Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2020

Diogo Duarte

Comparação de Ferramentas de Desenvolvimento de Linux Embutido

Comparison of Embedded Linux Development Tools

Universidade de Aveiro
2020

Departamento de Eletrónica,
Telecomunicações e Informática

Diogo Duarte

Comparação de Ferramentas de Desenvolvimento de Linux Embutido

Comparison of Embedded Linux Development Tools

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor João Manuel de Oliveira e Silva Rodrigues, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro, e coorientação empresarial do Mestre Sérgio Paulo Santos Silva, Diretor Técnico da empresa Globaltronic - Electrónica e Telecomunicações, SA.

o júri / the jury

presidente / president

Prof. Doutor Tomás António Mendes Oliveira e Silva
professor associado da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor António Manuel de Jesus Pereira
professor coordenador c/ agregação do Instituto Politénico de Leiria

Prof. Doutor João Manuel de Oliveira e Silva Rodrigues
professor auxiliar da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Sem dúvida de que todas as pessoas que conheci e com quem interagi contribuiram de alguma forma para a maneira com que eu cheguei a esta etapa da minha vida, e em consequência, a este trabalho. Contudo, gostava de dirigir umas palavras diretas de agradecimento às pessoas que contribuiram mais de perto para este projeto. Ao meu professor e orientador João Rodrigues, pela exelente orientação e paciência, mesmo quando ia perdendo o focus com projetos paralelos. Ao meu co-orientador empresarial Sérgio Silva, antes de mais pela oportunidade de poder desenvolver este projeto tão de perto na empresa, assim como pelo suporte e ajuda mesmo para coisas fora do projeto em si. Ao Salviano Soares, pela disponibilidade para me ajudar sempre que precisei e pela motivação para completar esta dissertação. Ao colaborador da Globaltronic Fábio Silva, o developer de Linux embutido da Globaltronic, pela ajuda e disponibilidade para explicar as dúvidas que tive sobre esta e outras áreas de interesse. A todos os outros colaboradores da Globaltronic, pelas trocas de ideias e disponibilidade para me ensinarem. E por fim, mas sem dúvida não menos importante, à minha família e amigos mais próximos pela compreensão e apoio, mesmo nos dias mais estressantes.

Palavras Chave linux embutidos, sistemas operativos, Armbian, Yocto.

Resumo

O crescente interesse na ligação de pequenos sensores à internet levou ao aparecimento de sistemas operacionais capazes de operar em qualquer hardware assegurando todas as funcionalidades de rede, interface gráfica, servidor, etc. A Globaltronic, uma empresa sedeada em Águeda, tem vindo a desenvolver a plataforma de hardware WiiPiiDo, que se caracteriza por ser um computador embebido altamente especializado para IoT e capaz de assegurar a ligação às redes NB-IoT-LTE Cat NB1 (Narrow Band IoT), permitindo o rápido desenvolvimento de soluções IoT completas para os utilizadores. Por tudo isto, é indispensável criar uma imagem Linux que garanta a fácil utilização de todas as potencialidades da plataforma de hardware. Neste contexto, analisamos o Projecto Yocto, que oferece um sistema de desenvolvimento composto por diversas ferramentas para criação de distribuições Linux para sistemas embutidos, e que tem ganho popularidade numa grande comunidade de utilizadores, especialmente empresas. Contudo, o Yocto não é a única escolha da comunindade de desenvolvedores de sistemas embutidos. De facto, o Armbian, que é uma distribuição baseada em Debian/Ubuntu especializada para sistemas ARM, aparece como uma escolha popular para o desenvolvimento de imagens nestes ambientes. Neste trabalho, iremos ver os passos necessários para testar a plataforma de hardware WiiPiiDo, desde o primeiro arranque até ao desenvolvimento do sistema operativo de suporte, não esquecendo o desenvolvimento dos drivers de suporte aos dispositivos integrados e os testes de desempenho. No final, as ferramentas de desenvolvimento para a criação das imagens vão ser comparadas, desde os resultados obtidos nos testes de performance, ao sistemas de construção em si.

Keywords embedded linux, operating systems, Armbian, Yocto.

Abstract

The increasing interest to connect small sensors to the internet took the development of operating systems able to operate in any hardware ensuring all network, graphical and server functionalities. Globaltronic, a company in Águeda, has developed a hardware platform call WiiPiiDo, that can be described as a embedded computer, power by an ARM SoC, highly specialized for IoT, ensuring connection to the Internet even in harsh conditions using NB-IoT- LTE Cat NB1 (Narrow Band IoT), does ensuring rapid development of complete IoT solutions for end-users. The development of a Linux image that exposes all the potential of the hardware platform is a must and will provide extra value to it. In this context, we take a look at the Yocto Project, which is a building environment that allows the creation of such a operating system, and that is gaining a crescent community of users and specially enterprises. Nevertheless, Yocto is not the only choice for the developer community for embedded platforms, in fact, a distribution like Armbian, a Debian/Ubuntu based Distribution that is specialized for ARM boards, appears as a popular alternative for embedded development in ARM development boards. In this work we will see the steps necessary to test the first boot of the hardware platform until the development of the supporting operating system, passing through the driver development and performance tests. In the end, the used build system will be compared, from the results of the tests performance, to the build system in itself.

Contents

List of Figures

List of Tables

Glossary

ADC	Analog-to-Digital Converter
ARM	Advanced RISC Machine
BIOS	Basic Input/Output System
BL31	Boot Loader Stage 3-1
BSP	Board Support Package
CLI	Command Line Interface
CPU	Central Processing Unit
DTB	Device Tree Blob
dtc	Device Tree Compiler
DTS	Device Tree Source
eMMC	Embedded MultiMediaCard
GPIO	General-Purpose Input/Output
GPS	Global Positioning System
HDMI	High-Definition Multimedia Interface
IoT	Internet of Things
MBR	Master Boot Record
NB-IoT	Narrow Band IoT
OOP	Object Oriented Programming
OS	Operating System
OTG	On-The-Go
PCB	Printed Circuit Board
PWM	Pulse-width Modulation
RAM	Random-Access Memory
RF	Radio Frequency
ROM	Read-Only Memory
RTCC	Real Time Clock and Calendar
SBC	Single Board Computer
SoC	System on a Chip
SPI	Serial Peripheral Interface
SPL	Second Program Loader
TUI	Terminal User Interface
UART	Universal Asynchronous Receiver-Transmitter
U-Boot	The Universal Boot Loader
UEFI	Unified Extensible Firmware Interface
USB	Universal Serial Bus
WTU	WiiPiiDo Test Utility

CHAPTER 1

Introduction

This chapter will give the introduction and explain the purpose and goals, as well as the structure of the whole thesis.

The emergence of IoT devices in the development industry brings new challenges to the design perspective. According to Fortune Business Insights report of 2019, the Global IoT Market was valued at 172 Billion Euros in 2018 and is expected to reach the 1000 Billion Euros by 2026 [1]. The IoT is a multidisciplinary paradigm in which many of the objects that surround us will be networked and connected to the Internet in order to provide new and more efficient services [2]. The diversity of IoT applications and technologies makes it difficult to present a general comprehensive statement for the requirements of IoT in hardware and software [3]. According to the Eclipse IoT Working Group surveys [4], for five years in a row Linux is the most used Operating System (OS) for embedded IoT devices, followed by Windows Embedded and FreeRTOS. Figure 1.1 summarizes the results obtained in the IoT Developer Surveys since 2015.

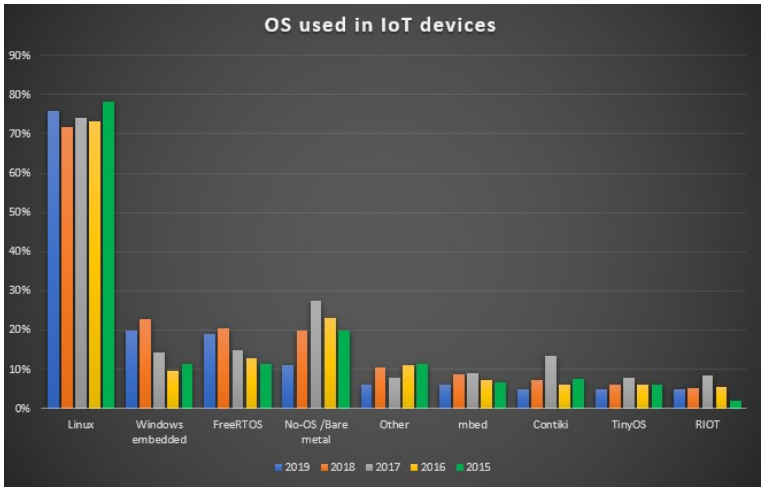

Figure 1.1: OS used in IoT devices

In the 2019 survey, the Linux results are further divided into several distributions. Figure 1.2, from the 2019 survey, shows that Debian and its derivatives like Raspbian and Ubuntu dominate the developer's preferences, followed by CentOS and Yocto.

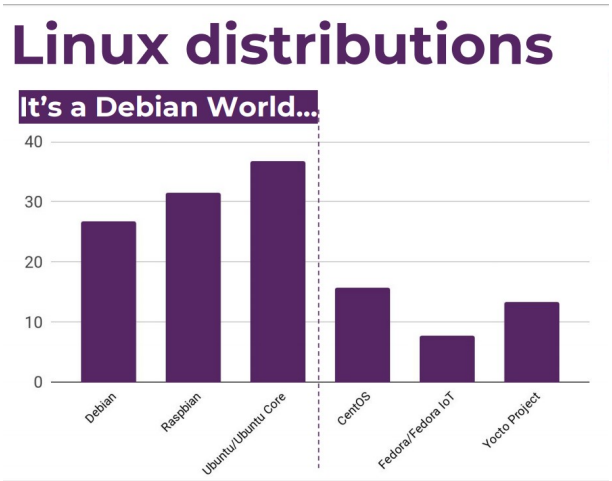

Figure 1.2: Linux Distribution preference [4]

The fast development of new architectures and platforms specially design to face the challenges of IoT and the developers preference in the risen of new IoT products gathering data, tracking usage, monitoring functionalities, automating systems and processes all over the world. Therefore, companies all over the world are investing in this new market by developing new platforms from fully embedded devices to SBC. This huge market is expected to reach 1.5 trillion Euros in 2020, and, according to Coldwell Banker study [5] more than a quarter of all consumers already own a smart-home device, i.e., a device that is connected and allows the automation of functions that once had to be controlled manually. It is, nevertheless, important to clarify the difference between an embedded system and a general purpose computer, since all of these are computational networked devices, but with different purposes.

An embedded system can be defined as a computer hardware system with integrated software that is designed for a specific task, or a small set of specific tasks [6]. Embedded systems are integrated in many devices nowadays, such as smartphones, routers, building tools, house appliances, cars, IoT devices, and many others. Since they have small sets of tasks to perform, embedded systems can be designed to minimize the size, the cost and power consumption, and to improve the performance and reliability of the system. This contrasts with personal computers, which are general-purpose devices, being used to perform a multitude of tasks, such as web browsing, gaming, video editing, etc. An embedded system is characterized by [7]:

- Having a single, or small set of tasks to perform
- Being optimized in terms of size, cost, performance, etc, only to provide the necessary requirements for the tasks to perform
- Often having to operate under real time constraints
- Being able to react to changes

1.1 The WiiPiiDo board

Capitalizing on the evolution of the market from purely embedded systems to SBCs, Globaltronic[1], a company based in Águeda (Aveiro, Portugal) that is specialized in the development of integrated electronics with hardware, firmware, software and prototyping, developed the WiiPiiDo board. This is the target board for the images that were developed in this thesis.

The core components of the board are listed below, with references to Figure 1.3.

- Quad-Core ARM Cortex A53 64-bit System on a Chip (SoC) (18)
- 8GB iNAND Embedded MultiMediaCard (eMMC) flash memory (15)
- 2GB DDR3 Random-Access Memory (RAM) (17)
- Wi-Fi and Bluetooth 4.0 (23)
- U-Blox Max M8Q, Global Positioning System (GPS) (22)
- Quectel BC66, Narrow Band IoT (NB-IoT) (20)
- 4K/30Hz HDMI (6)
- 1 Real Time Clock and Calendar (RTCC) (15)
- 3 Differential Analog-to-Digital Converters (ADCs)
- Raspberry Compatible GPIO Header (11)
- 40 Additional General-Purpose Input/Output (GPIO) Ports (12), (13)
- 1 MicroSD Card Slot (8)
- 4 USB Ports (2), (3)
- 1 USB-On-The-Go (OTG) Port (4)
- Gigabit LAN (1)
- Audio Jack Connector (5)
- MIPI Display (7)

Table 1.1 shows the comparison of the main features between the WiiPiiDo board, and 3 other popular SBCs on the market, the Raspberry Pi 4 Model B[2], the BeagleBone Black[3] and the Pine A64+[4].

From this table, we can highlight some the main advantages that the WiiPiiDo board has against the other boards as the internal storage, NB-IoT, ADCs and RTCC, as well as other non-component related features such as having an high working temperature threshold, from -20 to 70 ºC. This way, this board is more recommended for industrial applications, instead of home use, having more communication options than most other SBCs.

[1] Globaltronic Portugal – `https://www.globaltronic.pt/en/`
[2] Raspberry Pi 4 Model B – `https://www.raspberrypi.org/products/raspberry-pi-4-model-b/`
[3] BeagleBone Black – `https://beagleboard.org/black/`
[4] Pine A64+ – `https://www.pine64.org/devices/single-board-computers/pine-a64/`

	WiiPiiDo	**Raspberry Pi 4B**	**BeagleBone Black**	**Pine A64+**
SoC	Allwinner A64, 1.2GHz 4-Core	Broadcom BCM2711, 1.5 GHz 4-Core	TI AM3359AZCZ100, 1GHz	Allwinner A64, 1.2GHz 4-Core
RAM	2 GB DDR3	1,2,4 GB DDR4	512 MB DDR3	1,2 GB
Flash	4 GB	–	4 GB	–
Ethernet	Yes, Gigabit	Yes, Gigabit	Yes, 10/100M	Yes, Gigabit
Wi-Fi	Yes 802.11 a/b/g/n single-band	Yes 802.11 b/g/n/ac dual-band	–	–
NB-IoT	Yes	–	–	–
Bluetooth	Yes 4.0	Yes 5.0	–	–
RTCC	Yes	–	–	–
ADC	Yes	–	–	–
GPS	Yes	–	–	–

Table 1.1: SBCs Comparison

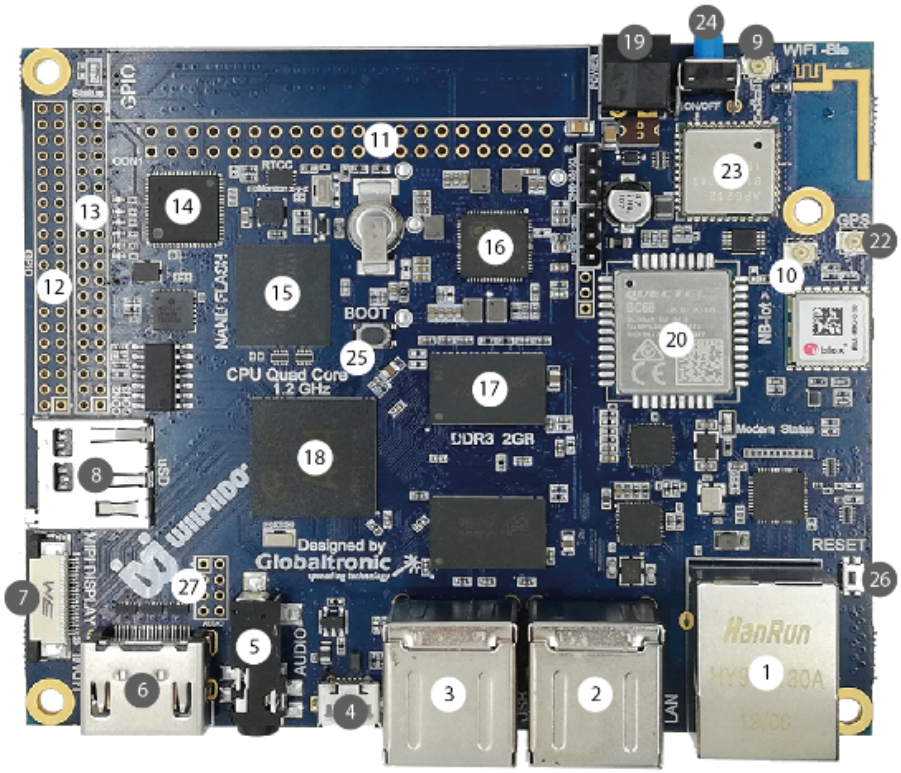

Figure 1.3: The WiiPiiDo Board Components [8].

1.2 Purpose and goals

The purpose of this work was to study and compare multiple embedded Linux building environments, highlighting their main differences and characteristics. With this research we selected the most appropriate environments to be used to generate Linux images that fully support the WiiPiiDo board. The chosen environments were selected to be versatile and allow easy maintainability by the manufacturer, but also to be able to accommodate multiple client profiles.

1.3 Thesis structure

In Chapter 1 the motivations for the development of this thesis are presented, as well as its objectives. Chapter 2 starts with a description of the Linux subsystems and functionality that generally require customization for deployment on an embedded system. Then, several currently existing tools for developing Linux distributions for embedded systems are presented and compared. In Chapter 3 we present an initial development of test images that were developed to do a quick first validation of the first prototypes of the WiiPiiDo board. In the last sections of this chapter, we also demonstrate the implementation of device drivers in the board. In Chapter 4 we describe the implementation of two embedded Linux images developed using the Armbian and Yocto building environments, respectively. In Chapter 5 the built images are evaluated. In a first scenario, the different components from the board are validated in the different built images, using a test utility developed for this purpose. In

a second scenario, the memory usage, CPU performance, as well as some internal metrics are evaluated and compared between different images and boards. Finally, in Chapter 6, the results of this thesis are summarized, and some of the possible future work is discussed.

CHAPTER 2

Linux Development Tools

This chapter is intended to situate the reader in the context of the project, giving brief notions of the basic information needed for the rest of the thesis, as well as present some building environments that were studied.

In the development that will be done further down the line, the changes that will be done to the configurations to be able to support the new board will be focused in the following:

- Bootloader
- Kernel
- Device Tree Source (DTS)
- Device Drivers

2.1 Bootloader

After a device is powered, the first process to run is the boot sequence. In a common desktop computer, this process is divided in multiple parts [9]:

The Hardware Boot The Hardware Boot consists of running a program directly from Read-Only Memory (ROM), the Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) depending on the machine, that will do some basic self-testing and read further parameters from non-volatile memory. One of these parameters will then tell the computer where the boot device is located, and load the OS loader from a fixed address in this device.

The OS loader At this stage of the boot sequence, the computer is trying to find the kernel in a device and load it. In computers, this loader is located in the first sector of the boot device, the Master Boot Record (MBR). Due to limitation in size and complexity of the bootloader, this process is usually divided in two parts:

1. The First stage loader, which is initialized in the MBR
2. The Second stage loader, that is located in a device with a larger capacity, and contains the full featured loader

In Linux, the most typical OS loader are LILO and GRUB.

The Kernel Initilization It is at this point in the boot sequence that the multiple components that exist in the computer are initialized.

Other At this stage, there are a couple more steps the root user-space process and the boot script, respectively. But as these are not important in this description, they will not be presented further.

In embedded Linux, a similar boot sequence is performed, however, instead of separating the first two steps in multiple programs, only a single program will perform those tasks [10]. This program is the bootloader, and most devices use U-Boot. U-Boot is an open source bootloader that can be built to work on a multitude of architectures, being popular in embedded Linux devices.

2.2 Kernel

The Kernel is a program that is loaded to RAM at the boot sequence, that contain critical functions required for the OS to work correctly [11, Chapter 1.4]. This functions consist in managing system resources, such as RAM, and processor time, as well as to manage and interact with hardware components.

2.3 Device tree

When working with desktop and server computers, as they are widely used, the firmware interfaces for this machines are standardized to make it easier for OS developers to integrate hardware their OS. This however does not happen in embedded systems and embedded Linux. As systems vary widely, and software and firmware is customized for each SBC, there is no pressure in the market to standardized the firmware interfaces [12]. This way, to make this easier on SBC, we make use of Device Trees in embedded Linux.

The Device Tree is a software data structure that is used to describe and configure the hardware in a system, allowing the kernel to remain the same, by separating hardware specific details from the kernel [13]. Prior to the adoption of the Device Tree in Linux, specific modifications required to configure the hardware had to be applied to the kernel source code directly. Device tree configurations are read during the booting process, in the kernel phase described in the previous section.

The device tree is developed in a human-readable data structure. These structures are stored as files, where `.dtsi` files, generally contain SoC-level definitions, and `.dts`, which source the respective `.dtsi` file, add board-level definitions. These are compiled using the Device Tree Compiler (dtc), resulting in Device Tree Blob `.dtb` files, which are the binaries objects read by the SoC in the boot sequence. Code 1 shows a template of a DTS file.

There also exists the concept of Device Tree Overlays. These are partial Device Tree that can be used to complement or overwrite the base Device Tree without needing to recompile the Kernel. The Device Tree Overlays can be developed, compiled and activated at runtime, inside the SBC, requiring a reboot to be loaded.

```
/dts-v1/;

/{
    compatible = "<manufacturer>,<model>";

    <node name>[@device address]{
        compatible = "<manufacturer>,<model>";
        reg = <start address length>;
    };

    //Additional device node descriptions

    :
    :
    :
};
```

Code 1: DTS Example File [14]

2.4 Device drivers

A device driver is a mechanism that allows the communication between a specific peripheral device and the kernel, using a well defined internal programming interface [15]. This enables the kernel to use the hardware without knowing the details of how it works.

In Linux, a device driver may be statically linked into the kernel, or it may be built as a separate kernel module. In the former case, the driver is loaded into RAM with the rest of the kernel at boot time. In the latter case, the driver can be loaded and unloaded dynamically by the running kernel, if and when required. A simple "hello world" kernel module is shown in Code 2.

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Code 2: Example Kernel Module

2.5 Tools for developing a Linux image

Several distributions and independent projects have developed multiple tools that simplify the creation and maintenance of a Linux image for embedded systems. In this section, we

present some of these tools.

2.5.1 The BuildRoot development tool

BuildRoot is an open source build system with a menu driven configuration tool, similar to the Linux kernel build system, that completely automates this process. It supports uLibc, a low-footprint alternative to the GNU standard C library, and BusyBox which combines many of the standard UNIX utilities and a shell into a single low-footprint executable [16].

BuildRoot is composed from a set of Makefiles that are used to generate a complete embedded Linux system. This is done by compiling an image for the kernel and bootloader, as well as generating a root file system for the target device. BuildRoot starts by generating a cross-compilation toolchain that will act as an environment used to build the whole system. This step is necessary because the architecture of the target system is often different from that of the host system that is compiling the system [17, Chapter 1]. That is the case with the WiiPiiDo, which has an Advanced RISC Machine (ARM) architecture, whereas the host system has an x86_64 processor.

2.5.2 The Yocto Project

The Yocto Project[1] derives directly from another open source project: OpenEmbedded. In 2003 some of the developers of the OpenZaurus project, a project for the Sharp Zaurus PDAs lineup, founded OpenEmbedded. Their goal was to create a build system for embedded Linux distributions based on a task scheduler inspired by the Gentoo Portage package system. This build system was dubbed BitBake.

In the meantime, in 2006, the embedded Linux start-up OpenedHand created Poky, which was a cleaner and easier-to-support fork of OpenEmbedded. In 2010, the Yocto Project was founded in the context of the Linux Foundation, providing the needed manpower to the OpenEmbedded Project for coherently organizing the metadata produced for building software for embedded systems. Poky was also donated by Intel for becoming the reference distribution of the project, thanks to its improvements to the OpenEmbedded build system [18].

2.5.3 The Armbian build system

Armbian[2] is a Ubuntu/Debian based lightweight distributions that are compiled specially for SoC, and ARM based boards. Officially born in 2015, Armbian started to support 17 boards, today it supports over 101 board configurations, with more than 3 Kernel/U-Boot branches for each SoC. Armbian uses the Debian `apt` package system and offers systems based on Debian Stretch and Buster or Ubuntu Xenial and Bionic. One of the user more appealing features is its dialog driven configuration utility that eases the configuration for inexperience users. It also provided specific runtime tools such as `armbian-config` which allows the user to change timezone, reconfigure language, locales, network, manage OpenSSHD settings, freeze kernel upgrades and toggle hardware settings.

[1] The Yocto Project – `https://www.yoctoproject.org/`
[2] Armbian – `https://www.armbian.com/`

2.5.4 ELBE

Originally developed by Linutronix, ELBE[3] is a Debian based system used to generate root file systems for embedded devices. ELBE is composed of the command `elbe`, which can be called with several subcommands to initialize the building environment, build a complete image for the target architecture, as well as debug and control of the environment. ELBE starts by creating a building environment, using a virtual machine, named `initvm` by default, which will be used to compile the images in the target architecture. The image configurations, such as architecture, SD card partitioning, etc, are all present in single XML file. For example, the package list of the default packages to install are presented in Code 3. A full example configuration file is also located in the Appendix A, in Code 29.

```
<pkg-list>
    <pkg>linux-image-686-pae</pkg>
    <pkg>grub-pc</pkg>
    <pkg>xserver-xorg-video-radeon</pkg>
    <pkg>xserver-xorg-core</pkg>
    <pkg>xserver-xorg-input-all</pkg>
    <pkg>xterm</pkg>
    <pkg>isc-dhcp-client</pkg>
    <pkg>net-tools</pkg>
    <pkg>network-manager</pkg>
    <pkg>mono-runtime</pkg>
    <pkg>slim</pkg>
    <pkg>awesome</pkg>
</pkg-list>
```

Code 3: ELBE Package List configuration example

2.5.5 OpenWrt

OpenWrt[4] , like the BuildRoot environment, is a collection of Makefiles, patches and scripts, which generates the cross-compilation toolchain, downloads Linux kernel, generates a root file system and manages 3rd party packages. Like BuildRoot, the cross-compilation toolchain uses uClibc. Therefore, developers can compile the custom firmware image for supported hardware architectures. In the OpenWrt source tree, there is no Linux kernel or any source code tarballs of 3rd party packages. The collection of Makefiles determines the version of Linux kernel to download, the version of the package tarball to be downloaded and compiled into the image that is created.

2.6 Comparison of development tools

To objectively compared the described building environments, minimal test images where built from each environment, where the images built from BuildRoot, Armbian, Yocto and OpenWrt where all targeted to the Pine A64 board, and the ELBE image was targeted to the Beagle Bone Black Board, as there was no official Pine A64 image that existed. We were not

[3] ELBE – `https://elbe-rfs.org/`
[4] OpenWrt – `https://openwrt.org/`

able to boot the ELBE image, so some parameters from this are not present in the final table. Table 2.1 then summarizes some preliminary differences that were observed from the different environments.

Observing this table, we can divide the environments in two types, the standalone and the Distribution-based environments.

In this sense, Armbian and ELBE can be categorized as Distribution-based environments, as they are directly based in Debian. By being based on an already established distribution, the images developed in this environments contain a multitude of already supported packages and facilities that make working in this images easier for an end-user. However, due to these, the images are not optimized, and can be harder to be trimmed to generate more compact images.

Alternately, BuildRoot, Yocto and OpenWrt end up being standalone environments, as they are not based in any already existing Linux Distribution. These environments generate more minimalistic images by default, and are customized for the target board. By result, the images generated in this environment are compact, which can be good when the target has limited resources, however, more time can be spent in tuning the environment, when compared to Distribution-based environments.

In the end, based in this finds, and as a way to be able to provide more options to the end-user we decided to develop two images, one from each of this categories. In the end, we used Armbian and Yocto.

Component	BuildRoot	Armbian	Yocto	ELBE	OpenWrt
Build system	Make	Shell	BitBake	Elbe CLI (Python)	Make
Configuration files	Configuration File and Patching	Configuration File and Patching	BitBake Recipes	Elbe XML File	Configuration File and Patching
Package manager	No	dpkg/apt	Yes[a]	dpkg/apt	ipkg/opkg
Package availability	No	Debian repositories (53320 packages)	No[b]	Debian repositories	OpenWrt repositories (10064 packages)
Host machine officially supported	Native	Native/VM/Docker	Native	VM (QEMU)	Native
Host machine environment size[c]	11 GB	15 GB	68 GB	17 GB	9.6 GB
Average build time	15 minutes	30 minutes	–[d]	45 minutes	42 minutes
Minimal image size	26 MB	610 MB	130 MB	–	7.3 MB
Minimal image installed packages	–	269	131	–	75
License	GPLv2+	GPLv2	MIT/GPLv2[e]	GPLv3+	GPLv2

Table 2.1: Build Environment Differences

[a] Yocto supports one or multiple package managers, including rpm, dpkg and opkg.

[b] Yocto has no official pre-built package repositories, but allows a user to add apt and user repositories.

[c] The size was taken after one compilation.

[d] Yocto differs from the other build environments in that the build time will depend on what was already built and what has changed since last build, as Yocto has `ccache`.

[e] Different components from the project have different licenses.

2.7 Synthesis

In this chapter we studied multiple building environments that allow a developer to create a custom Linux image for an embedded board. From this, we were able to extract the main features and differences from each studied environment and categorizing them in two different categories, distribution-based environments and standalone environments. In the end, we decided to use two different environments, one from each category, so as to give more options to an end-user which may have prefer either a totally custom or a more desktop-like image.

CHAPTER 3

Image Prototyping and Peripherals Integration

This chapter presents the steps performed in the developing of the first prototypes, as well as an example of a peripheral adaption to work in the WiiPiiDo board.

When the development of the images started, as the WiiPiiDo was still in development at the time, only a couple of prototypes where assembled. This was such that, if any hardware problem existed, it would be detected in this phase, before starting a larger production of the board. Therefore, it was required to validate as much as possible of the board, but with the precautions to not damage the prototypes, as there where only a couple of them, and a new production for testing could take weeks.

As such, instead of directly building the complete images using the chosen building environments, we started by validating the board in a more moderate way, where we purposely lower the requirements of the board, for example, lower the Central Processing Unit (CPU) and RAM frequency, as a prevention for problems that could occur with the prototypes.

This way, this first testing phase was divided in three parts:

1. **Simple Bootloader** – The first part for this test consists in compiling and running a simple bootloader, with an built-in shell, in which a few tests can be performed to validate the core board peripherals, such as the CPU, RAM and mass storage device.
2. **Minimal Linux Image** – Secondly, a minimal but complete Linux image that includes the bootloader, kernel and rootfs [1], was compiled and deployed to the board, for more in-depth testing of the board peripherals before we start developing the final images in the selected build environments.
3. **Driver Adaptation** – Finally, a few of the driver adaptations were done in this phase, as this image has a fast compilation time.

[1] Root File System – the top directory from the hierarchical filesystem present in Unix and Unix-like OS, where all other file systems are mounted in the boot up process [19]

With this approach we can quickly and safely validate and detect any hardware problems that the board may have in the core peripherals, before spending time to build a full image, which takes more work. Also, the time that was spent in this phase was not be totally lost, as, for example, the bootloader developed here can be used later in the full images, as well as the driver adaptations done in this phase.

3.1 Bootloader validation

3.1.1 Compiling the Bootloader

As mentioned in the previous section, we started by running a simple bootloader in the board.

To do so then, we first fetched the source files for the bootloader, U-Boot, from its official Git repository [20].

U-Boot makes use of a Makefile to build itself, and it requires to be configured for the target board before compiling it. This configuration is provided by a `.config` file that is present in the root folder of the source files. U-Boot already provides configuration files for its supported boards in the `configs` folder.

Seeing that a new board is being tested, no such file exists in this folder, but we can use a configuration file for an already supported board that has the same SoC as the one in WiiPiiDo, the Allwinner 64, as a base and reference for the configurations that will later be used in our board.

This way, the configuration file for the Pine A64 Plus [21], `pine64_plus_defconfig`, was chosen as the base for our configuration file. We chose the Pine A64 Plus, as it is one of the first, and well known SBCs featuring the Allwinner A64 SoC, having good support and stability from not only U-Boot, but also the kernel and the building environments that we will be using later on. This is the reason why we also used this board in those environments as a reference as well.

As such, the bootloader can be built by running the following in the command line:

```
$ make clean
$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make pine64_plus_defconfig
$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make
```

Code 4: Compilation of U-Boot with the Pine A64 configurations

The steps in Code 4 perform the following tasks:

1. Clean the repository
2. Create the `.config` configuration file using the Pine A64 configurations as base
3. Compile the bootloader image

By adding `ARCH=arm64` and `CROSS_COMPILE=aarch64-linux-gnu-` to the commands, we are specifying in the toolchain the target architecture and the cross-compiler to use.

After we let the compilation finish, we can verify that the compilation contains a warning, as presented in Figure 3.1. This is because it is missing the `PATH` for the Boot Loader Stage 3-1 (BL31) [22, chapter 4.12], binary, which is required to have a fully functional bootloader.

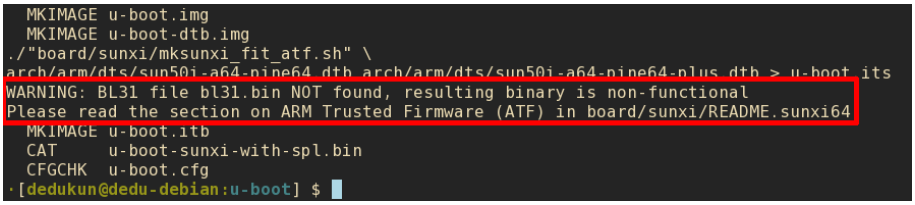

Figure 3.1: U-Boot Missing BL31 Warning

Therefore, to compile this binary we need to run the code described in Code 5, which will first fetch the source files for the BL31 and then compile the binary with the required target and cross compiler. Finally, it exports the variable `BL31` with the path of the built binary, which will then be used by the bootloader toolchain to retrieve this binary when needed.

```
$ git clone https://github.com/ARM-software/arm-trusted-firmware
$ cd arm-trusted-firmware
$ CROSS_COMPILE=aarch64-linux-gnu- make PLAT=sun50i\_a64 bl31
$ export BL31="$PWD/build/sun50i_a64/release/bl31.bin"
```

Code 5: Compilation of the BL31 for sunxi A64 SoCs

After the BL31 had been properly compiled and exported, the BuildRoot was recompiled using the `make` command once again, inside the U-Boot root folder.

3.1.2 Changing the configurations

At this point we already know that the bootloader is compiling correctly. Thus, we will start by making our desired modifications to lower the system requirements. This can was achieved by running the Make target `menuconfig`, which will start a Terminal User Interface (TUI) that allows us to modify with the bootloader configurations. As such, we executed the following command `ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make menuconfig`

After the command is executed, a menu similar to the one in Figure 3.2 will appear in the terminal.

Thereupon we can start to do our modifications. One of the precautions we can take is to reduce the RAM clock speed, as well as the CPU clock frequency. To do so, first it is required to check the correspondent peripherals datasheet to verify the slowest clock speed supported, which in this case is $333Hz$ for the RAM and $408MHz$ for the CPU.

To change RAM value, we need to first select the `ARM architecture` submenu, and in the option `sunxi dram clock speed`, change its default value to 333, as summarized in Code 6, which shows the menu progressing within the TUI, as well as the modified variable for the desired option.

```
ARM architecture  --->
    DRAM Type and Timing (DDR3 1333) --->
    (333) sunxi dram clock speed
    (3881915) sunxi dram zq value
```

Code 6: Change RAM clock speed

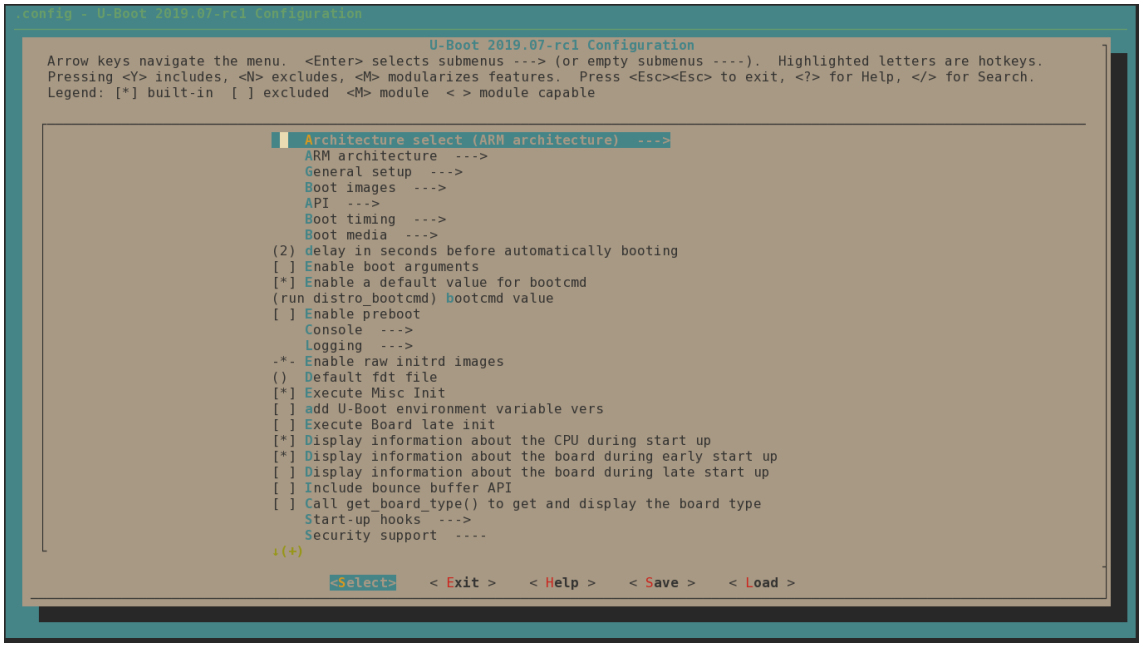

Figure 3.2: U-Boot Menuconfig Home

Similarly, to change the CPU frequency, inside the `Boot images` submenu, we changed the option `CPU clock frequency` to 408000000.

After the desired changes to the configurations are done, we can then save them by selecting `<Save>` in the bottom of the menu. A recompilation is then needed to build the bootloader binary with the latest modifications. This can be accomplished by executing `make` again.

3.1.3 Booting

At this stage, we can now deploy the bootloader to the test board. To achieve this, we flashed an SD Card, and booted from there.

However, before flashing, we first need to prepare an SD Card by deleting all of its partitions and formatting it, for example, as FAT32[2]. This can be accomplished in Linux with `GParted`[3].

After the SD Card is formatted, we will flash the compiled bootloader image, with the integrated Second Program Loader (SPL), which will copy the U-Boot from the SD Card to system RAM at boot time [23].

To do so, we executed the command in Code 7. The device `/dev/sdX` is being used here, as in later Codes, as a generic mountpoint for the SD Card in the build host machine.

```
$ sudo dd if=u-boot-sunxi-with-spl.bin of=/dev/sdX bs=1024 seek=8
```

Code 7: Flashing U-Boot

After the SD Card has been successfully flashed, we booted the board and started the validation of the core peripherals. To see the output from the board, we need to connect a

[2] *FAT32* – `https://support.microsoft.com/en-us/help/154997/description-of-the-fat32-file-system`
[3] *GParted* – `https://gparted.org/`

USB to TTL Serial converter, to the serial port of the board, as illustrated in Figure 3.3.

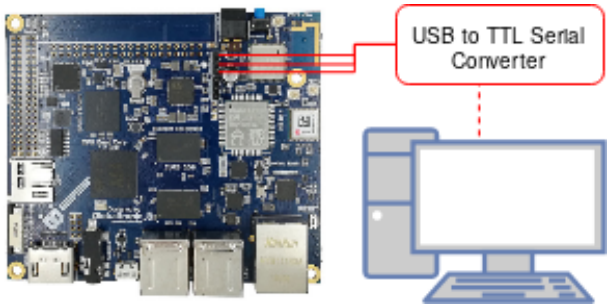

Figure 3.3: WiiPiiDo to PC Serial Connection

The image booted successfully, as shown in Figure 3.4. With this information, we can already validate that the CPU, RAM and SD Card Reader are working, as without these devices functioning, it would not be possible to boot from the SD Card. We also noticed that some USB where detected, as seen in the last lines of the output shown in Figure 3.4.

```
U-Boot SPL 2019.07-03716-gf96238e253 (Oct 21 2019 - 16:20:41 +0100)
DRAM: 2048 MiB
Trying to boot from MMC1
NOTICE:  BL31: v2.1(release):v2.1-841-g19adcb41
NOTICE:  BL31: Built : 11:15:31, Oct 21 2019
NOTICE:  BL31: Detected Allwinner A64/H64/R18 SoC (1689)
NOTICE:  BL31: Found U-Boot DTB at 0x408b148, model: Pine64+
NOTICE:  BL31: PMIC: Detected AXP803 on RSB.


U-Boot 2019.07-03716-gf96238e253 (Oct 21 2019 - 16:20:41 +0100) Allwinner Technology

CPU:   Allwinner A64 (SUN50I)
Model: Pine64+
DRAM:  2 GiB
MMC:   mmc@1c0f000: 0
Loading Environment from FAT... ** No valid partitions found **
In:    serial
Out:   serial
Err:   serial
Net:   phy interface7
Could not get PHY for ethernet@1c30000: addr 1
eth-1: ethernet@1c30000
starting USB...
Bus usb@1c1a000: USB EHCI 1.00
Bus usb@1c1a400: USB OHCI 1.0
Bus usb@1c1b000: USB EHCI 1.00
Bus usb@1c1b400: USB OHCI 1.0
scanning bus usb@1c1a000 for devices... 1 USB Device(s) found
scanning bus usb@1c1a400 for devices... 1 USB Device(s) found
scanning bus usb@1c1b000 for devices... 1 USB Device(s) found
scanning bus usb@1c1b400 for devices... 1 USB Device(s) found
       scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot:  0
=>
```

Figure 3.4: U-Boot First Boot

Depending on the installed U-Boot version, a few test commands exist in the Command Line Interface (CLI), which allow further testing [24]. An example of such command is the `mtest` command, which performs a simple RAM test, or the command `mmc`, which allows to do some basic read/write tests to the eMMC.

The final modifications are located at the Globaltronic's U-Boot Fork [25].

3.1.4 Boot using FEL

In the first revision of the board, a problem was detected in one of the prototypes with the SD Card Reader, which prevented us to boot the board from the SD Card. To alleviate this problem from disabling us to further test the rest of the board when the issue was being resolved, we made use of a special subroutine that exists in some Allwinner SoCs, including the Allwinner A64, that allowed us to boot directly over USB OTG, by loading the bootloader from the USB to RAM, enabling us to forgo the SD Card completely [26].

With this tool, we not only were able to test the bootloader, but the minimal Linux image as well, as FEL supports booting a complete system over USB, limited only by the board RAM size.

To use this mode, first we fetched the official tools from the sunxi repositories [4], and compiled the code. This was achieved by running the code in Code 8.

```
$ git clone https://github.com/linux-sunxi/sunxi-tools.git
$ cd sunxi-tools
$ make
```

Code 8: Compiling Sunxi-tools

After the tools were compiled, we connected a USB OTG cable, with no power, to the board. Subsequently, we forced the board to enter the FEL subroutine, which can be accomplished in 5 ways [27]:

- Pressing a dedicated button
- By holding a standard button in a specific manner
- By inputting special characters though serial console
- Using a special SD Card which enter FEL mode
- Having no valid boot image

In our case, WiiPiiDo has a dedicated button to enter FEL mode. Finally, we only need to execute the command described in Code 9, which will get the appropriate files and write them to the correct addresses in memory.

```
$ sudo ./sunxi-fel -v -p spl sunxi-a64-spl32-ddr3.bin write 0x44000 </path/to/arm-trusted-firmware/bl31.bin>
write 0x4a000000 </path/to/u-boot/u-boot.bin> reset64 0x44000
```

Code 9: Upload Bootloader using FEL

Unfortunately, the AArch64 branch for the U-Boot SPL does not support FEL, since FEL is done entirely in AArch32. So, to boot the Allwinner A64 from FEL we need to combine the AArch32 SPL, with the previously compiled U-Boot and BL31, as referenced in the official documentation [28].

3.2 Creating a minimal Linux image

After the core peripherals validation was completed, a minimal Linux image was built.

[4] *Sunxi Tools Git Repository* – `https://github.com/linux-sunxi/sunxi-tools`

The procedure for this build was similar to the U-Boot compilation, in which we will compile a minimal image, with minimal requirements. This will result in a lower building time, as well as to avoid possible kernel errors due to incorrect device integration when probing the hardware.

Therefore, to build the first complete Linux image, BuildRoot will be used, seeing that it allows full customization over the image to be built, and by default, it has a small image size compared to other building environments, and consequently, a faster compilation time, as it has a lot of kernel, U-Boot, and other settings disabled by default.

The source files for the build environments was fetch from the official sources [29]. All of the system requirements where also installed, which where found in the official documentation [17, chapter 2].

3.2.1 Building the image

As referred in 2.5.1, BuildRoot consist of a set of Makefiles, which will create the complete Linux Image that can then be flashed to the board, including the bootloader and the kernel with respective Device Tree Blobs (DTBs), the root file system, as well as BusyBox, which provides most of the GNU utilities in a single small executable [30].

Similarly to U-Boot, BuildRoot also has configuration files that contain the configurations for different boards already supported by the environment.

We will be using the same approach as we used with the bootloader, and first compile an image for the Pine A64 Plus, to serve as a base for the configurations.

As such, to build the default BuildRoot image for the Pine A64 Plus, we need to execute in the command line the Code 10.

```
$ git clone git://git.busybox.net/buildroot
$ cd buildroot
$ make pine64_defconfig
$ make
```

Code 10: Compilation of BuildRoot using the configuration file `pine64_defconfig`

The chosen building configurations already define the target board, so there is no need to specify the architecture and compiler to used, contrarily to the U-Boot process.

3.2.2 Configuration menu

After making sure that BuildRoot is compiling correctly, we started making our desired modifications to the image. Opposite to the U-Boot configuration menu, BuildRoot provides several interfaces to change its configurations, from curses (menuconfig) and ncurses (nconfig) TUIs, to interactive Qt (xconfig) and GTK (gconfig) interfaces.

BuildRoot can also independently invoke the configuration menu for the U-Boot (`uboot-menuconfig`), Kernel (`linux-menuconfig`), BusyBox (`busybox-menuconfig`) and uClibc (`uclibc-menuconfig`), which is the C library installed by default.

In our case, we will be using the curses configuration menu, so we start the TUI by running `make menuconfig`, which produces a display similar to the one in Figure 3.5.

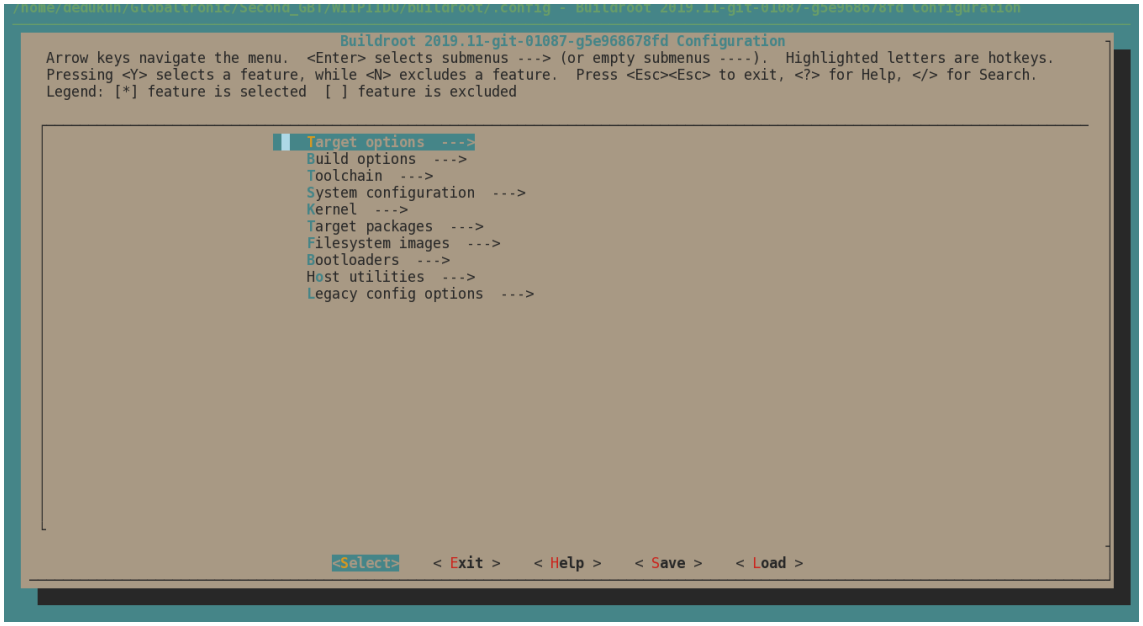

Figure 3.5: BuildRoot `menuconfig` Home

Some of the sub menus to take note are:

- `System configuration` – Configuration for the target system, such as the PATH, hostname, shell, locale, etc.
- `Kernel` – Top configuration for the kernel (the version to use, kernel patches, device tree blob, etc). For a more in-depth configuration we use the kernel configuration menu (linux-menuconfig).
- `Target packages` – The packages to be included in the filesystem. The majority of the coreutils are in BusyBox (busybox-menuconfig).
- `Filesystem images` – Configurations for the file system (ext4, btrfs, etc).
- `Bootloaders` – Top configuration for the bootloader (the version to use, the *defconfig* file, etc). For a more in-depth configuration we use the U-Boot configuration menu (uboot-menuconfig).

The modifications done in this stage result in:

Change CPU Governor As it was done in the U-Boot configuration, we will start by changing the CPU Frequency. In the Kernel however, there is not a fixed CPU Frequency value, but a CPU Governor, or in other words, the frequency profile. By default, the CPU is using an `ondemand` or `performance` profile, which scale the CPU Frequency by the current demand from the system, or have it always using the max frequency, respectively. We changed this option to `powersafe`, which will make the CPU always use the lowest frequency supported. This option belongs in the Kernel configurations, which can be called using the Make target `linux-menuconfig`. Inside the submenu `CPU Frequency scaling` in `CPU Power Management`, we modify the `Default CPUFreq governor` to `powersafe`, which will tell the Kernel to use the lowest supported CPU frequency. A summary of the TUI submenus if demonstrated in Code 11.

```
CPU Power Management  --->
    CPU Frequency scaling  --->
        Default CPUFreq governor (powersafe) --->
```

Code 11: Change the Kernel CPU Governor

Change the DTSs used Another option that we changed is the DTS used by the Kernel and bootloader. This is the because the version of BuildRoot in use does not contain a configuration file for the Pine A64 Plus, only the Pine A64, which belongs to the same family of SBCs, however, the configurations for this board exist internally in the U-Boot and Kernel repositories. As such, we can modify the BuildRoot options that states which DTS or configuration file to use in both the Kernel and U-Boot. In the case of the Kernel DTS definition, this option resides in the BuildRoot Configurations, inside the `Kernel` submenu, by the name of `In-tree Device Tree Source file names`, as summarized in Code 12. The desired value is `allwinner/sun50i-a64-pine64-plus`.

```
Kernel  --->
    (allwinner/sun50i-a64-pine64-plus) In-tree Device Tree Source file names
```

Code 12: Changing the Kernel DTS in BuildRoot

As for the, this modifications is described in Code 13.

```
Kernel  --->
    [*] U-Boot
        (pine64_plus) Board defconfig
```

Code 13: Changing the U-Boot configuration file in BuildRoot

Apply the modifications made to the bootloader The modifications made to the bootloader can be applied once again, following the steps described in Section 3.1.2, using the U-Boot configurations menu target `uboot-menuconfig`. Alternately, we can later, before booting the image, overwrite the bootloader compiled by the BuildRoot, with the one compile previously, by running the command in Code 7.

3.2.3 Booting

After a successful compilation, we can now boot the built image. All of built binaries reside in the folder `buildroot/output/images`, such as the bootloader, kernel, rootfs, etc, as demonstrated in Figure 3.6.

```
·[dedukun@dedu-debian:buildroot] $ ls output/images/
Image     sunxi-spl.bin  sun50i-a64-pine64.dtb      rootfs.ext2  sdcard.img  boot.scr    boot.vfat
bl31.bin  u-boot.bin     sun50i-a64-pine64-plus.dtb  rootfs.ext4  u-boot.itb  rootfs.tar
```

Figure 3.6: BuildRoot output images

BuildRoot will also generate the file `sdcard.img`, which contains the complete image in a single file. This way, to flash the image in a formatted SD Card, we execute the Code 12.

We can now boot the new image, and do further validations to the board devices. The default login credentials are username `root`, and no password.

```
$ sudo dd if=sdcard.img of=/dev/sdX bs=1M status=progress
```

Code 14: Flashing the BuildRoot Image

3.3 Peripherals integration

At this stage, we can now start doing a more thorough validation of each peripheral present in the board. To make this process more systematic, an approach was defined, which was based on the principle that when a peripheral is not working correctly, there are generally only three possible locations where modifications have to be made, which are:

- The Kernel configurations
- The DTS
- The Device Driver (Firmware)

The outlined approach is illustrated in Figure 3.7.

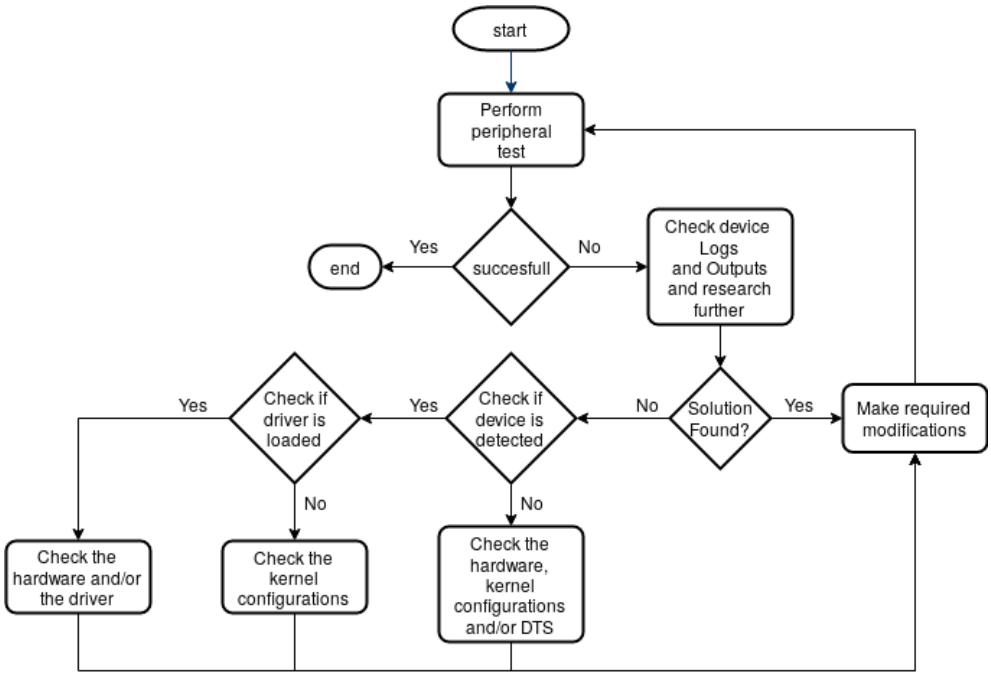

Figure 3.7: Device Validation Approach

To start the validation of a specific peripheral, a respective validation test needs to be specified. For example, to validate a GPIO port, a test to make sure that the GPIO is working correctly is to use the file system interface to manipulate an individual port and read/write values from/to it. Once this test is defined we can then perform it. If the test was successful, the peripheral is validated and working correctly, if not, we retrieve the outputs from the test, as well as possible kernel and/or system logs, provided, for example, by the command `dmesg` and the file `/var/log/syslog`, respectively. Once this information is gathered, we can then research further for a possible solution by consulting available resources, such as online documentation or forums, as well as other developers. If a solution was found in this process,

we then apply it and try the validation test again. If no solution was found, we can try to narrow down the potential causes of the problem by doing the following validations:

- `Verify if the peripheral is being detected.` If the device is not being detected correctly, there are generally three possible causes:
  - An hardware problem, such as the device not being powered or soldered correctly.
  - An error in the kernel configuration.
  - An error in the DTS.
- `Verify if the peripheral driver is being loaded.` If the device driver is not being loaded, then generally the problem must be in the kernel configurations.

Finally, if these validations were successful, then the problem must be in the device driver.

Errors in the device driver are the last possibility to consider in this approach because, unlike the kernel configurations and the DTS, the device driver should generally be independent of the particular target board or build environment being used.

3.3.1 Example: GPIO port

As an example of the device validation approach, when we tried to use a GPIO port as an output pin, we noticed that the file system interface for the device was absent (see Figure 3.8). Therefore, the Kernel configurations where checked and noted to not have the sysfs interface

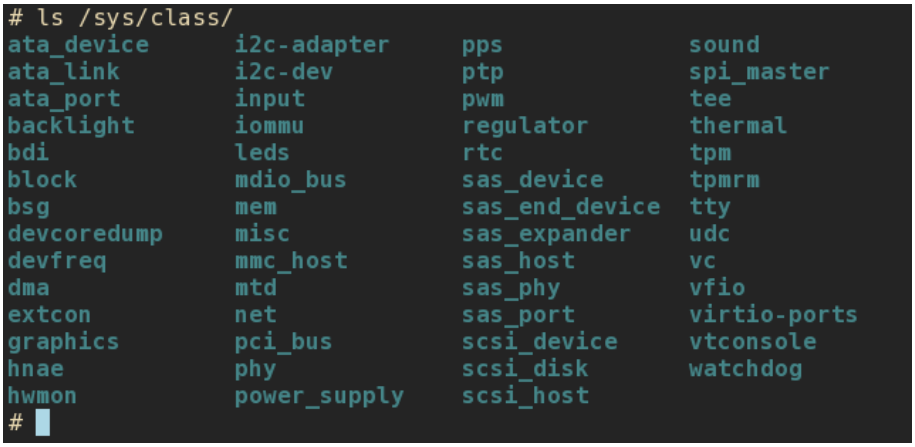

Figure 3.8: Missing GPIO sysfs interface. The `/sys/class/gpio` directory is absent.

for the GPIOs activated. The problem was easily fixed by toggling the sysfs interface option in the `GPIO Support` inside the `Device Drivers` submenu of Kernel configurations (Code 15).

```
Device Drivers  --->
    GPIO Support --->
        [X] /sys/class/gpio... (sysfs interface)
```

Code 15: Activating the sysfs interface for the GPIOs

3.4 Synthesis

In this chapter we looked over and defined a process to rapidly validate the components of the board in situations of simultaneous hardware and firmware development. We argue that in

such cases the development should proceed in stages, starting from a simple bootloader, moving on to a minimalistic but complete Linux image and finally, ending with the development of the final images in the chosen building environments. In each stage, different building environments may be used, allowing the developer to profit from the distinct advantages of each environment in different stages of development.

CHAPTER 4

Building The Final Images

This chapter presents the Armbian and Yocto building environments, as well as some of the modifications that were implemented to support the WiiPiiDo board.

With the in aim to compare the chosen build environments, we are going to show in some detail the process used to compile the final images.

4.1 Building an image in Armbian

As it was discussed in the Section 2.5.3, Armbian is a Debian-based distribution specially develop for ARM SoCs. The build system consists of a series of shell scripts, and the structure of the environment is summarized in Figure 4.1.

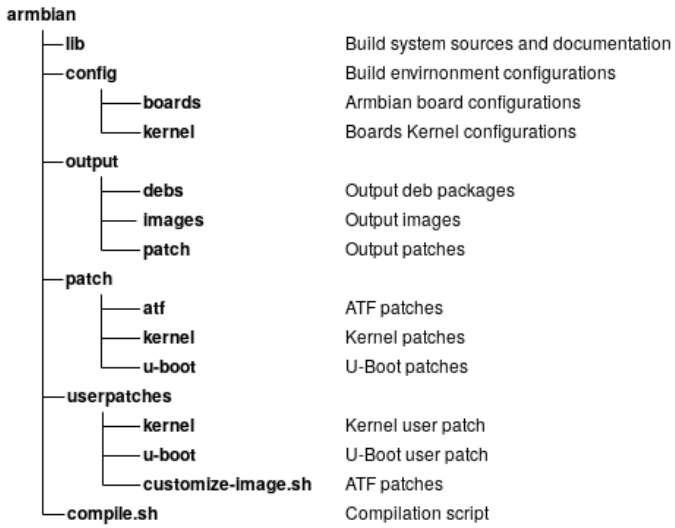

Figure 4.1: Armbian Folder Structure

Armbian has two types of configurations that can be made. There are user configurations, which are associated to the developers using the environment and are meant to be personal

modifications, associated only to the specific repository being used. Additionally, there are Armbian official board configurations, which are the permanent configurations associated with the build environment in itself. As such, the `userpatches` directory contains the user configurations, and the `patch` and `config` folders contain the Armbian official configurations.

4.1.1 Building a test image

To compile an Armbian image, we first fetched the source files [31] and used the script `compile.sh`. Armbian supports several alternative ways of building images:

- building natively in a host machine;
- building in a virtual machine using VirtualBox, for example;
- building with Vagrant, which manages VirtualBox images in an easily repeatable way;
- building inside a Docker container.

In our case, we decided to compile natively in our host machine, so that we could later fairly compare the compilation time between building environments, and to generally compile the Armbian images faster. To start the building process, we executed the commands in Code 16. The compilation option `NO_HOST_RELEASE_CHECK=yes` was required in our case, because we did not use the officially supported host OS.

```
$ git clone --depth 1 https://github.com/armbian/build armbian
$ cd armbian
$ ./compile.sh NO_HOST_RELEASE_CHECK=yes
```

Code 16: Armbian Quick Start

After the compilation started, a TUI will open asking a few options that need to be picked before the compilation can start. This options are presented in the following list, as well as selections that were made.

1. The output of the compilation
   - Just the bootloader and kernel
   - The full image [selected]
2. To open the kernel configuration menu before compiling
   - Yes
   - No [selected]
3. The target board [selected `pine64`]
4. The kernel version
   - `default` Legacy kernel (3.10.y)
   - `next` Mainline kernel (4.19.y) [selected]
5. The distribution and its release to build
   - `stretch` Debian 9 [selected]
   - `buster` Debian 10
   - `xenial` Ubuntu 16.04
   - `bionic` Ubuntu 18.04

6. The type of image
    - Server image [selected]
    - Desktop image
7. The image installed packages
    - Standard
    - Minimal image with less installed packages [selected]

As an alternative, we could also give this information to Armbian as command line options, the same way as it was done previously to stop the host machine verification. In a command line only format, the options selected previously are equivalent to the following command:

```
./compile.sh NO_HOST_RELEASE_CHECK=yes BOARD=pine64 BRANCH=next RELEASE=stretch
BUILD_MINIMAL=yes BUILD_DESKTOP=no KERNEL_ONLY=no KERNEL_CONFIGURE=no
```

The full list of the building options can be found in the official documentation [32, Build Options]. The output from the compilation is demonstrated in Figure 4.2.

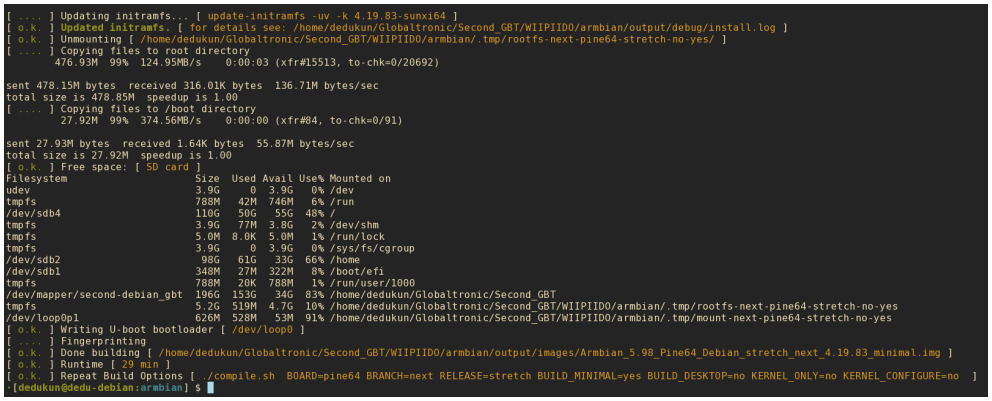

Figure 4.2: Armbian Compilation

4.1.2 Adding support for the WiiPiiDo

Once its verified that an image compiled successfully, we started adding the support for the WiiPiiDo board. This is mainly focused in four areas:

- Kernel and bootloader building configurations
- Kernel and bootloader DTSs
- Kernel and bootloader source code modifications
- Compiled image settings, like the target file system, installed packages, default user configurations, etc

These modifications are also divided in two phases that were discussed previously, which are respectively the user modifications and the Armbian official configurations. We started by focusing first in making our modifications as user patches, and later adapt them to be able to be added as Armbian official support.

Changing the build configurations To modify the kernel options, we invoked the kernel configuration menu by using the command line option `KERNEL_CONFIGURE=yes`, in conjunction with the option `KERNEL_KEEP_CONFIG=yes`, which tells Armbian keep the configurations

between compilations. After the modifications where validated, we copied the resulting file `.config` to `userpatches/linux-sunxi64-next.config` [32, User Configurations].

For the bootloader, there is no direct interface as the one used in the Kernel. In this case, we had to make a patch in the U-Boot source code. Alternately, we could have changed the U-Boot Repository used, for a custom repository with the changes made in the previous built U-Boot image. However, for initial development, we flashed the previous working U-Boot image, as in Code 7.

Changing the DTSs The development of the DTS is easy in Armbian as it supports Device Tree Overlays by default. As such, we can add the overlays to the SBC, and test them without the need to recompile a new image. The overlays are located at `/boot/dtb/allwinner/overlay`, and can be compiled and activated using the command `armbian-add-overlay`, that has the following syntax `armbian-add-overlay <overlay_file.dts>`. The file `/boot/armbianEnv.txt` is used to activate/deactivate, as well as to pass parameters to the overlays if needed.

Once again, there is no simple way to change the bootloader DTS. To do modifications to the bootloader DTS we can use the same methods as the ones described previously.

Changing the source code The method to change the source code of the kernel and bootloader is the same, which is to create patches with the desired modifications from the sources.

To do this, Armbian provides an interactive patch creating tool, which is started when passing the option `CREATE_PATCHES=yes` to the build system. Armbian will, right before starting the compilation, wait for the user to make its desired modifications to the kernel and bootloader respectively. Once the modifications are made, it will generate a patch file with them and apply it, as demonstrated in Figure 4.3.

```
[ o.k. ] * [l][c] lower-default-DRAM-freq-A64-H5.patch
[ o.k. ] * [l][c] sun8i-set-machid.patch
[ o.k. ] * [l][c] sunxi-boot-splash.patch
[ warn ] Applying existing u-boot patch [ /home/dedukun/Globaltronic/Second_GBT/WIIPIIDO/armbian/build/output/patch/u-boot-sunxi64-next
.patch ]
[ warn ] Make your changes in this directory: [ /home/dedukun/Globaltronic/Second_GBT/WIIPIIDO/armbian/build/cache/sources/u-boot/v2019
.04 ]
[ warn ] Press <Enter> after you are done [ waiting ]
[ o.k. ] You will find your patch here: [ /home/dedukun/Globaltronic/Second_GBT/WIIPIIDO/armbian/build/output/patch/u-boot-sunxi64-next
.patch ]
3.2.1.'/home/dedukun/Globaltronic/Second_GBT/WIIPIIDO/armbian/build/.tmp/atf-sunxi64-pine64-next/bl31.bin' -> './bl31.bin'
```

Figure 4.3: Armbian Source Files Patch

It is important to remember that Armbian is using the `git diff` command to generate the patches, so modifications made in files that are present in `.gitignore` will not be detected. Once the patches are created, they will be placed in the directory `outputs/patch`. If further compilations are still executed with the option `CREATE_PATCHES=yes`, the build system applies the previously generated patches, however, if it is not present, they will not be applied. To make these patches permanent, we placed them in the directories `userpatches/kernel/sunxi-next` and `userpatches/u-boot/u-boot-sunxi`, for the kernel and bootloader patches respectively.

Changing the image settings To change the image settings, we made use of the file `userpatches/lib.config`, which redefines some of the configurations used by Armbian,

such as the kernel and bootloader version used, as well as the initially installed packages [32, User provided configuration], as shown in the example in Code 17.

```
PACKAGE_LIST_ADDITIONAL="$PACKAGE_LIST_ADDITIONAL nodejs" # Add nodejs
BOOTSOURCE=https://github.com/Globaltronic/u-boot.git     # Change the U-Boot source
BOOTCONFIG=wiipiido_defconfig                             # Change the U-Boot defconfig
```

Code 17: Changing Armbian Configurations

4.2 Building an image in Yocto

Yocto is different from Armbian, as Yocto is not a distribution in itself, but a set of tools that allow the developer to create a full Linux Distribution. To do so, Yocto defines metadata, or sets of instructions and configurations used by the OpenEmbedded's build system, `bitbake`. Related recipes are then organized together in different layers, where there are:

- BSP layers, which contain the recipes necessary to add support for a system
- Application layers, that contain the necessary recipes to install an application
- Distribution layers

To better understand how to use the environment, we will first look at Poky [33], the reference repository that is recommended to be used as the base for any Yocto image. It contains the build system, `bitbake`, as well as the core recipes that will be used in the majority of the projects. The structure for this directory is summarized in Figure 4.4.

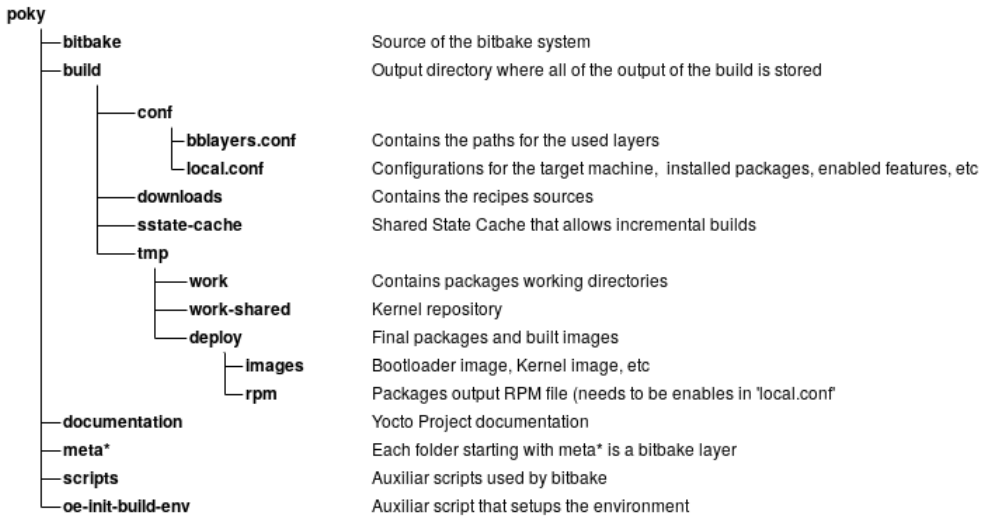

Figure 4.4: Poky Repository Structure

Some of the files and directories to take note from this Figure are:

- The `oe-init-bulid-env` script, which is used to initialize the build environment.
- The directory `build/conf`, that contains the main configurations for the image to be compiled, such as the configuration of the target board, the selection of the packages to be installed, the meta layers to included, as well as the target filesystem, etc.

- The `meta-*` directories, which represents a bitbake layer.

A layer can then be expanded further, as illustrated in Figure 4.5.

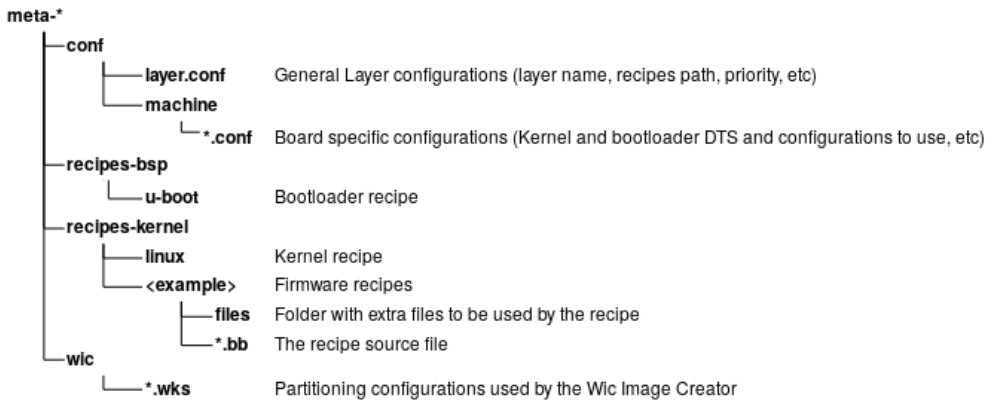

Figure 4.5: Yocto BSP Layer Structure

In this figure it is summarized a BSP layer, which consist in:

The `conf` directory This directory has the general configurations for the layer, such as the name of the layer, recipes path, the priority of the layer, which is used to chose a layer from multiples with the same name, and the layer's compatibility, i.e., in which versions of Poky does the layer work on. Also present in this directory are the board specific configurations, that contain the version and configurations of the kernel and bootloader to use, as well as the target file system, pre-installed packages, recipes dependencies, etc. Code 18 is a snippet of a board configuration file. In Appendix A, Code 30 there is the full configuration file that ended up being used in WiiPiiDo.

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-wiipiido"
PREFERRED_VERSION_linux-wiipiido ?= "4.19%"
KERNEL_CLASSES = "kernel-fitimage"
KERNEL_IMAGETYPE = "fitImage"
KERNEL_DEVICETREE = "allwinner/sun50i-a64-wiipiido.dtb"

MACHINE_EXTRA_RRECOMMENDS += "kernel-modules linux-firmware-brcm43430"

IMAGE_FSTYPES += "wic"
WKS_FILE ?= "wiipiido-bsp-image.wks"
```

Code 18: Yocto BSP Board Configuration Snippet Example.

The `recipes-*` directory This directory has the recipes that are present in the layer. Each recipe contains at least a `*.bb` file, which is the recipe in itself, and may have additional files used by the recipe in the separate folder `files`.

Each recipe contains a set of tasks that have the information of the instructions to be accomplished in a given step of the building process. The most important tasks that exist in a recipe are [34]:

1. **`do_fetch`** – Fetches the source code
2. **`do_unpack`** – Unpacks the source code into a working directory
3. **`do_patch`** – Locates patch files and applies them to the source code

4. **do_configure** – Configures the source by enabling and disabling any build-time and configuration options for the software being built
5. **do_compile** – Compiles the source in the compilation directory
6. **do_install** – Copies files from the compilation directory to a holding area
7. **do_package** – Analyzes the content of the holding area and splits it into subsets based on available packages and files
8. **do_package_write_rpm** – Creates the actual RPM packages and places them in the Package Feed area

Recipes can also inherit definitions from other recipes, and Yocto already provides base recipes for common building utilities used in most applications like Autotools, CMake, etc [35, Chapter 3.3.10]. Beyond the tasks definitions, the recipe also has information about the sources and licensing. Code 19 contains the source file of a simple recipe.

```
SUMMARY = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
   ${CC} helloworld.c -o helloworld
}

do_install() {
   install -d ${D}${bindir}
   install -m 0755 helloworld ${D}${bindir}
}
```

Code 19: Yocto Recipe Example [35, Chapter 3.3.21.1].

The `wic` directory The `wic` directory contains the partitioning configurations. An example configuration is presented in Figure 20.

```
part spl --source rawcopy --sourceparams="file=u-boot-sunxi-with-spl.bin"
     --ondisk mmcblk --no-table --align 8
part /boot --source bootimg-partition --ondisk mmcblk0 --fstype=vfat
       --label wiipiido --active --size=100M --align 20480
part / --source rootfs --ondisk mmcblk0 --fstype=ext4 --label platform --align 4096
```

Code 20: Wic Partitioning Configurations.

4.2.1 Building a test Image

Similarly to the other images that were created with the different build environments, first we started by building a test image, making use of Poky as the reference distro, as mentioned in the last section. As such, we first fetched the Poky repository from the Yocto Project, making sure we used the last stable version, which at the time of this development was version

2.7.1, codenamed `warrior`. The full system requirements for the build host are defined in the official documentation [36, Compatible Linux Distribution]

```
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git fetch --tags
$ git checkout tags/yocto-2.7.1 -b wiipiido
$ source oe-init-build-env
$ bitbake core-image-base
```

Code 21: Yocto Quick Start [36].

The first compilation took more than 1 hour, as bitbake is downloading all the required packages sources to compile, such as the kernel, bootloader, BusyBox and extra packages defined in recipes or local configurations. Further compilations are optimized by bitbake, as it automatically recognizes what has changed and will compile only recipes that received modifications since the last build.

By default, the target for the base image that was just compiled is set to be a `x86 QEMU` [1] image, which will work as a machine emulator and run images compiled in other architectures other than the one of the build host machine. Once the image was built, it was started with the command `runqemu qemux86`. This is a useful feature as it allows a developer to test recipes, applications, etc, in a virtual environment, without the need to have the specific hardware platform. After this image was checked to be working in QEMU, we started compiling the same image now for our test board. To accomplish this, we changed the target for the image. We started by adding a BSP layer for the Pine A64 board and compile it, which later was modified to support the WiiPiiDo BSP layer.

The Pine A64 BSP layer is not included in the Poky repository. As such, we used the OpenEmbedded Layer Index [37], a website that contains, as the name sugests, an index of multiple layer and recipes for the OpenEmbedded, and consequently, the Yocto Project. After finding the layer, we cloned it to the root of the Poky directory, and added it to the build environment, using the `bitbake-layer` command, as demonstrated in Code 22. Alternately, we could also manually add a new layer's path to the `conf/bblayers.conf` configuration file.

```
$ cd poky
$ git clone https://github.com/alistair23/meta-pine64
$ source oe-init-build-env
$ bitbake-layers add-layer ../meta-pine64
```

Code 22: Adding the Pine A64 BSP Layer

After the layer was added successfully, there were a couple of configurations that had to be adjusted. The first one was the target board for bitbake, which is defined in the `conf/local.conf` file. This configuration now needed to be changed to target the Pine A64 board. The second adjustment was to add the format of the root filesystem. The BSP Layer is already defining the root filesystem to be a Wic Image, however, we will be adding another

[1] *QEMU: Machine emulator and virtualizer* – `https://www.qemu.org/`

option so that `bmaptool` [2], the reference tool used in Yocto to flash images, can be used optimized. The described steps are illustrated in Code 23.

```
@@ -35,7 +35,8 @@
 #MACHINE ?= "edgerouter"
 #
 # This sets the default machine to be qemux86 if no other machine
is selected:
-MACHINE ?= "qemux86"
+MACHINE ?= "pine-a64-lts"
+IMAGE_FSTYPES += "wic wic.bmap"

 #
 # Where to place downloads
```

Code 23: Changes done to the local configurations

The only steps missing were recompiling the image and flashing it to a formatted SD Card, which were accomplished with the following commands:

```
$ bitbake core-image-base
$ sudo bmaptool copy tmp/deploy/images/pine-a64-lts/core-image-base-pine-a64-lts.wic /dev/sdX
```

When testing the image, we came across an error in the bootloader, which was not booting correctly. To alleviate us from debugging the bootloader at this stage, the old U-Boot image that had been compiled previously, as in Section 3.1, was used. To do so, we overwrote the malfunctioning bootloader as demonstrated in Code 7.

4.2.2 Adding support for WiiPiiDo

With the image now booting and working as expected, we started with adding the support for the WiiPiiDo board. This mainly took place in the respective BSP layer, however some modifications to the local configurations, the `build/conf/local.conf` file, were also made. The modifications that need to be done to add support for the WiiPiiDo consisted in:

- Modifying a recipe, explicitly the bootloader and kernel recipes
- Adding missing firmware and packages if necessary

In a later stage when the layer had been validated to be working correctly, the general metadata was also changed to contain the board information. This mainly consisted in changing the name of board in the layer configurations files, and recipes from `pine-a64` to `wiipiido`.

Modifying a recipe In a BSP layer, to add support to a new board, mainly the kernel and bootloader recipes will need to be modified. As both modifications follow the same steps, only the kernel changes will be used as an example in this step.

There are mainly two types of modifications that need to take place in the recipe, which are, respectively, changes to the configurations menu and changes to the source.

[2] *BMAP Tools* – `https://github.com/intel/bmap-tools`

To modify the kernel configurations, the `menuconfig` tasks was used, which invokes the default `menuconfig` from the Kernel sources, that can be modified as desired. Any other modification that needs to be done to the DTSs, or other source files were made with the `devshell` task, which starts a shell with the environment setup for developing/debugging the specified recipe. Code 24 presents the call for both tasks, respectively. In this case, `virtual/kernel` is the name of the kernel recipe.

```
$ bitbake -c menuconfig virtual/kernel
$ bitbake -c devshell virtual/kernel
```

Code 24: Invoke the kernel's menuconfig and development shell

After the desired changes were completed, we were able to simply rebuild the image to validate them. This however does not make the modifications persistent in the BSP layer. To do so, we needed to first generate patches and link them to the recipe.

The modifications to the Kernel configurations can be retrieve using the `diffconfig` task, which will generate a `*.cfg` file with the differences between the default layer configurations and ours.

For the other modifications, the patch can be built using `git`'s builtin patch command. As such, the modifications that were tested before were replicated in a cleaned state of the kernel's git repository. This was accomplished by copying the repository to another directory, and after cleaning the repository, rebuilding the changes. This has to be done as the recipe already patches the kernel separately, and we will want our modification to be in a different patch. This way, the commands needed to generate a patch to the source files are in Code 25.

```
$ bitbake -c devshell virtual/kernel
$ cp -r ../kernel-source <temporary/kernel/path>
$ cd <temporary/kernel/path>
$ git reset --hard
$ git clean -fdx
DO MODIFICATIONS TO SOURCE FILES
$ git add *
$ git commit -m <msg>
$ git format-patch -1
```

Code 25: Kernel Patch Procedure

At this point we added the newly formatted patches to the recipe. This was accomplished by copying the generated patches files to the `files` subfolder inside the kernel recipe directory, and then add the files path to the `SRC_URI` variable inside the recipe. The patches are automatically applied in proceeding builds.

Adding missing firmware When testing the image, it was noted that a few peripherals were not working correctly. This was due to the fact that some firmware was missing from the image. There are a couple of ways to fix the issue, we can add the required packages directly in the board configuration file, or alternately, we can add the packages to the local configuration file.

The syntax for both cases is the same, with only difference being that in the first case, the packages is directly added to the BSP layer, so anyone that uses the layer will have this packages installed by default, and in the second case only when explicitly specified by the user will the package be installed.

In our case we decided to go by the second route, to allow a user to explicitly specify if the firmware is to be installed. Code 26 shows the modification done to the `build/conf/local.conf`.

```
@@ -253,3 +253,6 @@
 # track the version of this file when it was generated. This can safely be ignored if
 # this does not mean anything to you.
 CONF_VERSION = "1"
+
+# LINUX FIRMAWARE
+IMAGE_INSTALL_append = " linux-firmware"
```

Code 26: Adding Missing Firmware

4.3 Synthesis

In this chapter we demonstrated how use the chosen building environments, Armbian and Yocto, and highlight the main differences between their building processes.

In Armbian we configure the kernel and bootloader using patches and configurations files, as well as add specific user configurations that modify the final rootfs. When the build process starts, Armbian will fetch the sources for the kernel and bootloader, apply the patches and configurations that we provided, compile and then generate the final image. During this process Armbian fetches all the packages integrated in the image from the official Debian repositories. Figure 4.6 summarizes this process.

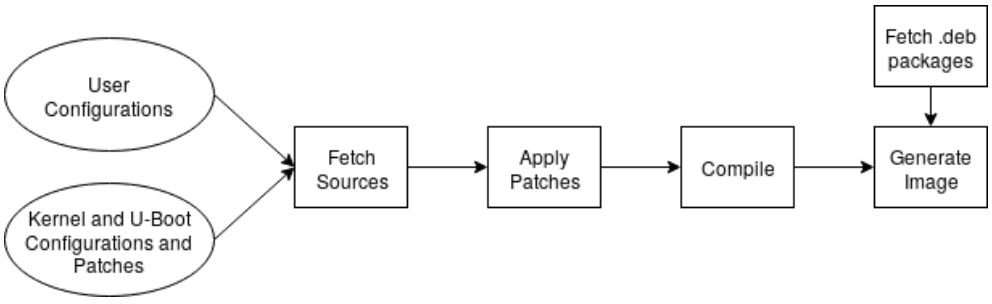

Figure 4.6: Armbian Build Process

In contrast, Yocto will fetch and compile all of the image software and firmware, from the kernel to program applications, not having public repositories with pre-built packages. Therefore, everything that is contained in the final image has an associated recipe that provides the steps that are required to build everything from source. Figure 4.7 summarizes the building process in Yocto.

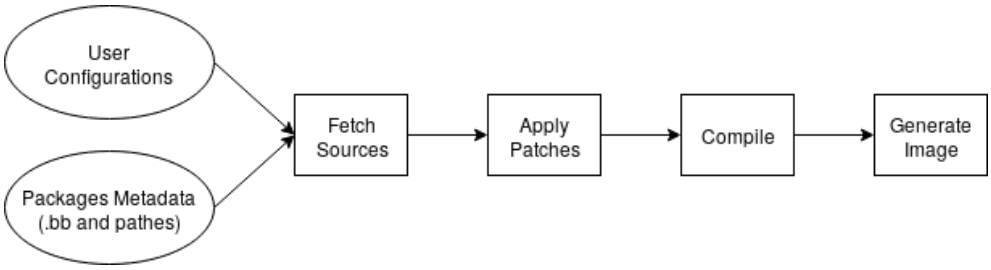

Figure 4.7: Yocto Build Process

CHAPTER 5

Testing Results

This chapter describes the procedures and presents the results of a couple of the tests that were performed on the developed images.

In other to validate the usability and performance of the two implemented images, some tests have been performed, that can be mainly divided in two parts:

Firstly, even though the majority of the board components were tested when developing the images, as mentioned in Section 3.3, it was required to create a single application that would integrate the maximum number of tests needed to validate the components of the board. The reasoning for this application was twofold. One, to verify that all the components were working correctly in a standardized way, and two, in a later phase, in the production line of the board, to validate if the boards are assembled correctly.

Secondly, we subjected the built images to a field test, and recorded performance metrics for an extended period of time, in order to try to quantify differences between them.

Other tests were performed, namely a temperature chamber test that placed the board in a chamber with temperatures ranging between -40 Celsius to 85 Celsius, while performing CPU intensive tasks. This was done to validate the working temperatures of the board components. However, as these tests were board-oriented, and not image-oriented, they are not presented here.

5.1 Automatic peripherals validation

As previously mentioned, a custom application was developed to validate the components of the board, which will later be used in the production line to validate the assembled boards.

To complement the test, a breadboard configuration was assembled, which allowed us to standardize the test. Later this breadboard configuration will be fully built as a Printed Circuit Board (PCB) test jig.

The test utility, which was named WTU, was developed in Python3. This programming language was chosen for several reasons. First, because it is a programming language that

we had prior experience with, it supports Object Oriented Programming (OOP), and it has a faster development time compared to other high-level languages such as Java or C/C++. Also, Python3 has a multitude of libraries that help using it as a scripting language, as well as to create an CLI, which was the intended interface for the final application. Finally, even at this early phase in the development, the team set a long-term goal of developing a library in a popular and user-friendly language that would enable the end-user to control the peripherals in the board. As such, this utility will be the building blocks for the future library to be developed.

5.1.1 Procedure

After the test utility and corresponding breadboard had been developed, the procedure used for the tests was to, one board at the time, connect the board to the breadboard test jig, and then run the test utility to collect the results, which were printed to the console. Figure 5.1 shows the output of the Wi-Fi test.

```
root@wiipiido_dev:~/wtu/wtu# python3 cli.py io-test
Staring IO Tests...
INIT [ACCEL] DISABLED
INIT [ADC] DISABLED
INIT [Ethernet] DISABLED
INIT [GPIO] DISABLED
INIT [GPS] DISABLED
INIT [NBIOT] DISABLED
INIT [OneWire] DISABLED
INIT [PWM] DISABLED
INIT [RTC] DISABLED
INIT [SInfo] DISABLED
INIT [USB_SPI] DISABLED
INIT [USB_SPI_GPIO] DISABLED
INIT [USB_UART] DISABLED
INIT [WIFI] OK
WIFI: SUCCESS

^CStopping Application...
FINISH [WIFI] OK
Stoped.
root@wiipiido_dev:~/wtu/wtu#
```

Figure 5.1: WTU Output

Once the automated tests were executed, and the corresponding results recorded, some manual tests were performed. The manual tests consist in validations which are not included in the developed utility, and generally require human intervention for the test to be performed and/or validated. The executed manual tests comprise in:

- Connecting the High-Definition Multimedia Interface (HDMI) to a monitor and verify if it works correctly
- Connecting a speaker to the audio port and play the audio test `speaker-test -Ddefault -c 6`
- Connecting a USB pen drive to each of the USB ports and validate the read/write functionality of the device
- Use the same test as previously pointed in the USB-OTG port

5.1.2 Test examples

Each type of peripheral device requires a specific testing procedure, a specific hardware, and specific testing code. It would be tedious and somewhat redundant to describe these details

for all the peripheral devices. So, here we describe the testing procedure and apparatus for just two of the devices.

The first example is the 1-Wire test, which is one of the simplest tests. It consists of connecting the 1-Wire interface to a 1-Wire temperature sensor, and collect the values from the sensor. This connection is demonstrated in the corresponding schematic, in Figure 5.2, where `PC4` is the 1-Wire Pin provided from the WiiPiiDo board.

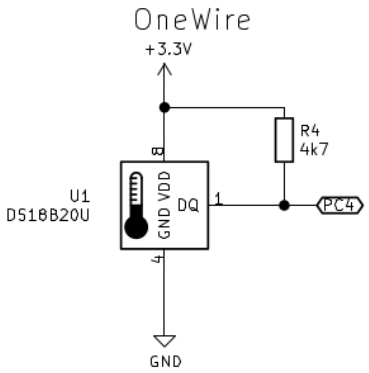

Figure 5.2: 1-Wire Test jig Schematic

Due to the `sysfs` interface for the 1-Wire, the test code in the application consists in reading the slave file for the 1-Wire interface, located at `/sys/bus/w1/devices/w1_bus_master1/<1-wire_identifier>/w1_slave`, and collecting the respective sensor values. As this reading had a significant delay, up in the seconds, and as the frequency of all the other tests is higher, the reading method for the 1-Wire is run in a separate process. The read data are then validated in the main application. An excerpt of the code for this test, the method that reads the sensor value, is shown in Code 27. The full code for this particular test is in Appendix A, in Code 31.

```
def read_wire(self):
    while self.running.value:
        try:
            tmp_str = WFile.readFrom(self.wire_file, readAll=True)[1]
        except Exception as exception:
            self.print_warning("File Read Error")

        self.read.value = int(tmp_str.split('=')[1]) # get just the temperature
        self.print_info(self.read.value)
        time.sleep(self.delay.value)
```

Code 27: 1-Wire Test Read Method

The second example is the USB-UART Bridge test. This test consists in reading/writing to/from the bridge, in a loop. The schematic for this test is shown in Figure 5.3.

In terms of code, we are once again using the sysfs interface to do this operations, iterating through the three devices provided by this bridge. The test code is presented in Code 28. Similarly to the last test, the full code is in the Appendix A, in this case located at Code 32.

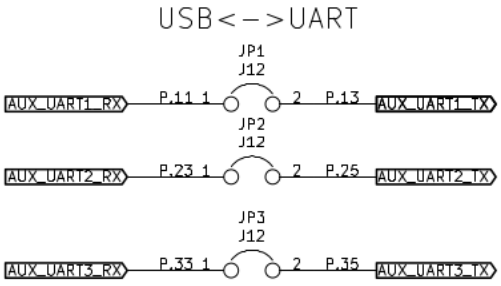

Figure 5.3: USB-UART Bridge Test jig Schematic

```
def test(self):
    if self.initialized():
        with serial.Serial('/dev/ttyUSB%d' % self.usb_dev, self.baudrate,
                                                timeout=self.timeout) as ser:
            # write
            self.print_info("USB%d - Write '%s'" % (self.usb_dev, self.test_string))
            ser.write(b'%s\n' % str.encode(self.test_string))

            # read
            line = ser.readline()       # read a '\n' terminated line
            line = line.decode()[:-1]  # decode back to string and remove '\n'
            self.print_info("USB%d - Read  '%s'" % (self.usb_dev, line))

            # test written and read strings
            if line == '':
                self.print_test("USB%d ERROR - Timeout" % self.usb_dev)
            elif self.test_string == line:
                self.print_test("USB%d OK" % self.usb_dev)
            else:
                self.print_test("USB%d ERROR - Strings are different" % self.usb_dev)

            # cycle through the usb devices
            self.usb_dev = self.usb_dev + 1 if self.usb_dev < 3 else 1
```

Code 28: USB-UART Bridge Test Method

5.1.3 Results

The results in this phase are summarized in two tables, Table 5.1 and Table 5.2. These represent the information provided from the manual test and the WTU, respectively.

Analyzing the tables we can verify that all of the automated tests were successful in both images, however, some of the manual tests failed, namely the Bluetooth test, for both images, and the Audio test for the Yocto image.

Component	Armbian	Yocto
Audio	Yes	No
Bluetooth	No	No
HDMI	Yes	Yes
USB	Yes	Yes
USB-OTG	Yes	Yes

Table 5.1: Manual Test Results

Component	Armbian	Yocto
1-Wire	Yes	Yes
Accelerometer	Yes	Yes
ADC	Yes	Yes
eMMC	Yes	Yes
Ethernet	Yes	Yes
GPIOs	Yes	Yes
GPS	Yes	Yes
NB-IoT	Yes	Yes
Pulse-width Modulation (PWM)	Yes	Yes
RTCC	Yes	Yes
USB-Serial Peripheral Interface (SPI) Bridge	Yes	Yes
USB-SPI GPIOs	Yes	Yes
USB-UART Bridge	Yes	Yes
Wi-Fi	Yes	Yes

Table 5.2: WTU Results

The Audio not working in Yocto is probably due to a missing kernel patch, as any configuration, DTS and firmware associated with this interface was checked to be similar to the ones used in the Armbian image.

Regarding the Bluetooth interface, the root of the problem still has not been detected. We are excluding the problem being an hardware anomaly as the Bluetooth is not a standalone module, but a combo module with the Wi-Fi, which is working. As such, we argue that the cause for the Bluetooth not being working is probably due to some missing firmware, or Bluetooth-related package missing or malfunctioning.

5.2 Field test

There was some discussion when deciding how the image would be compared in a final stage. The obvious answer that we came across was to deploy the application that the majority of boards will be running when deployed in the real world and do some field testing. To to so, a NodeJS application that enables a user to control and monitor devices that work with Radio Frequency (RF), was deployed to the test boards. As a way to get some sort of reference for the developed images, a Raspberry Pi 3 Model B[1] running Raspbian, the default and official image for this board, was also put to the same test.

5.2.1 Procedure

Three boards were prepared for this procedure, with the equipment and software summarized in Table 5.3. In each board, we deployed the respective OS, always choosing the smallest image when possible, i.e., for board P1, a minimal Yocto image was chosen, like shown in Section 4.2.1, for board P2, we chose the Armbian server image with minimal packages, as demonstrated in Section 4.1.1, and finally, for board P3 we chose the Raspbian Lite image.

[1] *Raspberry Pi 3 Model B* – `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`

ID	Board	OS	SoC	RAM
P1	WiiPiiDo	Yocto	Allwinner A64 (1.2 GHz)	2 G
P2	WiiPiiDo	Armbian	Allwinner A64 (1.2 GHz)	2 G
P3	Raspberry Pi 3B	Raspbian	Broadcom BCM2837 (1.2 GHZ)	1 G

Table 5.3: Test Equipment

Afterwards, we installed only the NodeJS application and its requirements. After verifying that the application was running correctly in all the boards, these were placed in the same room, which contained RF devices that were interacting with the boards. A test script was then deployed to each machine to gather performance metrics with a frequency of 6 times per minute, which are then saved locally. The metrics being monitored in this test were:

- Memory usage, with statistics from the `/proc/meminfo` file.
- CPU usage, containing the average usage during the test. This information was obtained using the `iostat -c` command.
- Number of running processes, which were retrieved by parsing the `/proc/stat` file.

A second script was also deployed to one of the P1 board, that synchronizes all of the gathered data once every 5 minutes.

Figure 5.4 illustrates the field test setup.

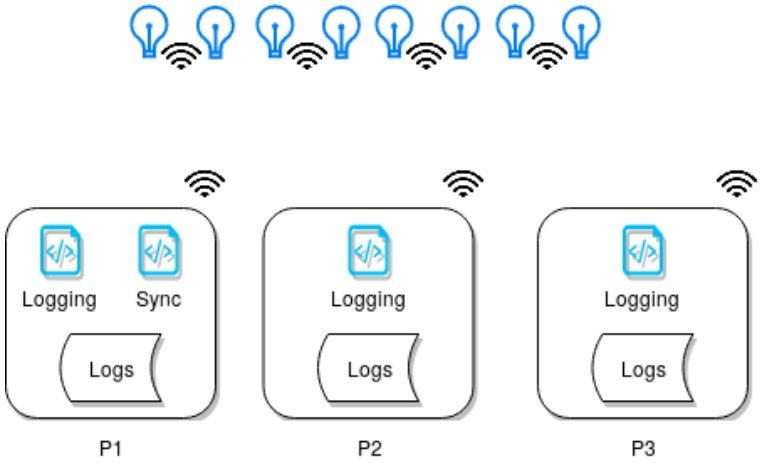

Figure 5.4: Field test setup

5.2.2 Results

The first results presented are the memory test results, as shown in Figure 5.5. The information included in these graphs contain the following data [38]:

- `Buffers` – The memory that is in use in buffer cache
- `Cached` – The memory in disk cache
- `Active` – The memory that has been recently used and that is not reclaimed unless being absolutely necessary.

Observing the graphs, we see that board P1 has the lowest memory usage from all the boards, and it stays almost constant during the full field test. Board P2 presents higher average values

compared with before, but similar behaviors for the Cached and Buffers memory. However the Active memory in this board possesses an aggressive usage, being reclaimed in definite periods of time. On the other hand, board P3 has the highest average values from all the tested boards, and a greater growth of the memory usage over time. This may be due to the fact that this board has half of the total memory compared with the other boards, and the OS is managing the memory differently.

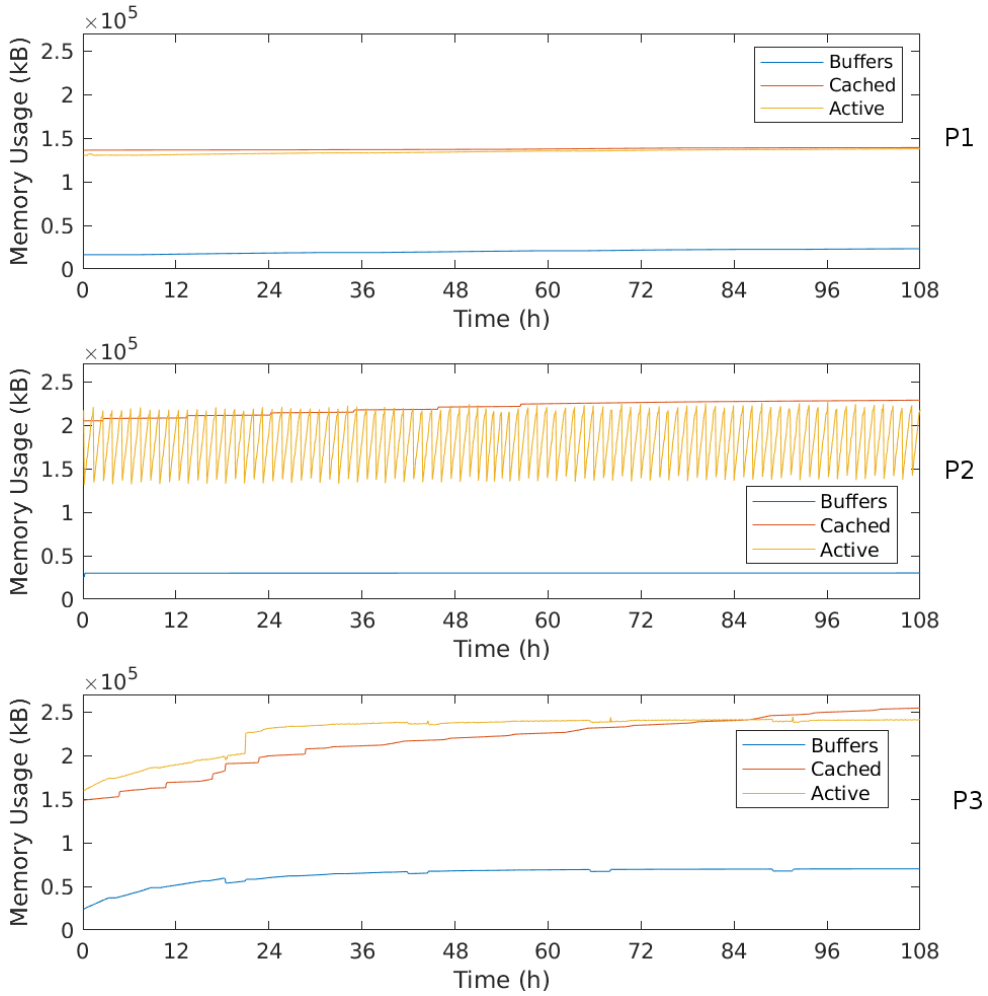

Figure 5.5: Memory usage from boards P1, P2 and P3

The second graph presented in Figure 5.6 contains the CPU performance test results. From this figure with can observe that board P1 and P3 had similar behaviors during the field test, with averages values around 0.6%. Board P2 however shows a completely different behavior, with increasing values over time, stabilizing around 28%.

Finally, Figure 5.7 illustrates the number of running processes during the test. From this figure we can see an average number of process for each image being between 52-60 for P1, 64-84 for P2 and 62-65 for P3. The different behaviors may be due to the different services running in background in each board by default.

To conclude, some of the results were as anticipated, such as board P1, the Yocto image, being the one using less resources overall. This was unsurprising seeing that this was the most

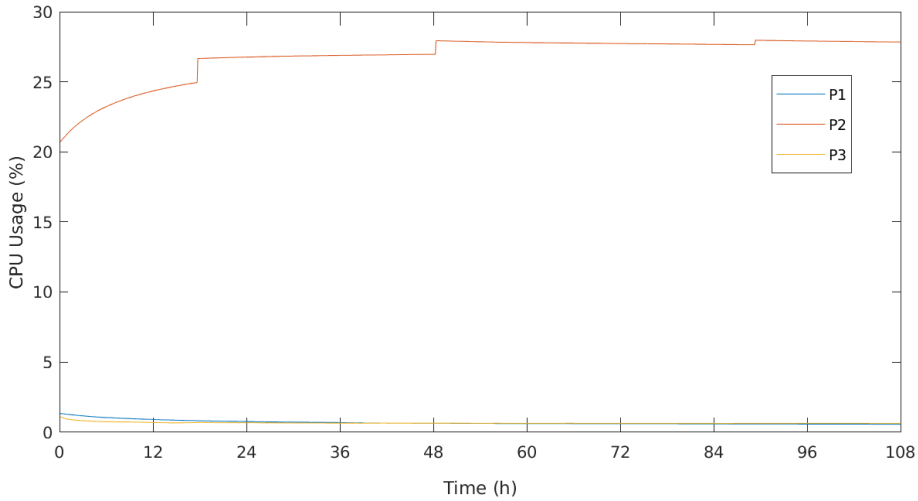

Figure 5.6: CPU usage

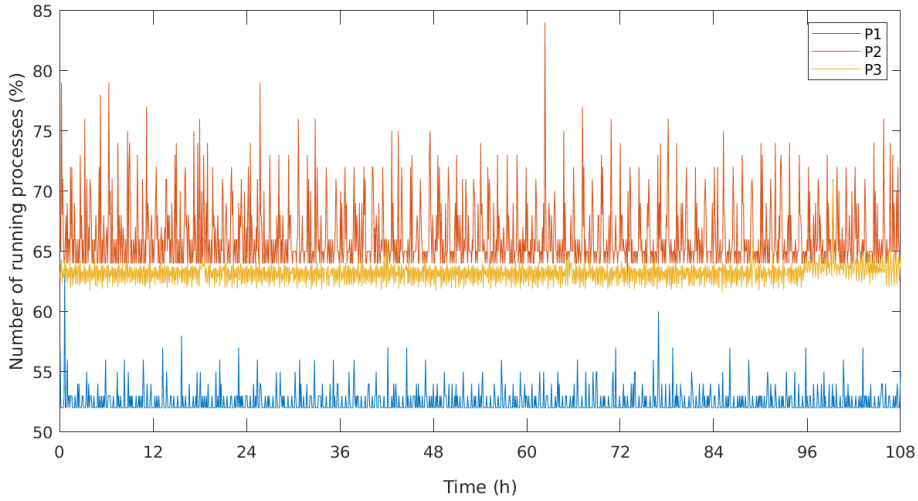

Figure 5.7: Number of running processes

compact image, with fewer installed packages and running services. Board P3 also showed expected results, with the default values from the tests being higher compared to board P1, and with slightly different behaviors. These differences are estimated to result from how the different OSs are managing the boards resources. That being said, the results provided by board P2 were not as expected, from the behavior of the Active memory, to the CPU usage during the test. The definite cause for these results was not found. However, we argue that the most likely cause was an hardware anomaly in the board that was used. This must be the case as, due to logistic reasons, the WiiPiiDo boards used in these tests were the first prototypes assembled, which presented a few problems in some hardware components.

5.3 Synthesis

In this chapter we performed several tests to validate the developed images. From these, we were able to confirm that the majority of the peripherals from the board are working correctly, with only two components not working at the time of this writing. There were also some

significant differences detected between images during the performance tests, that we believe may be due to hardware anomalies. Concluding, the performance test were inconclusive, with problems detected *a posteriori* that may be unrelated to the developed images, but we were not able to confirm this during this work.

CHAPTER 6

Conclusions

This chapter presents the conclusions, contributions and future work.

6.1 Conclusions

In this thesis we have described the development of custom Embedded Linux images that explore the full potential of the custom hardware platform WiiPiiDo. For that, two environments were used to build embedded Linux images. However, during the initial development phase, it was rapidly noted that starting to develop the final images from the get-go was not the best idea, since we had no experience in the environments that we had to use, and since these were the first images being developed for a new custom board that was not yet fully tested. In such cases, we argue that it is best to use a different approach, which allows us to safely boot images to the board, with an incremental level of complexity, minimizing possible errors that could occur. This also made the validation of the most important peripherals faster. As such, the used approach was divided in three phases:

1. Building of a simple bootloader, which allowed us to test mainly the core board peripherals, such as SoC, RAM and mass storage.
2. Building of a complete but minimalistic image, which allowed for fast compilation times, and rapid validation of some of the simplest peripherals from the board, such as the GPIOs, SPI, etc. This phase also provided the opportunity to integrate the tested peripherals in this environment.
3. Developing the final images in the intended environments.

From the final images that were developed, we concluded that both environments have their place in an new embedded hardware development, with distinct purposes that each is trying to fill. Armbian, being Debian-based, has the advantages and disadvantages of being a desktop-class Linux image in an embedded Linux system. Armbian is a familiar distribution for end-users, with access to the Debian Package repositories, and provides utilities to help the end-user to configure the image at runtime. This makes it more attractive as an image for

	Armbian	Yocto
Desktop-class image	+	–
Flexibility	–	+
Learning curve/Ease of use	+	–
Maintainability	–	+
Runtime configuration utilities	+	–
Community	–	+
Documentation and online resources	–	+
Supported architectures	–	+

Table 6.1: Summary of differences between the Armbian and Yocto build environments.

a multi-purpose user-controlled system. On the other hand, in situations where the target system is intended to have specific set of tasks, with a lower end-user control of the setup, Yocto is preferable. Yocto has more flexibility, allowing developers to create custom images tailored for their needs.

Another key difference between the environments is in how they work and their ease of use. Armbian will be easier to use if a developer is already somewhat familiar with embedded Linux development and shell scripting. Yocto, on the other hand, has a unique build system, which presents multiple concepts and configurations files that need to be understood by the developer when using the environment. This is responsible for a more difficult learning process, but on the other hand is what makes Yocto such a flexible environment, with a good collaborative development support. Additionally, the Yocto build system provides ways to freeze a layer to a specific version of the environment. This characteristic simplifies the maintainability.

Finally, although the community for both environments is vibrant and active, the Yocto community is larger, with more learning resources available.

Table 6.1 summarizes the main differences between the used build environments.

6.2 Contributions

In this work the following contributions were made:

- A comparative analysis of different Linux development environments (Section 2.5).
- Proposition of a thee phase approach to incrementally test and validate images in a new SBC (Chapter 3).
- Based on the experience obtained when integrating the peripherals, a procedure was proposed to help new developers start this process (Section 3.3).
- Creation and validation of the BSP Layer in Yocto for the WiiPiiDo (Subsection 4.2.2).
- Development of automatic scripts for board testing and validation (Section 5.1).

Part of this work was described in a paper accepted for publication in the Computing Conference 2020 [39].

6.3 Future Work

There is still a significant amount of work that could be conducted, which can be summarized in the following list:

- Complete the Armbian and Yocto support, namely, the integration of remaining invalidated devices.
- Finish the Yocto bitbake application recipes that provide the WTU and Globaltronic's proprietary software.
- Make all the developed work public and create pull requests to the official repositories, namely the Armbian, U-Boot and Kernel repositories.
- Evolve the WTU into a Python library that allows end-users to easily control the board peripherals.
- Continue maintaining the developed images.

APPENDIX A

Source Files

This appendix presents complete source files that are referenced in the thesis.

```
<ns0:RootFileSystem xmlns:ns0="https://www.linutronix.de/projects/Elbe"
                    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                    created="2009-05-20T08:50:56" revision="6"
                    xsi:schemaLocation="https://www.linutronix.de/projects/Elbe dbsfed.xsd">
    <project>
        <name>i386-stretch-grub</name>
        <version>1.0</version>
        <description>for testing 32bit with grub2</description>
        <buildtype>i386</buildtype>
        <mirror>
            <primary_host>ftp.de.debian.org</primary_host>
            <primary_path>/debian</primary_path>
            <primary_proto>http</primary_proto>
        </mirror>
        <suite>stretch</suite>
    </project>
    <target>
        <hostname>i386-stretch</hostname>
        <domain>elbe-rfs.org</domain>
        <passwd>foo</passwd>
        <console>ttyS0,115200</console>
        <images>
            <msdoshd>
                <name>sda.img</name>
                <size>1900MiB</size>
                <grub-install/>
                <partition>
                    <size>remain</size>
                    <label>rfs</label>
                </partition>
            </msdoshd>
        </images>
        <fstab>
            <bylabel>
                <label>rfs</label>
                <mountpoint>/</mountpoint>
                <fs>
                    <type>ext4</type>
                    <tune2fs>-i 0</tune2fs>
                </fs>
            </bylabel>
        </fstab>
        <finetuning>
            <rm>var/cache/apt/archives/*.deb</rm>
        </finetuning>
        <pkg-list>
            <pkg>linux-image-686-pae</pkg>
            <pkg>grub-pc</pkg>
            <pkg>xserver-xorg-video-radeon</pkg>
            <pkg>xserver-xorg-core</pkg>
            <pkg>xserver-xorg-input-all</pkg>
            <pkg>xterm</pkg>
            <pkg>isc-dhcp-client</pkg>
            <pkg>net-tools</pkg>
            <pkg>network-manager</pkg>
            <pkg>mono-runtime</pkg>
            <pkg>slim</pkg>
            <pkg>awesome</pkg>
            </pkg-list>
            </target>
            </ns0:RootFileSystem>
```

Code 29: Example of an minimal ELBE configuration file.

```
#@TYPE: Machine
#@NAME: WiiPiiDo Board
#@DESCRIPTION: An Allwinner based development board http://www.globaltronic.pt/en/product/wiipiido/

require conf/machine/include/arm/arch-arm64.inc

PREFERRED_PROVIDER_virtual/bootloader ?= "u-boot"
EXTRA_IMAGEDEPENDS += "u-boot"
UBOOT_MACHINE ?= "wiipiido_defconfig"
UBOOT_BINARY ?= "u-boot-sunxi-with-spl.bin"
UBOOT_ENV ?= "boot"
UBOOT_ENV_SUFFIX ?= "scr"
SPL_BINARY ?= "spl/sunxi-spl.bin"
UBOOT_ENTRYPOINT = "0x40080000"
UBOOT_DTB_LOADADDRESS = "0x4FA00000"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-wiipiido"
PREFERRED_VERSION_linux-wiipiido ?= "4.19%"
KERNEL_CLASSES = "kernel-fitimage"
KERNEL_IMAGETYPE = "fitImage"
KERNEL_DEVICETREE = "allwinner/sun50i-a64-wiipiido.dtb"

MACHINE_EXTRA_RRECOMMENDS += "kernel-modules linux-firmware-brcm43430"

IMAGE_FSTYPES += "wic"
WKS_FILE ?= "wiipiido-bsp-image.wks"

IMAGE_BOOT_FILES ?= " \
    fitImage \
    boot.scr \
    "

WKS_FILE_DEPENDS ?= " \
    mtools-native \
    dosfstools-native \
    virtual/bootloader \
    virtual/kernel \
    "

SERIAL_CONSOLES = "115200;ttyS0"
MACHINE_FEATURES = "alsa apm keyboard rtc serial screen touchscreen  \
                    usbgadget usbhost vfat ext2 ext3 wifi"
```

Code 30: Yocto BSP Board Configuration File Example.

```
from auxiliar.file_handler import WFile
from tests.wt import WT
import multiprocessing
import time

class OneWire(WT):
    def __init__(self, config=None, verbose=False):
        WT.__init__(self, runAsRoot=True, config=config, verbose=verbose)
        self.wire_file = WFile.find_files("/sys/bus/w1/devices/w1_bus_master1", "w1_slave")

        self.process = multiprocessing.Process(target=self.read_wire)
        self.read = multiprocessing.Value('i', -1)
        self.running = multiprocessing.Value('b', True)

        self.delay = multiprocessing.Value('f', 1)

    def parse_config(self):
        super(OneWire, self).parse_config()
        if not self.enabled:
            return

        current_config = self.configs['OneWire']

        if 'Delay' in current_config:
            tmp = self.parse_config_time(
                current_config['Delay'], 'Delay', dest_scale=1)
            if tmp > 0:
                self.print_info("Using 'Delay' [%fs]" % tmp)
                self.delay.value = tmp
            else:
                self.print_warning("Error parsing 'Delay'")
                self.print_warning("Using default 'Delay' [%fs]" % self.delay)

    def pos_config(self):
        if self.enabled:
            if len(self.wire_file) > 0:
                self.wire_file = self.wire_file[0]
                self.process.start()
            else:
                self.print_error("Device not Found!")
                self.start_success = False

    def read_wire(self):
        while self.running.value:
            try:
                tmp_str = WFile.readFrom(self.wire_file, readAll=True)[1]
                self.read.value = int(
                    tmp_str.split('=')[1])  # get just the temperature
                self.print_info(self.read.value)
            except Exception as exception:
                self.print_warning("File Read Error")
                self.read.value = NaN

            time.sleep(self.delay.value)

    def test(self):
        if self.initialized():
            if self.a.value is None:
                self.print_warning("%s" % self.read.value)
            else:
                self.print_test("%s" % self.read.value)

    def finish(self):
        if self.initialized():
            self.running.value = False
            self.process.join()
            super(OneWire, self).finish()
```

Code 31: 1-Wire Test Code

```
from auxiliar.logger import Logger
from tests.wt import WT
import serial

class USB_UART(WT):
    def __init__(self, config=None, verbose=False):
        WT.__init__(self, runAsRoot=False, config=config, verbose=verbose)
        self.usb_dev = 1
        self.test_string = "test"
        self.baudrate = 19200
        self.timeout = 1

        self.acceptable_baud = [9600, 19200, 38400, 57600, 115200]

    def parse_config(self):
        super(USB_UART, self).parse_config()
        if not self.initialized():
            return

        current_config = self.configs['USB_UART']

        if 'TestString' in current_config:
            tmp_str = current_config['TestString']
            if tmp_str.isalnum():
                self.test_string = tmp_str
                self.print_info("Using 'TestString' [%s]" % self.test_string)
            else:
                self.print_warning(
                    "'TestString' needs to be alpha numerical [%s]" % tmp_str)
                self.print_warning(
                    "Using default 'TestString' [%s]" % self.test_string)

        if 'BaudRate' in current_config:
            tmp_num = self.parse_config_number_in_range(
                current_config['BaudRate'], 'BaudRate', self.acceptable_baud)
            if tmp_num[0] == True:
                self.baudrate = tmp_num[1]
                self.print_info("Using 'BaudRate' [%d]" % self.baudrate)
            else:
                self.print_warning(
                    "Using default 'BaudRate' [%d]" % self.baudrate)

        if 'TimeOut' in current_config:
            tmp = self.parse_config_time(
                current_config['TimeOut'], 'TimeOut', dest_scale=1)
            if tmp > 0:
                self.print_info("Using 'TimeOut' [%fs]" % tmp)
                self.timeout = tmp
            else:
                self.print_warning("Error parsing 'TimeOut'")
                self.print_warning(
                    "Using default 'TimeOut' [%fs]" % self.timeout)

    def test(self):
        if self.initialized():
            with serial.Serial('/dev/ttyUSB%d' % self.usb_dev, self.baudrate, timeout=self.timeout) as ser:
            # write
            self.print_info("USB%d - Write '%s'" %
                            (self.usb_dev, self.test_string))
            ser.write(b'%s\n' % str.encode(self.test_string))

            # read
            line = ser.readline()       # read a '\n' terminated line
            line = line.decode()[:-1]  # decode back to string and remove '\n'
            self.print_info("USB%d - Read  '%s'" % (self.usb_dev, line))

            # test written and read strings
            if line == '':
            self.print_test("USB%d ERROR - Timeout" % self.usb_dev)
            elif self.test_string == line:
            self.print_test("USB%d OK" % self.usb_dev)
            else:
            self.print_test(
                "USB%d ERROR - Strings are different" % self.usb_dev)

            # cycle through the usb devices
            self.usb_dev = self.usb_dev + 1 if self.usb_dev < 3 else 1
```

Code 32: USB-UART Bridge Test Code

Referências

[1] Fortune Business Insights, *IoT Internet of Things (IoT) MarketReport*, https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307, [Online; accessed 23-October-2019], 2019.

[2] L. Atzori, A. Iera, and G. Morabito, «The Internet of Things: A survey», *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2010.05.010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128610001568.

[3] F. Samie, L. Bauer, and J. Henkel, «IoT Technologies for Embedded Computing: A Survey», Oct. 2016. DOI: 10.1145/2968456.2974004.

[4] Eclipse IoT Working Group, *IoT Developer Surveys*, https://iot.eclipse.org/iot-developer-surveys/, [Online; accessed 23-October-2019], 2019.

[5] Diana Olick, CNBC, *Just what is a 'smart home' anyway?*, May 2016. [Online]. Available: https://www.cnbc.com/2016/05/09/just-what-is-a-smart-home-anyway.html (visited on 11/23/2019).

[6] D. Molloy, *Exploring BeagleBone: tools and techniques for building with embedded Linux*. Wiley, 2019.

[7] Tutorials Point, *Embedded Systems - Overview*, 2019. [Online]. Available: https://www.tutorialspoint.com/embedded_systems/es_overview.htm (visited on 11/23/2019).

[8] Globaltronic, *WiiPiiDo*, 2019. [Online]. Available: http://www.globaltronic.pt/en/product/wiipiido/ (visited on 11/23/2019).

[9] Oron Peled, *Linux Programmer's Manual - Boot*, 2015. [Online]. Available: http://man7.org/linux/man-pages/man7/boot.7.html#top_of_page (visited on 11/23/2019).

[10] Kunal Singh, *An overview of Linux Boot Process for Embedded Systems*, 2008. [Online]. Available: https://www.embeddedrelated.com/showarticle/59.php#comments (visited on 11/23/2019).

[11] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2005, ISBN: 0596005652.

[12] G. Likely and J. Boyer, «A symphony of flavours: Using the device tree to describe embedded hardware», in *Proceedings of the Linux Symposium*, vol. 2, 2008, pp. 27–37.

[13] S. Barrett and J. Kridner, «Bad to the Bone: Crafting Electronic Systems with BeagleBone Black», *Synthesis Lectures on Digital Circuits and Systems*, vol. 10, no. 3, pp. 1–417, 2015.

[14] Embedded Linux Wiki Community, *Device Tree Usage*, 2019. [Online]. Available: https://elinux.org/Device_Tree_Usage (visited on 11/23/2019).

[15] A. Rubini, J. Corbet, and A. Oram, *Linux Device Drivers*, 3rd ed. O'Reilly Media, 2005, ISBN: 0596005903.

[16] J. Diamond and K. Martin, «Managing a Real-Time Embedded Linux Platform with Buildroot», in *Proceedings, 15th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2015): Melbourne, Australia, October 17-23, 2015*, 2015, WEPGF096. DOI: 10.18429/JACoW-ICALEPCS2015-WEPGF096. [Online]. Available: http://lss.fnal.gov/archive/2015/conf/fermilab-conf-15-432-ad.pdf.

[17] BuilRoot Community, *BuildRoot System Requirements*, 2019. [Online]. Available: https://buildroot.org/downloads/manual/manual.html (visited on 11/23/2019).

[18] R. Purdie, *Yocto Project Architecture Whitepaper*, 2009. [Online]. Available: https://wiki.yoctoproject.org/wiki/Yocto_Architecture (visited on 11/23/2019).

[19] The Linux Information Project, *Root Filesystem Definition*, 2006. [Online]. Available: http://www.linfo.org/root_filesystem.html (visited on 11/23/2019).

[20] U-Boot Community, *U-Boot Official Repository*, 2019. [Online]. Available: https://gitlab.denx.de/u-boot/u-boot (visited on 11/23/2019).

[21] Pine64 Community, *Pine A64*, 2019. [Online]. Available: https://www.pine64.org/devices/single-board-computers/pine-a64/ (visited on 11/23/2019).

[22] Trusted Firmware Community, *Trusted Firmware-A Documentation*, 2019. [Online]. Available: https://trustedfirmware-a.readthedocs.io/en/latest/getting_started/porting-guide.html (visited on 11/23/2019).

[23] S. Wood. (2013). TPL: SPL loading SPL (and, SPL as just another U-Boot config), [Online]. Available: https://www.denx.de/wiki/pub/U-Boot/MiniSummitELCE2013/tpl-presentation.pdf (visited on 11/23/2019).

[24] L. Osborn. (Nov. 2018). Using U-boot as production test strategy – really?, [Online]. Available: https://blog.asset-intertech.com/test_data_out/2018/11/using-u-boot-as-production-test-strategy-really.html (visited on 11/23/2019).

[25] U-Boot Community and Globaltronic, *Globaltronic WiiPiiDo U-Boot Fork*, 2019. [Online]. Available: https://github.com/Globaltronic/u-boot (visited on 11/23/2019).

[26] SUNXI Community, *FEL/USBBoot*, Aug. 2019. [Online]. Available: https://linux-sunxi.org/FEL/USBBoot (visited on 11/23/2019).

[27] ——, *FEL*, Mar. 2018. [Online]. Available: https://linux-sunxi.org/FEL (visited on 11/23/2019).

[28] ——, *A64 FEL Botting*, Nov. 2019. [Online]. Available: https://linux-sunxi.org/A64#FEL_booting (visited on 11/23/2019).

[29] BuilRoot Community, *BuildRoot Official Sources*, 2019. [Online]. Available: https://buildroot.org/download.html (visited on 11/23/2019).

[30] BusyBox Community, *BusyBox: The Swiss Army Knife of Embedded Linux*, 2008. [Online]. Available: https://www.busybox.net/about.html (visited on 11/23/2019).

[31] Armbian Community, *Armbian Git Repository*, 2019. [Online]. Available: https://github.com/armbian/build (visited on 11/23/2019).

[32] ——, *Armbian Documentation*, 2019. [Online]. Available: https://docs.armbian.com (visited on 11/23/2019).

[33] Yocto Project contributors, *Yocto Project Poky Repository*, 2019. [Online]. Available: https://www.yoctoproject.org/software-item/poky (visited on 11/23/2019).

[34] S. Lengfeld, *Blog – Yocto Recipes vs Packages*, 2017. [Online]. Available: https://stefanchrist.eu/blog/2017_09_15/Yocto%20Recipes%20vs%20Packages.xhtml (visited on 11/23/2019).

[35] Yocto Project Contributors, *Yocto Project Development Tasks Manual*, 2019. [Online]. Available: https://www.yoctoproject.org/docs/2.7.1/dev-manual/dev-manual.html (visited on 11/23/2019).

[36] Yocto Project contributors, *Yocto Project Quick Build*, 2019. [Online]. Available: https://www.yoctoproject.org/docs/2.7.1/brief-yoctoprojectqs/brief-yoctoprojectqs.html (visited on 11/23/2019).

[37] OpenEmbedded and Yocto Project contributors, *OpenEmbedded Layer Index*, 2013. [Online]. Available: https://layers.openembedded.org/layerindex/ (visited on 11/23/2019).

[38] Geek Diary user 'admin', *Understanding /proc/meminfo file (Analyzing Memory utilization in Linux)*, 2019. [Online]. Available: https://www.thegeekdiary.com/understanding-proc-meminfo-file-analyzing-memory-utilization-in-linux/?PageSpeed=noscript (visited on 11/23/2019).

[39] D. Duarte, S. Silva, J. Rodrigues, S. Soares, and A. Valente, «Comparison of Embedded Linux Development Tools for the WiiPiiDo distro development», in *Advances in Intelligent Systems and Computing*, (Accepted), Springer, 2020.