[50] and an **image creation** phase that generates all the

necessary deliverables. Appendix A describes how to get the required tools.

4.1.3.1 Compiling

Tizen uses a tool called GBS[51] to build its sources. It has the capability to build from local, remote or a configuration of the last two. This features are selected and configured under a configuration file name *.gbs.conf*, as displayed in Listing 4.1.

```
[general]
tmpdir=/var/tmp/
profile = profile.tizen3.0_mobile
work_dir=.

[repo.tizen_latest]
url = http://download.tizen.org/releases/trunk/mobile/latest/

[repo.tizen_local]
url = ${work_dir}/tizen/sources/

[repo.tizen3.0_arm]
url=${work_dir}/pre-built/toolchain-arm/

[profile.tizen3.0_mobile]
repos=repo.tizen3.0_arm, repo.tizen_local
buildconf=${work_dir}/scm/meta/build-config/build.conf
```

Listing 4.1: Tizen GBS configuration file.

Firstly it starts looking for a profile definition, in this example *tizen3.0_mobile*, then recursively finds out which repositories it should use, and after that it gets which packages should be built by checking the **buildconf** file and starts building.

To compile Tizen the command on Listing 4.2 is used.

```
#gbs build -A {architecture}
gbs build -A arm
```

Listing 4.2: Tizen build command.

The compilation outputs a set of *.rpm* packages to be used on the image creation phase.

4.1.3.2 Image Creation

Just like the compilation, Tizen image creation allows to be configured to use local or remote *rpm* repositories. The image creation is done by a tool named **mic**[52]. Mic uses a kickstarter[53] file as a configuration to produce the system images.

```
repo --name=Tizen-main --baseurl=https://download.tizendev.org/snapshots/trunk/common/@BUILD_ID@/
    repos/tizen-main/armv7l/packages/ --save --ssl_verify=no

%packages

@common
@apps-common
(...)
@target-wandboard
%end

%prepackages
eglibc
systemd
busybox
(...)
tizen-coreutils
%end
(...)
```

Listing 4.3: Kickstart file.

In Listing 4.3 it's possible to observe a repository and some packages configurations. This file can also define the target system partitions, commands to be run on the boot sequence and other in device configuration.

Finnaly, the command presented in Listing 4.4 will generate the system images.

```
gbs createimage --ks-file=wandboard.ks
```

Listing 4.4: Creating a Tizen image.

4.1.4 Deploying

Tizen produces three images when successfully built:

platform.img - The *rootfs* of the system.

data.img - The container of the default applications their supporting libraries.

ums.img - Where the default media content is located.

Tizen has a more complex partition scheme than that presented in Section 3.6 Deployment, although the first two partitions remain the same other two need to be created, as seen on Figure 4.3.

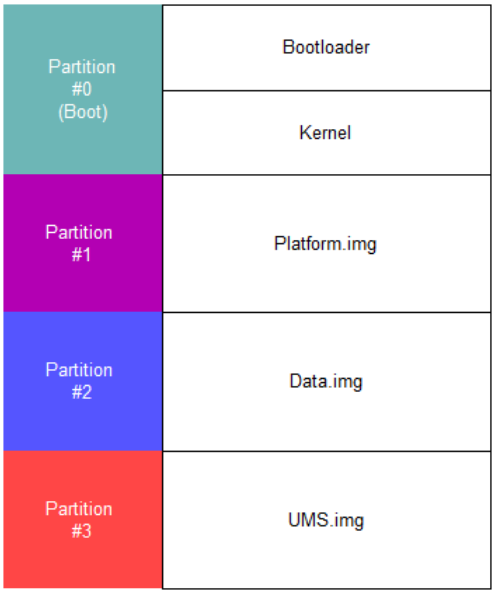

Figure 4.3: Tizen device partitions.

The need to have more partitions is to isolate each type of data into a separate container ensuring that the data isn't corrupted by mistake.

Consider, as an example, that instead of partitions Tizen used an offset scheme, where the *data* and *platform* location where on a known offset from the end of the *platform* image. Besides being harder to access, the system could override data from the other two images.

Finally the partition scheme allows the system to mount each of the images from different places (e.g. one from the SD Card, one from the Flash).

4.2 Android

Android is a Linux based operating system that just like Tizen runs across a wide set of devices like smartphones, smart watches, televisions, etc. Android is currently owned by Google and it's the most used operating system worldwide with about 78 % market share[11]. Android also offers a public SDK and store to let developers publish their own applications into the system.

4.2.1 Architecture

Android architecture is divided into *Application Framework*, *Libraries* and *Runtime* as Figure 4.4 shows.

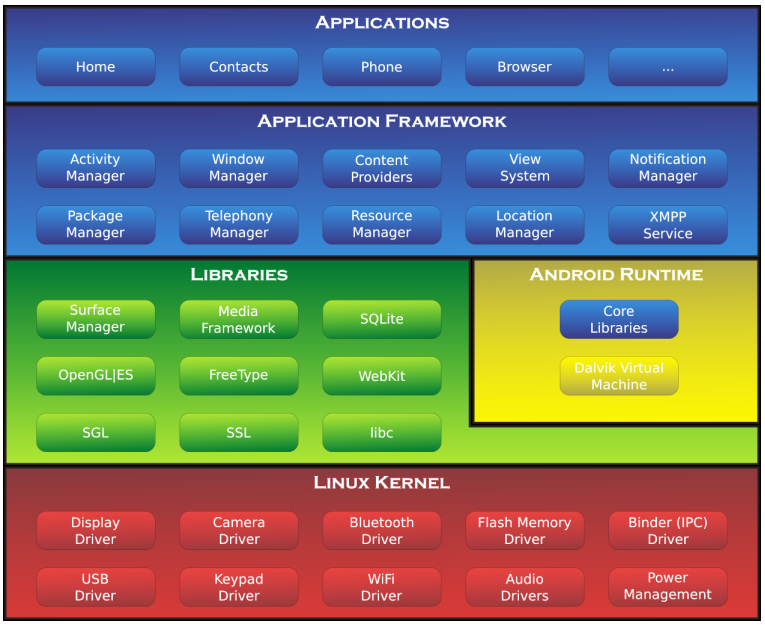

Figure 4.4: Android Architecture [2].

The **Application Framework** is a service layer to Android applications. Its main purpose is to manage applications lifecycle (Activity Manager), its resources (Resource Manager) and offer APIs to let them interact with the system such as, location (Location Manager), telephony (Telephony Manager) or notifications (Notification Manager).

The **Libraries** are the foundation of the Android application system. In this layer it can be found the access to database engines (like SQLLite), advanced graphics drawing (OpenGL), playback of media (Media Framework) or network operations and features. The Libraries layer is a set of java libraries that are in most cases only native wrappers.

Every Android application runs under a Dalvik Virtual Machine instance. Dalvik is a Virtual Machine[54], designed for Android and mobile. The main goal was to create a more efficient virtual machine that could run on low memory scenarios with multiple instances running at once. The standard Java SE functionalities were reimplemented for Dalvik and are available with the **Core Libraries** component.

4.2.2 Folder Structure

Just like Tizen, Android has its sources available via a Git Repository (see Appendix B for further detail). Figure 4.5 illustrates the source folder structure of Android

[2] Adapted from http://mobisoftinfotech.com/resources/blog/android/android-is-open-source/.

5.1 Lollipop.

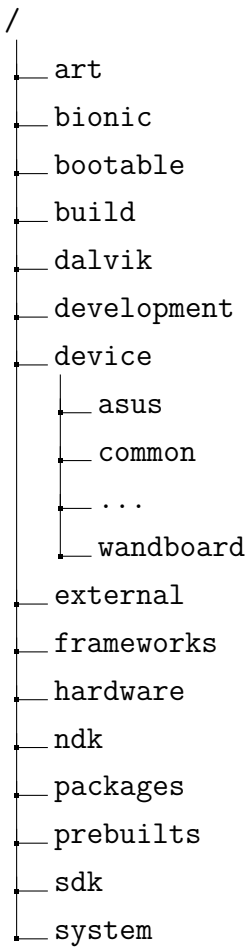

Figure 4.5: Android folder structure.

In the **art** and **dalvik** folders there are the sources for the two virtual machines that Android supports. The **bionic** contains Android version of the libc[55]. The **bootable** contains the Android recovery system and in **build** it can be found the main scripts and configurations to build the system. The **device** folder contains the device specific files and components and the **external** has all the projects imported or adapted to Android. The **frameworks** folder has the sources for the *Libraries* and *Application Framework* layers. The **hardware** contains the Android *HAL* definition and concrete implementation for the supported devices. The **ndk** and **sdk** have the sources for the native and manage development kits that Android has for its applications. The default Android applications source like calendar or contacts can be found on the **packages** folder. On **prebuilts** is a misc of pre-built tools like toolchains or the Android emulator. Finally the **system** folder contains core Android sources, like the initial program that runs on the on the boot phase, networking daemon, etc.

4.2.3 Building

Similarly to the Linux kernel, Android uses a set of makefiles to build the system, but instead of have a recursive build, Android probes all folders and subfolders of the system looking for *Android.mk* files, then it merges every file on a single makefile and only then it starts building.

To build Android there are two scripts that must run first: the **envsetup** and **lunch**. The envsetup partially configures the build system to build Android, it probes the **device** folder for which devices are supported and defines a set of commands, one of them being lunch. Lunch is the one that configure the rest of the build system, with the necessary environment variables for a target device, as shown on Listing 4.5.

```
$ . build/envsetup.sh

$ lunch wandboard-eng

PLATFORM_VERSION=5.0
TARGET_PRODUCT=wandboard
...
TARGET_CPU_VARIANT=cortex-a9
BUILD_ID=1.0.0-alpha-rc1
OUT_DIR=out

$ make
```

Listing 4.5: Configuring and building Android.

After the build is complete it will output three images under the folder *out/target/product/{device}/*, the **system.img** the *rootfs*, the **ramdisk.img** generated from the root, **recovery.img** and if configured a **data.img**, containing apps and default data).

The kernel and bootloader can also be hooked to Android build, if so it will also generate a **boot.img**.

4.2.4 Deploying

Android has a more complex partition scheme than Tizen, in total there are six partitions to accommodate all the system needs, as Figure 4.6 shows.

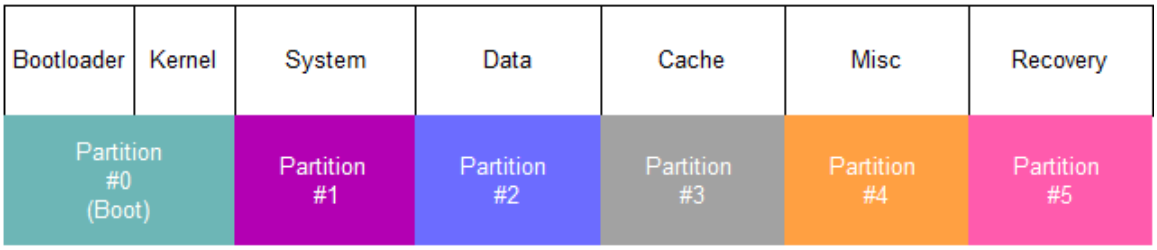

Figure 4.6: Android device partitions.

The *System*, *Data* and *Recovery* partitions are the holders of the images created in the build phase. The *Cache* is where Android will store the most used data and application components and the *Misc* contains important system, carrier and manufacturer configurations. Appendix C presents in more details how to deploy Android.

4.3 Android based Operating Systems

With the rise of Android, many manufactures dropped their house operating systems and started to develop their products directly to Android.

Even not being mandatory, many manufacturers open-sourced they components and device drivers or just published their deliverables. This means that the **foundation** for an operating system, that could run on over 18,000 different devices[56] with more than one billion users[57], is "free" and available for all.

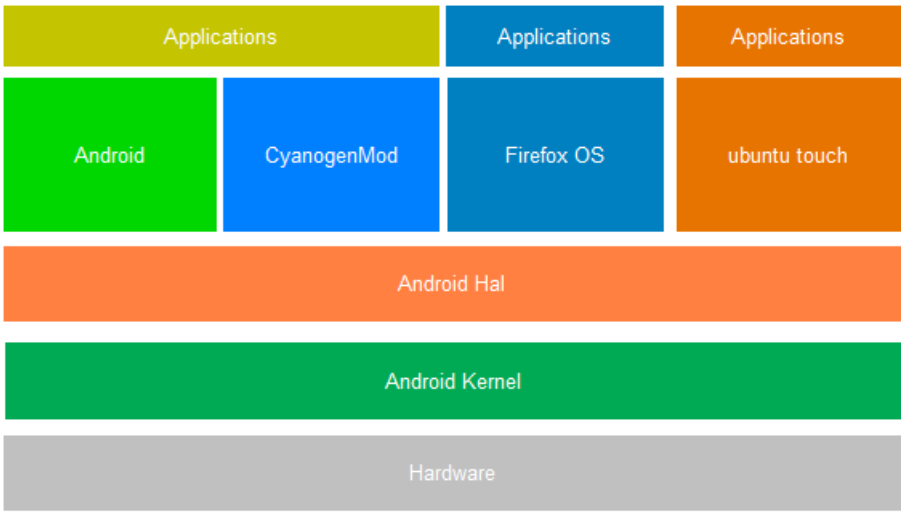

Figure 4.7: Android based OS architecture.

With so many supported devices organizations like CyanogenMod, Mozilla or Canonical started to make their new operating systems on top of Android "device foundation".

Androids **device foundation** handles all the hardware **dependent** components of an operating system, such as the kernel, bootloader or any device drivers. Figure 4.7 shows exactly how these systems are built, they sit on top of Androids hardware, kernel and HAL.

This approach brings many advantages to operating system creators. Firstly, they don't need to worry about any low level engineering, focusing only on the core of the operating system itself such as its features, interface or app model.

Secondly, the target market for these systems is exactly the same as Android, since these run on its foundations.

Thirdly, the use of Androids HAL, removing the requirement to create sensor frameworks or other device abstractions to be used by the rest of the system.

Finally, the tooling, these systems tend to use exactly the same build and deploy mechanisms as Android simplifying any manufacturers and creators work.

The device foundation that Android created is very interesting for any individual or organization that wishes to create their own operating system, besides all low level work being done and available, it enables creators to launch a system that can run on millions of devices.

To conclude, this foundation if used "as is" can allow one to build an entire new operating system just like today we make applications to Linux operating systems. Although it would be something more complex, it's now possible to create an entire new OS without even write a single line of code that runs on *kernel mode*[58][59].

4.4 Porting

Porting any Linux based operating system to a new hardware requires two adaptations. The first is porting the **kernel** to support all hardware and specifications that the target device contains. The other is the own systems adaptations, all the systems studied contain some kind of abstraction layer, abstracting kernel features.

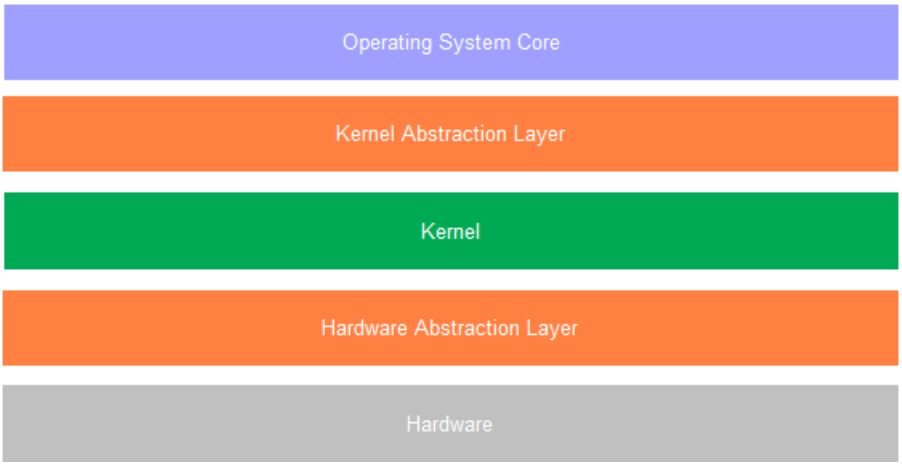

Figure 4.8: Porting overall structure.

The **Kernel Abstraction Layer**, shown in Figure 4.8, exists for two reasons. Firstly to abstract the kernel service layer and respective device drivers, enabling the platforms to rely on their interface to access devices instead of relying on kernel one, turning this layer the single point of change if the kernel ever changes its interface.

Secondly, these operating systems may actively change the kernel to add or improve non baseline features. Having this abstraction layer doesn't mean that the system don't actively use the kernel services directly in core components.

4.4.1 Kernel

Porting Linux kernel isn't an easy task, simply because there are three different types of ports, the **architecture**, **platform** and **board** ports.

The least common is the *architecture* port[60], that consists on adding support for a new architecture to Linux. To accomplish this task, it's needed to write all memory allocation related features, task scheduling, exception, interrupt handling, etc. Basically anything architecture related needs to be redone.

Adding a new *platform* (processor or family)[61] to Linux is more common, it consists mainly in adding devices to Linux, accomplished by adding some structures to some makefiles, configure the platform memory map and interrupts sources, I/O mapping, etc.

A *board* port consists mostly of configuration, since the architecture and platform should be already in the system. Still it can be needed some adaptations, mostly

initialization routines, or, when the desired board contains hardware (peripherals) that aren't supported yet in Linux, a device driver [62] should be built.

Finally for all recent kernels (>2.6.26) any **arm** platform or board port must include a dts describing its hardware.

4.4.2 Tizen

Tizen abstraction layer is called OEM Adaption Layer (OAL) as Figure 4.9 shows.

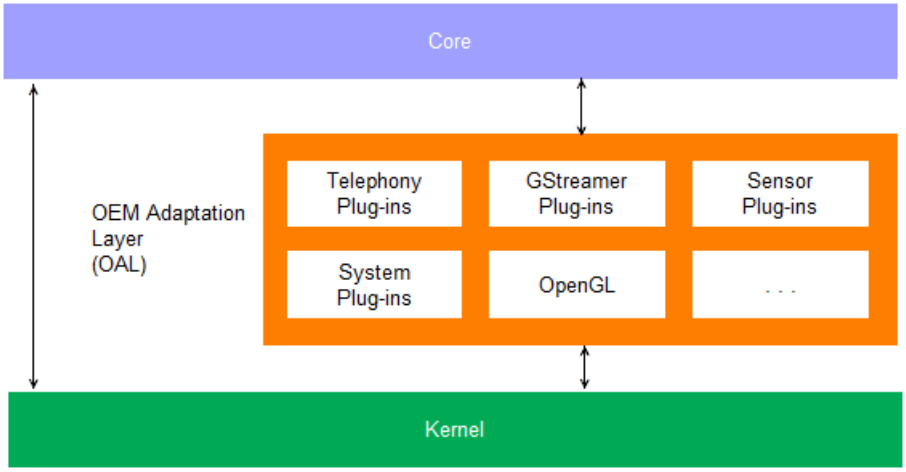

Figure 4.9: Tizen OAL.

The OAL is composed by a known set of contracts that OEMs must implement in order to give the Tizen system the ability to communicate with the target specific hardware.

```
typedef struct {
  int (*OEM_sys_get_display_count) (int *value);
  int (*OEM_sys_get_backlight_min_brightness) (int index, int *value);
  ...
  int (*OEM_sys_set_cpufreq_scaling_min_freq) (int value);
  } OEM_sys_devman_plugin_interface;

const OEM_sys_devman_plugin_interface *OEM_sys_get_devman_plugin_interface();
```

Listing 4.6: Example of OAL contract.

Listing 4.6 shows how the OAL contracts are defined, throughout its different components. For each of them there is a *struct* or an *abstract class* defining the prototypes for the functions that need to be implemented.

After the OAL is fully implemented it must be compiled as **shared libraries**[63] with specific names found on the official documentation[64] and moved to *user/lib* under the **rootfs**.

Tizen automatically loads these libraries and configuration files on the boot phase and uses them throughout the execution.

4.4.3 Android

Unlike Tizen, Android runs under a modified Linux kernel. The major changes [65] that Android did were Wakelocks, Binder, Anonymous Shared Memory, Alarm and Logger. These changes are incremental, meaning that they don't affect any vanilla kernel feature, just complement them.

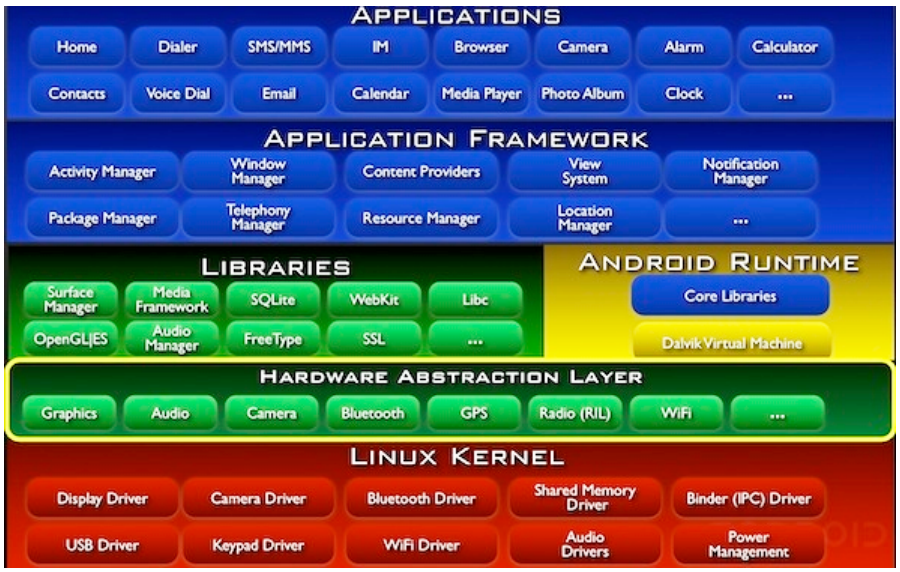

Figure 4.10: Android Architecture with HAL layer [3].

Just like Tizen, Android HAL, seen in Figure 4.10, is a set of known contracts and interfaces that manufacturer should implement. The result is also shared libraries[66] that should be located on the system *rootfs* under *system/lib/hw*.

[3] Adapted from
http://maksim.golivkin.eu/blog/2012/08/26/where-does-android-fragmentation-hide/.

5

Boot time OS selection

There are possibly hundreds of operating systems on the market. Between the proprietary and the open-source ones, the Linux based operating systems represent a large percentage of the market. The mobile phone has a constant presence on our lifes, we literally take them to everywhere we go, but sometimes the features of the mobile operating system isn't enough for a specific task or moment. If we had the ability to change for instance our Android OS for an Ubuntu to do some advanced tasks it would be an improvement of what we have right now.

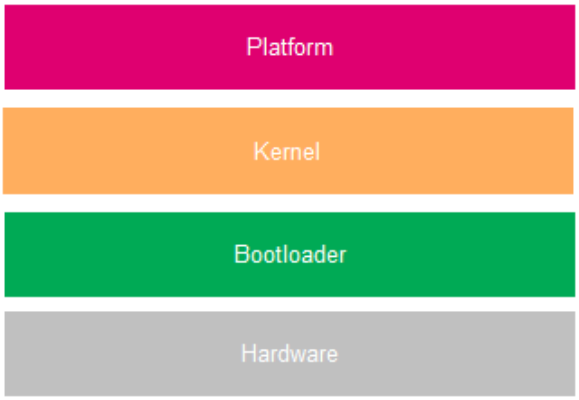

Figure 5.1: Device Components.

To accomplish an **Operating System Switch (OSS)** feature one must look into the operating system architecture, in Figure 5.1, in order to figure out which layer, or layers, one must intervene.

The platform is the result of the rest of the system. When it starts the hardware is

fully initialized, all the core services and data taken care off, excluding it automatically simply because the platform doesn't have enough control of the system until it is given to it. Following the "control" premise, only the kernel or the bootloader have the necessary leverage to accomplish such feature, since the entire boot process and system initialization is handled or started by them.

The kernel is a good candidate to take responsibility of the OSS, it's the kernel that finishes the hardware configuration and controls most of the system. Lastly, the kernel is the platform foundation and starter. But there is an issue, since systems like Android or Tizen can have completely different initialization requirements, as an example, Android needs a RAM FileSystem (ramfs) [67] in order to boot and Tizen doesn't. These per system requirements are usually passed as arguments to the kernel when the bootloader initializes it. So only the bootloader has the necessary leverage to allow such a feature as OSS.

U-Boot, as previously explained, was the used bootloader for this project. To understand how one could implement the OSS functionality it's necessary to know how its runtime works.

5.1 U-Boot Runtime

One of U-Boot features is a debug console where the whole system (bootloader, kernel and platform) can write debug messages. This console work both ways meaning that one can intervene with the system while it's deployed and running, as seen in Listing 5.1.

```
U-Boot 2014.10 (Jul 10 2015 - 10:37:24)
CPU: Freescale i.MX6Q rev1.2 at 792 MHz
Reset cause: POR
Board: Wandboard
I2C: ready
DRAM: 2 GiB
MMC: FSL_SDHC: 0, FSL_SDHC: 1
Display: HDMI (1024x768)
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
WANDBOARD=>
```

Listing 5.1: U-Boot prompt.

While the system is still on control of the bootloader, it has a wide set of commands and options that can be used. One of those commands is the *printenv* that shows the environment variables of the U-Boot.

```
...
baudrate=115200
boot_fdt=try
bootargs=root=/dev/mmcblk2p7 rootwait rw console=ttymxc0,115200
bootargs_base=console=ttymxc0,115200
bootcmd= ...
bootramdisk=boot/uramdisk.img
bootscript=echo Running bootscript from mmc ...; source
bootsys=bootm ${loadaddr}
boottizen=...
fdt_file=boot/imx6q-wandboard.dtb
image=boot/zImage
loadfdt=fatload mmc ${mmcdev}:${mmcpart} ${fdt_addr} ${fdt_file}
loadimage=fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} ${image}
mmcroot=/dev/mmcblk2p9 rootwait rw
...
```

Listing 5.2: U-Boot environment variables.

These environment variables in Listing 5.2, are stored on a file named *uEnv.txt*. They define the locations for the kernel(image), DTB(fdt_file) and other important files such as the Android ramfs(bootramdisk) or the partition where the **rootfs**(mmcroot) is. It is also in this environment that the commands that load the DTB(loadfdt) or the kernel(loadimage) into memory are defined.

The **bootcmd** is the bootloader entry point, where it will eventually execute the other commands to load the data into memory and call the kernel.

Finally, the **bootargs** are the parameters that will be passed to the kernel. Throughout the execution of the bootcmd, this variable is changed depending on the boot sequence. Listing 5.3 shows an example of a final value of the bootargs.

```
Kernel command line: root=/dev/mmcblk2p9 rootwait rw console=ttymxc0,115200 init=/init
```

Listing 5.3: Kernel arguments.

5.2 Dependencies

It's common to look to an operating system as three closed static and dependent layers, the bootloader, kernel and the platform. But as seen on Listing 5.2, the kernel to be loaded and the rootfs are just configuration settings.

Therefore, updating a kernel from one version to another is a simple copy and replace of a file or change U-Boot environment variable. Just like the kernel changing the rootfs to be loaded is updating an environment configuration variable or directly update the *uEnv.txt*.

The only dependencies present on a regular Linux based operating system is the hardware for the bootloader and kernel, and the kernel SCI (and possibly some **/dev**, special devices) to the platform.

5.3 Concretization

When a Linux operating system boots, it's the bootloader that tells the kernel on which partition the platform(*rootfs*) is. Therefore, if we have more than one rootfs on our physical storage in different partitions, it's possible to boot them.

Figure 5.2: Operating System Switch partition scheme.

As an example, consider that the systems can be stored in a single partition. In Figure 5.2, we have a single bootloader and kernel and two operating systems each one in a different partition.

The kernel must be an Android kernel due to its changes but since Tizen works with a vanilla kernel, the add-ons that Android sets on the kernel won't cause any issue when running Tizen.

Therefore if, for instance, we had the bootloader configured to tell the kernel that the rootfs is on the second partition, the kernel would boot Android, but if somehow we could configure the bootloader to change the partition number before the system boots we could also boot Tizen on the same hardware.

This can be more complicated. For instance, Android boot is not directly related to the rootfs parameter that is passed to the kernel, since it needs other files/operations to be loaded/executed before initialing.

Since it's U-Boot that handles all booting operations and configurations, creating specific boot commands for each of the target platforms enables one to load any operating system that is on a physical device.

To achieve this behavior manually, one must define U-Boot commands for each of the target platform, as Listing 5.4 shows.

```
bootcmd=
boot_tizen=run loadimage; run mmcargs; bootz ${kerneladdr} - ${dtb_addr}
boot_droid=run loadramdisk; run loadimage;run mmcargs; bootz ${kerneladdr} ${initrdaddr} ${dtb_addr}
```

Listing 5.4: U-Boot boot commands.

By having the **bootcmd** empty, the bootloader will stop the boot process, giving one the chance to choose which operating system will be loaded. The **loadimage** command reads the kernel image from the physical storage and places it on a known RAM address (*kerneladdr*). The **mmcargs** loads the device DTB and prepares the kernel parameters such as the rootfs location, console, etc. The **loadramdisk**, loads into memory a ramfs [68]. Finally, the **bootz** starts the kernel boot process, it receives as parameter: the location where the kernel was loaded (*kerneladdr*); the ramfs location (*initrdaddr*); and finally, the DTB location (*dtb_addr*).

Tizen doesn't need a ramfs to be booted so this parameter is omitted. By doing so, the kernel is configured to load the system **rootfs** using the *root* parameter (as shown on Listing 5.3) passed to it.

```
$ run boot_tizen
```

Listing 5.5: Booting Tizen.

Unlike Tizen, Android requires a ramfs, so the *boot_droid* command loads the ramfs and passes its location to the kernel. Therefore the kernel ignores the root parameter and uses the ramfs as the **rootfs**.

```
$ run boot_droid
```

Listing 5.6: Booting Android.

Finally, to manually boot any of the systems the commands on the Listing 5.5 or Listing 5.6 must be executed, booting Tizen or Android respectively.

5.4 Automating OSS

Nowadays there is an app for pretty much everything, so why not an app to change the device operating system?

It was explained in this chapter that it's possible to switch from operating system simply by creating and calling U-Boot commands before it executes the kernel. It was also explained that the U-Boot commands are stored in a single file named *uEnv.txt*. So admitting that we have 2 commands, one for booting Android and another for booting Tizen (or any other OS), if we changed the *uEnv.txt* **bootcmd** definition from one command to another we could effectively change the OS that the system would run on the next startup.

The issue is that the boot partition (where the file is) isn't usually mounted (available) on the operating systems so they don't have access to it.

A generic way to change that file, is to create a kernel module or device driver that mounts the partition and changes the text value. This module would be used by an extension of the system HAL and finally called by an user application on the target operating system.

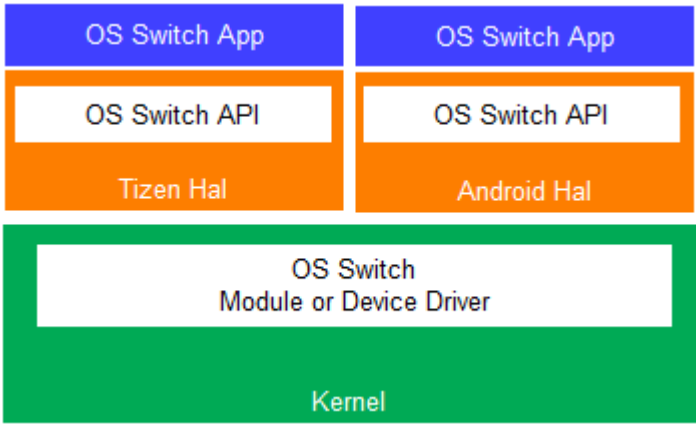

Figure 5.3: Multiple OS switch framework.

Observing Figure 5.3 it's possible to see where development would be required, besides adding to the kernel this feature. It would be necessary to extend each system HAL so that applications could be made to swap system.

Finally, there are two other options to automate this feature: creating a custom rootfs or a bootloader adaptation, where their only goal is to display a list of OSs available and reset the system configuring it with the user selection.

Although appearing simpler creating a custom rootfs is more complex than the direct support into the systems, simply because it would be necessary to create a new phase on the boot sequence, possibly changing how the kernel loads or even the bootloader.

The bootloader option is more interesting. However, since its goal is not to be an

application, its access to hardware is limited to the required by the kernel. The other issue is that the own bootloader boot sequence would be altered.

5.5 Sharing Data between Operating Systems

Right now the OSS although having multiple operating systems running, they don't share any data. As an example, photos, videos or documents from Android can't be seen/used on Tizen. This is because when the systems load, the partitions that they load are hard-coded on the sources, in Android the *fstab*[69] file and in Tizen directly on the *systemd*[70] initialization services. This limitation is only valid for internal memory, if an SD Card is added to the system, both OS can see and work with it. Adapting the fstab and systemd to mount a shared data partition would enable the systems to share data between them.

6

Conclusions

This document describes the architecture and basic functionality of Linux based operating systems. The main goals were: to create an extended jump-start to anyone that wishes to study, build or port systems into new hardware; and investigate if it was possible to have more than one system on a single device and how could one manage them. Given the complexity of working with production operating systems, only the required subjects were approached and explained avoiding the unnecessary complex and overwhelming components and tools.

The initial study showed just how big and complex operating systems are, how they are divided, how they boot, run and finally the enormous tool and skill set necessary to actively and consciously work, modify and deploy this systems to devices.

One would think that being Android and Tizen open-source operating systems, that they would be very well documented and that their internal structure would be clean and organized. Actually that isn't true, since Tizen's documentation is very poor and limited, the best sources of documentation available is being produced during their annual conference[71] and also by some small communities around it.

On the contrary, Android's problem is that there is too much documentation without referencing the target version. Android changes between versions and has a lot of spin-off projects and although it tries to maintain backwards comparability it is not uncommon to find contradicting documents explaining the same concept.

Despite Tizen being in its third version, there isn't an unique window system defined and no explanation why, right now it is possible to have Tizen running with

Wayland[72] and Xorg[49], in general there is a lack of transparency and course.

Tizen is simple and clear enough to be understood without detailed documentation, as it still doesn't have any legacy baggage, so generically it's a very good system to approach the operating system field.

Android fragmentation is a big issue on Android and it reflects on its sources, the system structure is very confusing and not clear at all where its architecture layers are. As an example, to add a new device in Android you need at least to add files in five different folders (device, external, hardware, frameworks, packages), this is mainly because the Android HAL is scattered across all of them.

Configuring Android is not an easy task. For instance, in the Linux kernel, you have available a little graphical application to add or remove kernel features, on Android there isn't anything like it. To customize Android features one must almost perfectly understand how it is build and in some cases change sources to enable or disable features.

Even with all of those issues, Android is still on top of mobile operating systems and that is because of its constant evolution by Google. Just like the Linux, Android growth is so large simply because it's open-source and everyone can work, report or even fix issues.

Android is also breaking another barrier, it's turning into a foundation, an operating system base foundation. Firefox OS and Ubuntu touch can be the first Android based operating systems on the market but, as the time goes by, this may turn out to be a common trend and allow small teams to build and release their own "Android distributions" just as we see today with Fedora or Ubuntu releasing Linux distributions.

The Operating System Switch has a tremendous potential. Changing operating systems enables users to experience new OS or use them for specific tasks or moments. Today mobile phones are as powerful as computers, making them, possible work machines for regular tasks.

Although this project is mobile oriented, is still accurate for most Linux distributions, most of them are actually more simple than Android and Tizen on its core. After having a bootloader and kernel running on a board, it's possible to run literally any Linux distribution (compiled to the target architecture), the only steps needed are to write the distribution rootfs into some non-volatile memory and configure the bootloader where it is.

Each operating system on the market is distributed with some fixed kernel version. These kernels are sometimes updated with systems updates where the device ROM

is replaced. Both Tizen and Android bring a recommended kernel with their sources, so to test if the actual system is dependent on a specific kernel version, the original kernels were scrapped and it was used the mainline Linux kernel[73] (The Android kernel was patched with their extra features).

Both systems run fine with the kernel upgrade. This is because the kernels that are in our devices are the ones that the manufactures worked on. What is important is that the minimum kernel requirements for the system are fulfilled, if the new generation kernels don't change that requisites, then every distribution can virtually work on any kernel.

Throughout the development of this project, many system images were downloaded and run under Wandboard, without a single modification, but on Chapter 4 - Mobile Operating Systems, it was explained that both Android and Tizen require some adaptations in order to fully work.

The images work on the device because although most application functionality is dependent on the system adaptations, the essential of the system is only dependent on the kernel itself, so the system runs without issues but is limited feature-wise.

The other reason is that these images are made to have the minimum possible feature-set usually to test the devices and not the system itself.

Finally, this project started to be a Tizen port. Its initial goal was to understand Tizen architecture and mechanics, and port it to the Wandboard. Throughout the project one issue was found that undermined its completion. While porting the Xorg window system, it was found that its hardware acceleration support is built on top of a Direct Rendering Manager (DRM) [1]. Wandboard uses a Vivante GC graphic engine which has implemented the DRM functionality but unfortunately its source isn't open-source. It was tried to contact the manufacturers of both graphic card and processor to get the sources or the necessary deliverable without any success. Despite that, it was still possible to use Tizen with the Wayland window system, but its performance wasn't acceptable to continue the project. So the project was scrapped but all the previous knowledge acquired studying Tizen helped to quickly understand Android internals and later to implement the OSS.

[1] DRM is an interface between the graphic cards and the Linux system.

6.1 Future Work

Throughout this project it was always avoided turning it into a porting project (although initially there was made a board Linux port to the older hardware). Therefore before selecting the hardware, it was checked if the systems were already ported to the target system, allowing focus on the project purposes. In future developments, making the actual port to a system is something that must be done, understood and documented.

The other component that would be interesting to explore is the automation of the OSS feature. Although all of the conceptual work being complete, none of them was actually applied and developed. This besides enabling the automation of the OSS, would also enable one to explore the different systems HAL and extend them.

Finally, just like the automation of the OSS, the share data feature of the OSS wasn't also developed. Although appearing simple adaptations, changing the systems *fstabs* or the *systemd* services, can have serious repercussions on the rest of the system. It also be worth to explore this capability and add it to the rest of the OSS feature.

Bibliography

[1] Simon, July 2015. URL http://research.microsoft.com/en-us/um/people/bibuxton/buxtoncollection/detail.aspx?id=40.

[2] Ericsson gs88 / penelope, July 2015. URL http://www.gsmmachine.com/ericsson-gs88-penelope-2513.html.

[3] Palm os, July 2015. URL http://www.palmsource.com/palmos/.

[4] Symbian, July 2015. URL http://en.wikipedia.org/wiki/Symbian/.

[5] Blackberry os, July 2015. URL http://us.blackberry.com/software/smartphones/blackberry-10-os-10-2-1.html.

[6] Windows mobile, July 2015. URL https://msdn.microsoft.com/en-us/library/bb847935.aspx.

[7] Apple. iphone, July 2015. URL https://www.apple.com/iphone/.

[8] Google. Android, July 2015. URL http://www.android.com.

[9] Don Reisinger. Mobile app revenue set to soar to 46 billion in 2016, July 2015. URL http://www.cnet.com/news/mobile-app-revenue-set-to-soar-to-46-billion-in-2016.

[10] Samsung. Bada, July 2015. URL http://www.bada.com.

[11] IDC. Smartphone os market share, q1 2015, July 2015. URL http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[12] Mozilla. Firefox os, July 2015. URL https://www.mozilla.org/en-US/firefox/os/2.0/.

[13] Ubuntu. Ubuntu touch, July 2015. URL http://www.ubuntu.com/phone.

[14] Tizen. Tizen website, July 2015. URL https://www.tizen.org/.

[15] Jolla. Sailfish os, July 2015. URL https://sailfishos.org/.

[16] ARM. Cortex-m3, technical reference manual. Technical report, ARM, 2006. URL http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf.

[17] LinuxDevices Archive. Snapshot of the embedded linux market. Technical report, LinuxDevices Archive, 2007. URL http://archive.linuxgizmos.com/snapshot-of-the-embedded-linux-market-april-2007/.

[18] Doug Abbott. *Embedded Linux Development with Yocto Project*. Packt Publishing, 2014. ISBN 1783282339.

[19] Daniel Manchón Vizuete. *Instant Buildroot*. Packt Publishing, 2013. ISBN 1783289457.

[20] Linux target image builder, July 2015. URL http://ltib.org/.

[21] Qemu, July 2015. URL http://wiki.qemu.org/.

[22] Skyeye, July 2015. URL http://skyeye.sourceforge.net/.

[23] Google, 2014. URL https://static.googleusercontent.com/media/source.android.com/en//compatibility/5.0/android-5.0-cdd.pdf. Android 5.0 Compatibility Definition.

[24] freescale. Imx53qsb, July 2015. URL http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=IMX53QSB.

[25] Wikipedia. Hardware acceleration, July 2015. URL https://en.wikipedia.org/wiki/Hardware_acceleration.

[26] Wandboard. Wandboard, July 2015. URL http://www.wandboard.org/.

[27] Mobile phone sales forecast, July 2015. URL http://qz.com/418769/theres-still-plenty-of-money-in-dumb-phones/.

[28] Android beats ios for app downloads, but revenues are still a different story, July 2015. URL http://www.theguardian.com/technology/2015/jan/28/android-ios-app-downloads-revenues-app-annie-google-play-app-store.

[29] idc. Smartphone vendor market share, q1 2015, July 2015. URL http://www.idc.com/prodserv/smartphone-market-share.jsp.

[30] Samsung remains king of the android market with 65% share of all android devices, July 2015. URL http://info.localytics.com/blog/samsung-remains-king-of-the-android-market.

[31] Is this the first wearable computer? 300-year-old chinese abacus ring was used during the qing dynasty to help traders, July 2015. URL http://www.dailymail.co.uk/sciencetech/article-2584437/Is-wearable-computer-300-year-old-Chinese-abacus-ring-used-Qing-Dynasty-help html.

[32] Project brillo, July 2015. URL https://developers.google.com/brillo/.

[33] The internet of your things, July 2015. URL https://dev.windows.com/en-us/iot.

[34] Raspberry pi 2 model b, July 2015. URL https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[35] Colin Walls. Using a memory management unit, July 2015. URL http://www.embedded.com/design/operating-systems/4428102/Using-a-memory-management-unit.

[36] μclinux, July 2015. URL http://www.uclinux.org/index.html.

[37] ecos, July 2015. URL http://ecos.sourceware.org/.

[38] Freertos, July 2015. URL http://www.freertos.org/.

[39] Toolchains, July 2015. URL http://elinux.org/Toolchains.

[40] Doug Abbott. *Linux for Embedded and Real-time Applications*. Newnes, 2012. ISBN 0124159966.

[41] freescale. Sabre board, July 2015. URL http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=RDIMX6SABREBRD.

[42] Redboot, July 2015. URL https://sourceware.org/redboot/.

[43] Barebox, July 2015. URL http://www.barebox.org/.

[44] Das u-boot – the universal boot loader, July 2015. URL http://www.denx.de/wiki/U-Boot/.

[45] Rob Landley. Ramfs rootfs initramfs, July 2015. URL https://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt.

[46] Sio-Iong Ao. *Advances in Electrical Engineering and Computational Science*. Springer, 2010. ISBN 9048184894.

[47] Tizen. Tizen sdk, July 2015. URL https://developer.tizen.org/downloads/tizen-sdk.

[48] Jon Raasch. *Smashing WebKit*. Wiley, 2011. ISBN 1119999138.

[49] X.org foundation, July 2015. URL http://www.x.org/wiki/.

[50] Rpm package manager, July 2015. URL http://www.rpm.org/.

[51] Git build system, July 2015. URL https://source.tizen.org/documentation/reference/git-build-system.

[52] Mic image creator, July 2015. URL https://github.com/01org/mic.

[53] Pykickstart, July 2015. URL https://fedoraproject.org/wiki/Pykickstart.

[54] Gilad Bracha-Alex Buckley Tim Lindholm, Frank Yellin. The java® virtual machine specification. Technical report, Oracle, 2013. URL http://docs.oracle.com/javase/specs/jvms/se7/html/.

[55] The gnu c library (glibc), July 2015. URL http://www.gnu.org/software/libc/.

[56] Android fragmentation visualized, July 2015. URL http://opensignal.com/reports/2014/android-fragmentation/.

[57] Tom Warren. Google touts 1 billion active android users per month, July 2015. URL http://www.theverge.com/2014/6/25/5841924/google-android-users-1-billion-stats.

[58] Firefox os architecture, July 2015. URL https://developer.mozilla.org/en-US/Firefox_OS/Platform/Architecture.

[59] Ubuntu touch architecture, July 2015. URL https://developer.ubuntu.com/en/start/ubuntu-for-/porting-new-device/.

[60] Jake Edge. Porting linux to a new architecture, July 2015. URL https://lwn.net/Articles/597351/.

[61] Tak-Shing Wookey. Porting the linux kernel to a new arm platform. Technical report, Aleph One, 2002. URL http://www.linux-arm.org/pub/LinuxKernel/WebHome/aleph-porting.pdf/.

[62] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly, 2005. ISBN 0596555385.

[63] Shared libraries, July 2015. URL http://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html.

[64] Tizen porting guide, July 2015. URL https://wiki.tizen.org/wiki/Porting_Guide.

[65] Karim Yaghmour. *Embedded Android*. O'Reilly, 2013. ISBN 1449308295.

[66] Google. Android interfaces, July 2015. URL https://source.android.com/devices/.

[67] Gilad Ben-Yossef Philippe Gerum Karim Yaghmour, Jon Masters. *Building Embedded Linux Systems*. O'Reilly, 2008. ISBN 0596529686.

[68] ramfs, July 2015. URL https://wiki.debian.org/ramfs.

[69] fstab, July 2015. URL https://wiki.archlinux.org/index.php/Fstab.

[70] systemd, July 2015. URL https://wiki.archlinux.org/index.php/Systemd.

[71] Tizen developer conference, July 2015. URL https://www.tizen.org/events/tizen-developer-conference/.

[72] Wayland, July 2015. URL https://wiki.archlinux.org/index.php/Wayland.

[73] Kernel.org, July 2015. URL https://www.kernel.org/.

Environment Setup

Each operating system has its development environment, consisting mainly on libraries or tools required to build or deploy the sources. The following steps were made on a 64bit machine running Ubuntu Mint Qiana.

A.1 Toolchains

There are two set of toolchains necessary to download to build the systems. A **bare metal** toolchain, that must be used to build the bootloader, this toolchain strips down the available libraries to its core, removing any filesystem or input/output functionality. The other toolchain is a regular one for the target architecture in this case *arm*. The bare metal toolchain can be downloaded from https://launchpad.net/gcc-arm-embedded or if the host machine is running Ubuntu a regular apt-get command (Listing A.1).

```
$ sudo apt-get instal gcc-arm-none-eabi
```

Listing A.1: Download bare metal toolchain.

The regular toolchain can be found in https://www.linaro.org/downloads/ or installed using the command shown on Listing A.2

```
$ sudo apt-get install gcc-arm-linux-gnueabi
```

Listing A.2: Download regular toolchain.

A.2 Linux

Linux doesn't need any setup besides the toolchains. The graphical configuration tool to configure the kernel needs the Qt Framework installed in the host machine. To install Qt one can download it from their website on https://www.qt.io/download/ or via command, shown in Listing A.3.

```
$ sudo apt-get install qt3d5-dev
```

Listing A.3: Download Qt framework.

A.3 Tizen

Tizen requires the installation of some tools that it uses to compile and deploy the system, those tools are presented on the system public repository under http://download.tizen.org/tools/latest-release/Ubuntu_12.10/. After adding the repository to the source list, the commands on Listing A.4 must be executed.

```
$ sudo apt-get update

$ sudo apt-get install gbs mic bmaptool
```

Listing A.4: Install tools required by Tizen.

A.4 Android

Just like Tizen, Android requires some tools to be installed, for the latest releases the steps are on the Listing A.5.

```
$ sudo apt-get update

$ sudo apt-get install openjdk-7-jdk

$ sudo apt-get install bison g++-multilib git gperf libxml2-utils make python-networkx zlib1g-dev:
    i386 zip
```

Listing A.5: Install tools required by Android.

B

Download Sources

B.1 Linux

The Linux mainline is available on github. To download a simple git clone command is enough, shown in Listing B.1.

```
$ git clone https://github.com/torvalds/linux.git
or
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

Listing B.1: Download Linux kernel.

The other versions of the kernel are available on https://www.kernel.org/ by git or direct download.

B.2 Repo tool

Both Tizen and Android use the repo tool to download its sources. Listing B.2 demonstrates how to download and configure it.

```
// Setup folders
$ mkdir ~/bin
$ PATH=~/bin:$PATH

// Download the tool
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
```

```
// Configure to be executable
$ chmod a+x ~/bin/repo
```

Listing B.2: Download and Initialize Repo.

B.3 Tizen

Although Tizen sources are available, it requires a registration on their website (https://www.tizen.org/user/register). The registration gives access to their Gerrit website (https://review.tizen.org). In the profile (https://review.tizen.org/gerrit/#/settings/http-password) one can find the password needed in order to execute the command in Listing B.3, that initializes the repository.

```
$ repo init -u https://<Gerrit User>:<Gerrit Pass>@review.tizen.org/gerrit/p/scm/manifest -b tizen -
    m mobile.xml
```

Listing B.3: Initialize the repository.

After initializing the user, edit the file .repo/manifests/_remote.xml and swap the fetch property from ssh to http, as shown in Listing B.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
  <remote name="tizen-gerrit"
         fetch="https://<Gerrit User>:<Gerrit Pass>@review.tizen.org/gerrit/p"
         review="https://review.tizen.org/gerrit"/>
</manifest>
```

Listing B.4: _remote.xml file.

Finally, the sync command, shown in Listing B.5, must be executed in order to clone the sources.

```
$ repo sync
```

Listing B.5: Download Tizen sources.

B.4 Android

After repo initialization downloading the sources are just configuration and syncing, as shown in Listing B.6.

```

// Configure to download Lollipop
$ repo init -u https://android.googlesource.com/platform/manifest -b android-5.1.1_r1
```

```
$ repo sync
```

Listing B.6: Configure repo and download the sources.

Deploying to an SD Card

Wandboard uses an i.MX6 micro-controller configured to boot from the SD Card. The SD Card must be arranged in specific partitions and the bootloader (Program Image in Figure C.1) must be on a known location.

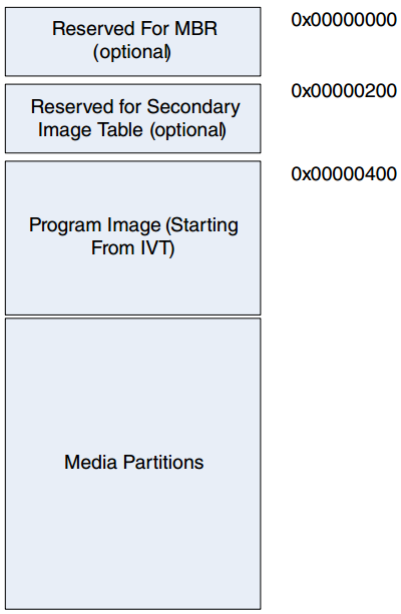

Figure C.1: i.MX 6 basic partition scheme.

The processor will copy the first 70 KB from the address 0x400 of sd card to the ram and start run what it just copied. Since 70KB is too small to fit the u-boot bootloader, this board needs a first stage bootloader in order to work, for u-boot

this means that the SPL module is needed. When SPL takes over the hardware, the location of u-boot and kernel becomes responsibility of the configuration and not the hardware, meaning that SPL knows how to retrieve u-boot and that u-boot knows how to get the kernel image.

C.1 Basic Partitions

The Figure 3.7 illustrates exactly the minimum partitions required to the system. First a boot partition that must be configured as an *fat32* filesystem with the bootloader and kernel, then a partition with any desired filesystem type (ext2, ext3 or ext4 as examples) containing the rootfs. To create the basic partitions the tool fdisk can be used, as shown in Listing C.1

```
card = //sd card device /dev/sdb
BOOT_PARTITION_START = 0
BOOT_PARTITION_SIZE = 8 //mb
ROOTFS_PARTITION_SIZE = 4096
FAT_FILE_SYSTEM = c
EXT4_FILE_SYSTEM = 83



// Create partitions
sfdisk --force -uM ${card} << EOF
${BOOT_PARTITION_START}, ${BOOT_PARTITION_SIZE}, ${FAT_FILE_SYSTEM},
,${ROOTFS_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM}
EOF

// Format partition to fat
mkfs.vfat -n boot ${card}1

// Format partition to ext4
mkfs.ext4 ${card}2 -Lrootfs
```

Listing C.1: Create basic partition.

C.2 Bootloader and Kernel

With the partitions created, the final step is to setup the bootloader and the kernel into the SD Card.

```
$BOOT_PATH = ... //mounted location of the first partition (boot)

// Set the SPL
dd if=SPL of=${card} bs=1k seek=1;
```

```
// Add u-boot
cp u-boot.img $BOOT_PATH/boot/;

// Add the Linux kernel
cp zImage $BOOT_PATH/boot/zImage;

// Add the device dtb
cp imx6q-wandboard.dtb $BOOT_PATH/boot/;
```

Listing C.2: Setup boot partition.

In Listing C.2, the SPL is being set exactly where the i.MX6 processor is going to get the initial program: on the *0x400* (1KB). Since the SPL is configured to check the u-boot on the file system, the u-boot is not placed in any fixed memory location, being instead just a file on the boot partition. The same thing applies to the kernel and dtb: when the SPL loads the u-boot it will try to find the kernel and dtb image in the file system loading them to memory and starting the kernel.

C.3 Tizen

Tizen has two more partitions besides the basic ones, as seen in Figure 4.3, so two extra partitions must be made and formatted before placing the images on the SD Card, as shown in Listing C.3.

```
card = //sd card device /dev/sdb
BOOT_PARTITION_START = 0

BOOT_PARTITION_SIZE = 8 //mb
ROOTFS_PARTITION_SIZE = 4096
UMS_PARTITION_SIZE = 1536
DATA_PARTITION_SIZE = 1536

FAT_FILE_SYSTEM = c
EXT4_FILE_SYSTEM = 83



// Create partitions
sfdisk --force -uM ${card} << EOF
${BOOT_PARTITION_START}, ${BOOT_PARTITION_SIZE}, ${FAT_FILE_SYSTEM},
,${ROOTFS_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM}
,${DATA_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM}
,${UMS_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM}
EOF

// Format partition to fat
mkfs.vfat -n boot ${card}1

// Format partition to ext4
mkfs.ext4 ${card}2 -L rootfs
```

```
mkfs.ext4 ${card}3 -L data
mkfs.ext4 ${card}4 -L ums
```

Listing C.3: Create Tizen partitions.

The final step is copying the images to the SD Card, shown in Listing C.4.

```
dd if=rootfs.img of=${card}2 bs=4098 conv=notrunc
dd if=data.img of=${card}3 bs=4098 conv=notrunc
dd if=user.img of=${card}4 bs=4098 conv=notrunc
```

Listing C.4: Creating a Tizen SD Card.

C.4 Android

Listing C.5 shows that Android shares Tizens base partitions and adds two others.

```
card = //sd card device /dev/sdb
BOOT_PARTITION_START = 0

BOOT_PARTITION_SIZE = 8 //mb
ROOTFS_PARTITION_SIZE = 4096
DATA_PARTITION_SIZE = 1536
CACHE_PARTITION_SIZE = 512
MISC_PARTITION_SIZE = 8
RECOVERY_PARTITION_SIZE = 96

FAT_FILE_SYSTEM = c
EXT4_FILE_SYSTEM = 83



// Create partitions
sfdisk --force -uM ${card} << EOF
${BOOT_PARTITION_START}, ${BOOT_PARTITION_SIZE}, ${FAT_FILE_SYSTEM},
,${ROOTFS_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM},
,${DATA_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM},
,${CACHE_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM},
,${MISC_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM},
,${RECOVERY_PARTITION_SIZE}, ${EXT4_FILE_SYSTEM},
EOF

// Format partition to fat
mkfs.vfat -n boot ${card}1

// Format partition to ext4
mkfs.ext4 ${card}2 -L rootfs
mkfs.ext4 ${card}3 -L data
mkfs.ext4 ${card}4 -L cache
mkfs.ext4 ${card}5 -L misc
mkfs.ext4 ${card}6 -L recovery
```

Listing C.5: Create Android partitions.

Finally, some of Android partitions don't contain default data. These partitions will be filled with information at runtime. Listing C.6 shows how to deploy Android images.

```
dd if=rootfs.img of=${card}2 bs=4098 conv=notrunc
dd if=data.img of=${card}3 bs=4098 conv=notrunc
dd if=recovery.img of=${card}6 bs=4098 conv=notrunc
```

Listing C.6: Creating an Android SD Card.