Creating tailored OS images for embedded systems using Yocto

Rafael Araújo

UMinho | 2019

Universidade do Minho
Escola de Engenharia

Rafael Amaro Vieira Araújo

Creating tailored OS images for embedded systems using Yocto

December 2019

Universidade do Minho
Escola de Engenharia

Rafael Amaro Vieira Araújo

Creating tailored OS images for embedded systems using Yocto

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral

December 2019

Acknowledgements

I would like to thank Professor Jorge Cabral for giving me the chance to develop this project with his guidance. I would also like to thank Professor Matjaz Colnaric of Univerza v Mariboru who guided and helped me during my Erasmus experience in Slovenia where all the theoretical research was done. Finally, i would like to thank all the ESRG members for all their help and support.

A big thanks to my family that was always by my side. Thanks for all ”How’s the dissertation going?” even without having any idea about the content of my dissertation. And of course, thanks for the financial part. I hope to return the favour soon.

A huge thanks to Vlad for being my teammate on this journey since the very beginning. Thanks to my best friend Joana for always being present regardless of distance and being my biggest support in this journey. Thanks to Rafael for sustaining my nervous and angry attacks when something didn’t go as desired. Thanks to Bruno for always accepting to do something fun to relax. Thanks to my friend Veronica for cheering me on with her aunt memes and for all the deep conversations about life. Thanks to Tiago for the motivation to stay focused even in the wrong way. Thanks to all the members of ”Chat dos manos” for all the funny moments shared there.

So to all of you, I say thank you!

STATEMENT OF INTEGRITY

Resumo

É indiscutível que a tecnologia é cada vez mais importante nas nossas vidas, tendo sido possível assistir á sua rápida evolução nas últimas décadas. Como consequência os sistemas embebidos tornaram-se progressivamente mais importantes e presentes no quotidiano, sendo responsáveis pelo desempenho de tarefas da grande maioria dos dispositivos modernos, tanto a nível pessoal como industrial.

Definido por muitos como a combinação entre hardware e software (com o objetivo de desempenhar uma tarefa específica) os sistemas embebidos têm de possuir um sistema operativo adequado para tal. Estes sistemas operativos diferenciam-se dos SO regulares, pois apresentam um grande nível de customização sem pôr em causa a sua viabilidade e eficiência. Todavia, o conhecimento necessário para a utilização destas ferramentas é enorme, impossibilitando a sua utilização por grande parte dos consumidores.

O hardware presente num sistema embebido é limitado, tornando necessário a criação deste software com o propósito de garantir a eficiência dos recursos e tirar o melhor proveito dos mesmos. Para tal, é necessário escolher quais as funcionalidades do sistema que devem ser utilizadas e quais são desnecessárias e podem, por isso, ser eliminadas. Esta customização pode ser realizada por diversas ferramentas existentes no mercado como o caso do Yocto. Utilizando esta ferramenta, é possível que o sistema seja construido apenas com as caraterísticas necessárias para o cumprimento do propósito para o sistema final. A personalização deste sistema embebido através da ferramenta Yocto é realizado através de um menu de texto, alteração de ficheiros e shell scripts, o que pode não ser intuitivo para todos os utilizadores.

O tópico desta dissertação é a automatização de todo este processo, de modo a possibilitar a total customização de uma imagem para um sistema embebido, mas com uma linha de aprendizagem reduzida através de uma interface gráfica de fácil compreensão.

Palavras-chave: *GUI*, *Hardware*, Sistemas embebidos, Sistema operativo, *Software*, *Tailored OS*.

Abstract

There is no doubt that technology is more and more important in our lives and it has been possible to watch its rapid evolution over the last decades. As a consequence, embedded systems become progressively more important and present in everyday life. These embedded systems are responsible for the task performance of the vast majority of modern devices, both personal and industrial.

By many defined as the combination of hardware and software, to perform a specific task, embedded systems must have a suitable operating system to do so. These operating systems differ from regular OS as they have a high level of customization without compromising their viability and efficiency. However, the knowledge required to use these tools is vast, making it impossible for most consumers to use them. To facilitate its usage, an abstraction layer is provided through a GUI.

The hardware normally present in an embedded system is limited, making it necessary to create this software to ensure resource efficiency and make the most of it. This requires choosing which of the system features to use and which ones are not needed and can be eliminated. This customization can be performed by various tools available in the market such as Yocto. Through the usage of this tool, the system can be built only with the necessary characteristics to fulfill the desired purpose for the final project. The customization of this embedded system through the Yocto tool is accomplished through a text menu, file modification, and shell scripts.

The goal of this dissertation is to automate the entire process making it possible to fully customize an image for an embedded system but with a reduced learning curve through an User-friendly graphical interface.

Keywords: Embedded systems, GUI, Hardware, Operating system, Software, Tailored OS.

Table of Contents

List of Figures

List of Listings

Glossary

AMD Advanced Micro Devices.

AOT Ahead-of-time.

aufs Advanced multi-layered unification filesystem.

BSP Board support package.

CD-ROM Compact disc read-only memory.

CLI Command-line interface.

COSMOS C Open Source Managed Operating System .

CPU Central processing unit.

DDR SDRAM Double Data Rate Synchronous Dynamic Random-Access Memory.

DVD Digital optical disc.

GNOME GNU Network Object Model Environment.

gpio General-purpose input/output.

GPU Graphics processing unit.

GUI Graphical user interface.

HDMI High-Definition Multimedia Interface.

I/O Input/Output.

IoT Internet of Things.

IP Internet Protocol.

IPC Interprocess communication.

ipkg Itsy Package Management System.

LFS Linux From Scratch .

LXDE Lightweight X11 Desktop Environment.

MIPI Mobile industry processor interface.

MIPS Microprocessor without Interlocked Pipelined Stages.

MMC MultiMediaCard.

OBS Open Build Service.

OS Operating System.

PPC PowerPc.

QEMU Quick EMUlator.

RAM Random-access memory.

ROM Read-only memory.

SDHC Secure Digital High Capacity.

SDK Software development kit.

SoC System on chip.

SPI Serial Peripheral Interface.

SSH Secure Shell.

UART Universal Asynchronous Receiver/Transmitter.

USB Universal Serial Bus.

VFS Virtual file system.

VS Visual Studio.

Chapter 1

Introduction

In an increasingly technological world, embedded systems play an important role and are present everywhere. Although the abundance of these systems in our day-by-day life, their usage can be unnoticed. For example, the presence of the embedded systems in a microwave provides their use without realizing that they were essential in making the meal. The tendency is for their importance to increase in our daily lives due to their low cost and size. However, a problem arises due to the increasing number of different systems: each one is different from the previous one and new software is required for its proper functioning.

To be able to see the evolution that has occurred in the embedded systems in recent years we can go back to 1974 where the first general purpose Microprocessor family was announced. Almost 50 years later, it is possible to find a microprocessor in the most common everyday objects. With the increasing appearance of new embedded systems, the same evolution on the part of the software is required. This growth of embedded systems cannot be supported by General-Purpose Operating Systems because these are not compatible with the embedded environment, creating the need for embedded software development at the same pace.

With that said, it is possible to conclude that an embedded system consists of two subsystems: **Embedded Hardware** and **Embedded software**. Both are equally important because it is not possible to operate any embedded system without the coexistence of both. The embedded systems have a great number of advantages such as small size and low cost. Unlike the typical OS is most used to (e.g. computers), these systems are designed to accomplish a specific task more efficiently, spending less resources. However, there are some disadvantages with the use of these systems. The developer needs to have a deeper knowledge to work with them and even with the recommended knowledge it can be complicated to design or upgrade these systems. To take full advantage of the system some requirements are demanded from developers such as: **Performance**, **Efficiency**, **Reliability**, **Robustness**, **Safety**, **Security** and

Usability.

1.1 Contextualization

As the name suggests, embedded means that something is designed and built as an integral part of a system or device. An embedded system may be standalone or may be part of an even larger system. Because of all benefits, the number of sold microprocessors is constantly increasing leading to a growing need of the design of these systems.

As mention before the embedded systems are composed of two elements: **Embedded Hardware** which is composed by a microprocessor with electronic components on the periphery or a microcontroller that already contain every needed component. On the other hand, **Embedded Software** which allows operations and tasks to be performed on the previously mentioned hardware. The individual existence of each is not useful and there must be a coexistence of software and hardware to create an embedded system.

The creation of this software is complex and requires a deep knowledge of embedded systems, making this option unfriendly to the user. Despite the usage of tools that make the development possible in a more automated way, the available user-friendly options have not the best application in the embedded environment, and options for embedded systems implying a high learning curve and deep knowledge.

Yocto Project is one of the tools that assists the developer in creating custom Linux-based systems regardless of the hardware. This project provides a set of tools that embedded system developers can use to create tailored Linux images for embedded and IoT devices, or anywhere a customized Linux OS is needed. Still, as mentioned before, the usage of this instrument is not possible for all users. To make it easy to use, a graphical user interface can be used to remove all necessary knowledge and create an abstraction the process. With this software, it is be possible to create a minimal operating system with a simple choice of options via graphical user interface customized for the desired hardware. In this case, the supported architectures are Raspberry Pi family microcontrollers. The figure 1.1 represents an overview of the process involved in creating software for embedded systems.

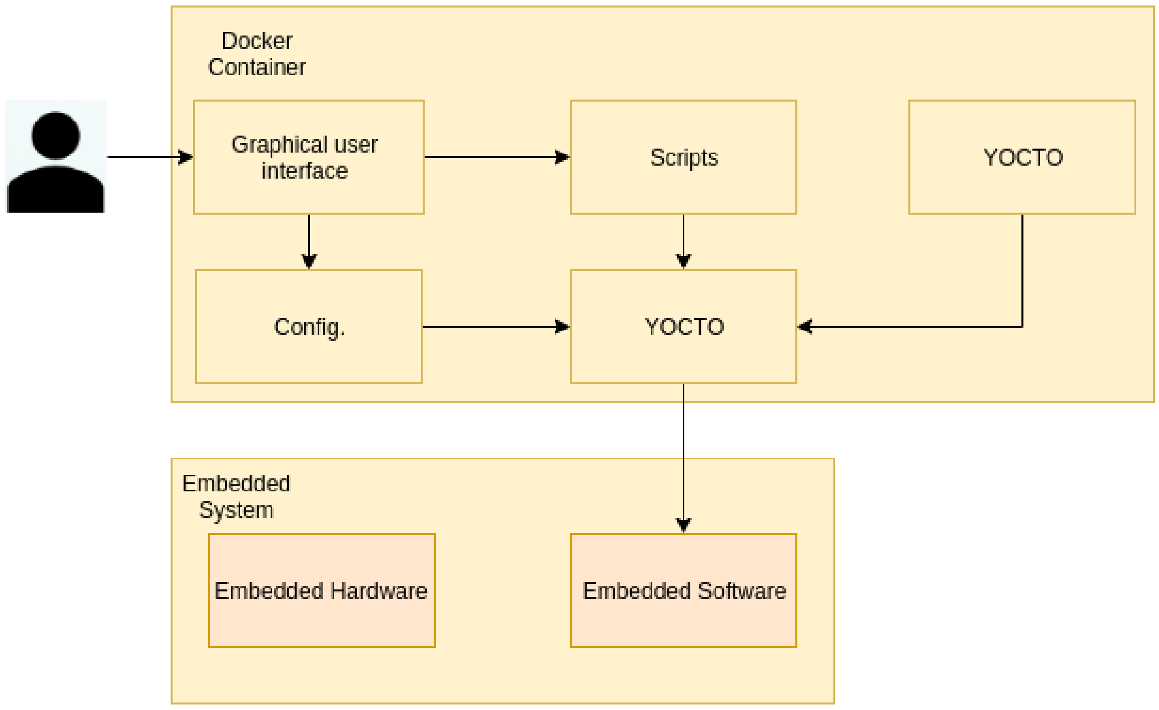


Figure 1.1: Process Overview.

1.2 Motivation

Everyday it is possible to find more and more different hardware combinations that arises the need for software to support them and to be created in a short period.

As the result of a proper study of the workflow of Linux-based operating systems, kernel and Hardware specifications it is possible to create an operating system from scratch. However, it is necessary to invest a large amount of time in this learning process to make it achievable option. It is also possible to create customized OS for embedded system without a steer learning curve by using available tools that provide a set of features that simplify the process. Yet, what if it was possible to reduce to zero the need for prior knowledge of all these concepts? A tool that creates an abstraction of all these processes comes to mind.

1.3 Objectives

Taking into account the motivation, the aim of this dissertation is the development of a tool that connects the user space with Yocto Project to create an abstraction layer. Therefore, the objectives of this dissertation to make this possible are as follows:

- In-depth study of the Yocto workflow.
- Study of kernel, busybox and Linux configurations.
- Study of Raspberry Pi family microcontrollers.
- Design and creation of the basic minimal image.
- Creation of scripts for host system preparation.
- Creation of a docker container.
- Creation of scripts to connect the Yocto Project and the customization software.
- Creation of a software that tailors the final OS.
- Creation of the final tailored images based on the minimal image through the customization software.
- Function test of new images.

1.4 Dissertation Structure

This document is split into six chapters and its structure follows a logical order for a simple understanding.

The first chapter introduces the basic concept behind this dissertation and a brief contextualization. Finally, it presents the motivation for the development of this project and its objectives.

The second chapter introduces the theoretical concepts to make possible an easy understanding of the project development. To make this possible a brief description of the kernel, Embedded Linux, Yocto Project and Shell scripts is provided to the reader. Finally, to show the importance of the concept and also the different interpretations of the problem it is presented the SOTA where several similar projects are showed.

The third chapter gives an overview of the system and a further selection of which components were used and the reason for their choices.

The fourth chapter is divided into three sections, corresponding to the creation of the base image, the creation of the GUI and finally the integration of everything with the docker. It focuses on how this project was developed and explains the path taken.

Chapter five shows the tests that were made, along with some considerations about the obtained results.

Chapter six presents the main conclusions relative to this project, as well as future improvements that can be made.

Chapter 2

State of the Art

The embedded systems are present in abundance in the market and they offer flexibility, efficiency and features like low-cost and low consumption. Embedded systems are the art of designing and choosing the right combination of software and hardware to achieve a final goal. On the embedded software side, it is necessary to create a software that fulfill the requirements of the user and achieve a certain goal.

In order to show the importance and relevance of this concept the next section will present different types of interpretations. Each one with similar goals and features for the reader to become familiarized with the concept and idea.

2.1 Basic Concepts

To make clear the implementation of this thesis, some basic concepts will be explained in the chapter.

2.1.1 Kernel

The kernel is a central component of an OS and does not interact directly with the user. However, it is the interface between the user applications and the hardware, including the processor, memory and disk drives. A computer does not need to have a kernel to work because it is possible to load and run programs directly on bare metal machines but it is usually not very practical. The critical code of the kernel is normally loaded into another area of memory protected from access by other applications or less critical parts of the OS. It has to be small as possible while still providing all the essential services needed by the other parts of the OS.

On most systems, it is one of the first programs loaded on start-up to be able the handle the rest of the start. The kernel provides the following services:

- Scheduling
- Dispatching
- Communication among tasks
- Tasks management
- Memory management
- I/O management

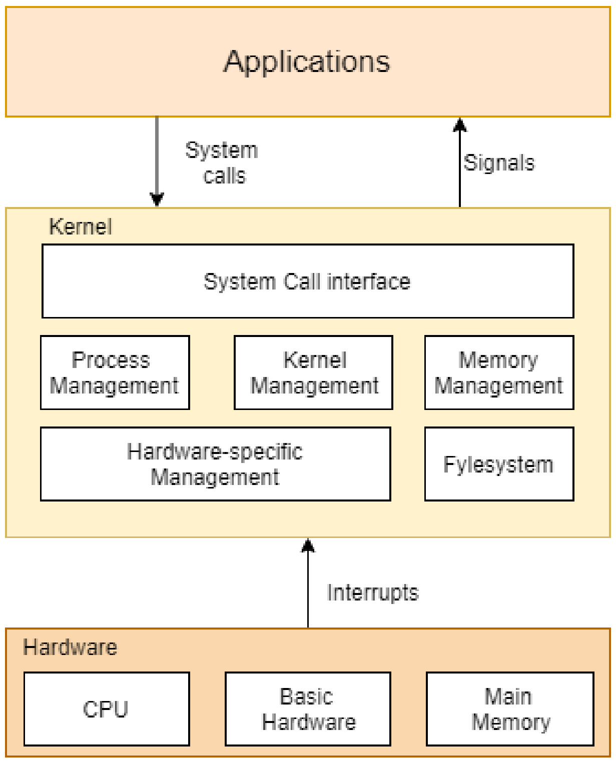


Figure 2.1: Kernel Overview.

It is possible to build a kernel in different ways when building from scratch. They can be classified into two types:

- **Monolithic**, contains all the operating system core functions, device drivers, file system management and system server calls. If the target device is supported, it works with no additional software. Modern versions of this type of kernel, like in Linux, can load modules to the kernel at runtime. This feature allows an easy extension of the kernel as required minimizing the amount of code running in kernel space. In monolithic kernels is easier to get access to hardware because if a program needs information it has a more direct line to access it and does not have to wait in a queue to get the information.

- **Microkernel** has the only purpose just to manage the basic hardware and much everything else in the computer can be handled in user mode. The main advantage is portability because they do not need to worry about a change of hardware like a video card or a keyboard. The footprint is smaller than in monolithic Kernels and they tend to be more secure because only specific processes run in user mode which does not have the high permissions as a supervisor. Another advantage of this kernel is security and stability, for example, if a basic service crashes only the service would be corrupted leaving the rest of the system still functional.

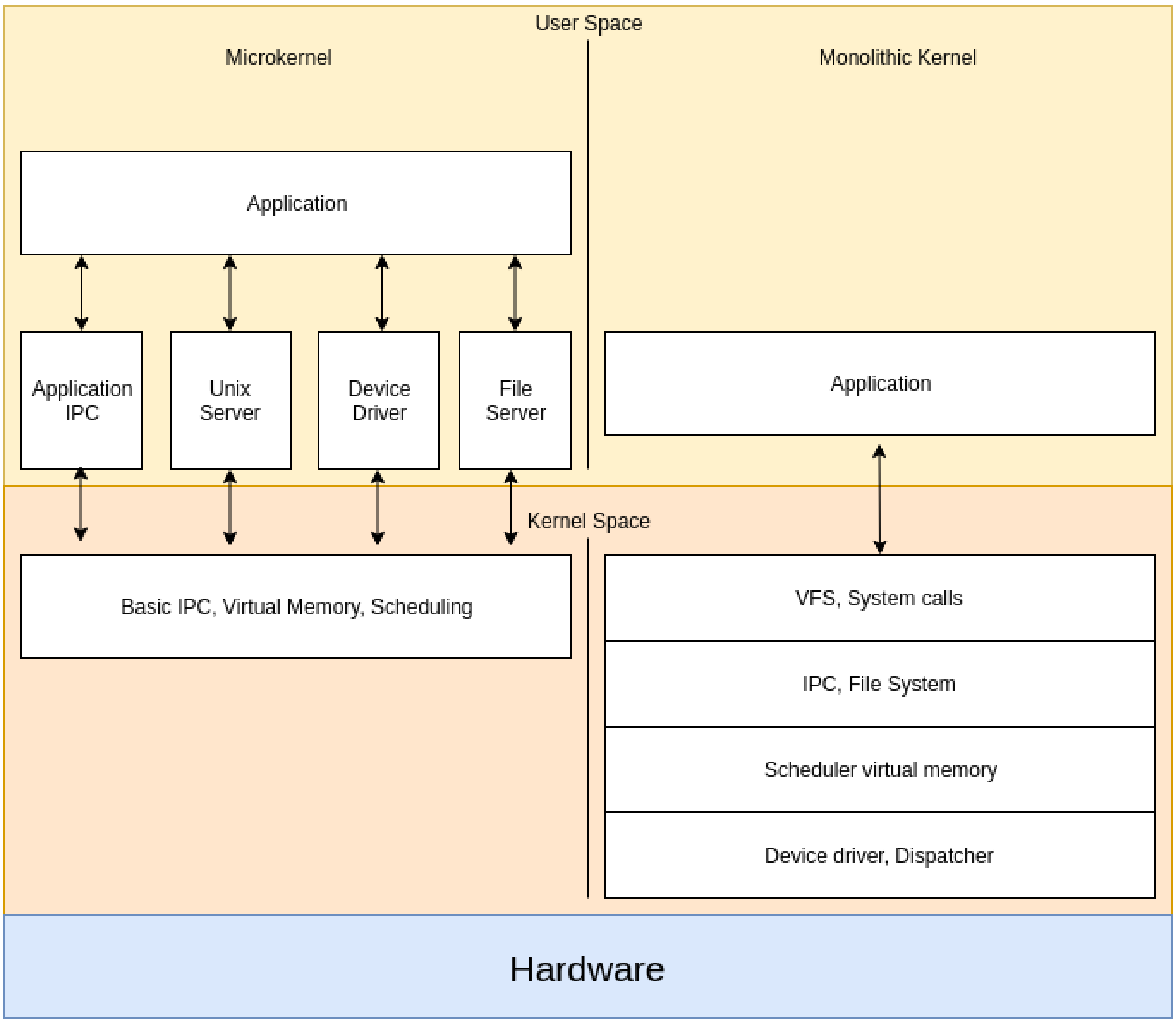


Figure 2.2: Microkernel vs Monolithic kernel.

2.1.2 Linux Kernel

Linux kernel is a large and complex Monolithic Kernel. This kernel provides an execution environment in which applications may run. To get the advantages of Microkernels without performance penalties the Linux kernel offers the capacity to load and unload some portions of the kernel code which are called modules. The main advantages of using modules are[1]:

- **Modular Approach**: This makes it easy to develop new modules because any module can be linked and unlinked at runtime.
- **Platform independence**: A module does not depend on a fixed hardware platform.
- **Spared main memory usage**: A Module can be linked to the running kernel when it is needed and unlinked when it is no longer useful. This feature is quite important for embedded systems.
- **No performance penalty**: Once linked in, the object code of the module is equivalent to the object code of the statically kernel.

The Linux kernel can be extremely small, not only because of its ability to dynamically load and unload modules but also because of its ease of customization. This ability to minimize the kernel has also led to rapid growth in the embedded world. The kernel configuration is saved in a file called *.config* in the top directory of the kernel source tree and can be created from scratch. The most basic method of configuring is to use the *make config* method. Linux kernel can be extremely small, not only because of its ability to dynamically load and unload modules but also because of its ease of customization.

2.1.3 Embedded Linux

Specifically customized for embedded systems, Embedded Linux is an open-source operating system with real-time constraints. it is designed for devices with a small size, less processing power and minimal features that aim attention to a very small and specific set of designated functionalities.

Following are some of the features of the embedded system:

- **Application Oriented**:

  - Runs a single program

  - The knowledge of the application behaviour can be used to minimize resources

- **Efficient**:

  - Lower power consumption

  - Low cost

  - Fast

  - Small in size and weight

- **Dependable system**:

  - **Reliability:** The system must work properly since start-up.

  - **Maintainability:** The system must work a few moments after an error occurrence.

  - **Availability:** The system needs to be ready for use when needed.

  - **safety:** Does not cause danger on the environment when fails.

  - **Security:** Provides communication confidentiality and authentication.

- **Dedicated user interface**

- **Hybrid system**, with analog part and digital part.

- **Environment interaction**, usually connected to the physical environment through sensors and actuators.

- **Reactive**:

  - The behaviour of the system has dependency on the input and current state.

  - Model state machine is usually appropriate

- **Timing Constrains**, the system must react to a stimuli from the controlled object or operator within the time interval dictated by the environment.

The most significant difference in comparing to desktop Linux is the fact that they have and require very few resources(e.g RAM, ROM, CPU capacity). Additionally, they are design from both hardware and software perspective, taking into account a specific application or functionality.

Following are some of the key differences:

- Linux kernel for embedded systems is configured differently. The set of device drivers and file systems that is needed differs in both. An embedded system may need a flash file system, unlike desktop Linux.
- Embedded Linux focuses on tools needed for development and debugging. The main target is given to a set of cross-development mechanisms that allow developers to build applications for their targets. Under other conditions, Desktop Linux focus is paid to a set of packages that are useful for users with an aim on productivity.
- Windowing and GUI interfaces differ in both systems.
- Single-user mode is the most used in embedded systems, unlike desktop Linux that has systems administration as an important requisite.

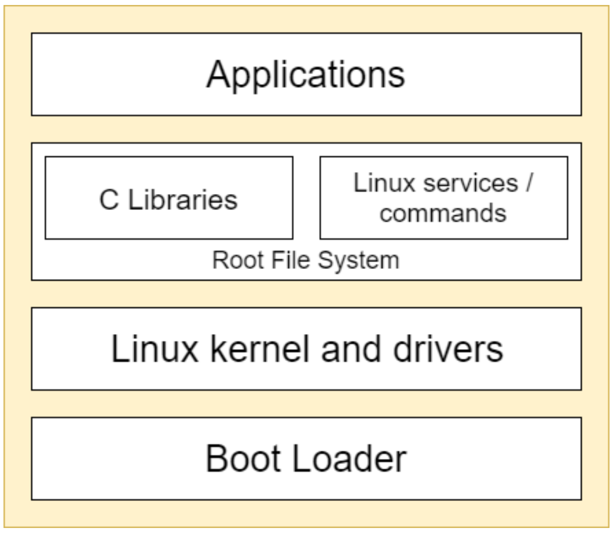


Figure 2.3: Embedded Linux overview.

2.1.4 Yocto

Yocto Project is not an embedded system Linux distribution itself. It is an open source project that provides templates, tools, and methods. This set of tools helps the developer to create custom Linux-based systems with the capacity and flexibility to fit requirements and constraints. Yocto is a collection of recipes, configuration files and dependencies that are used to create a tailored image based on Linux for specific needs. This tool allows the developer to change any configuration as needed. Important when the final target is an embedded system that must have full control over the software running. The following list represents features and advantages of the Yocto Project:

- **Yocto supports** a wide range of architectures(e.g Intel, ARM, MIPS, AMD, PPC) and it is possible to use and create a board support package for a custom architecture.
- **Layer model** is a development model for embedded systems and it is one of the biggest differences in yocto. This method allows at the same time collaboration and customization. Every layer contains sets of instructions that tell the Openembedded system what to do. Every layer can change previous instructions or settings allowing the customization of the project to fit the final product requirements. This method gives to the user the capacity of organization in layers simplifying future customization and reuse.
- **Flexibility** between developers. Because of the layer model, it is possible to work together on the same project. Working in different layers, a working group can create a distribution that fit in the final product needs.
- **Ideal for embedded devices** the user can use the yocto project to create the exact image with everything needed. It can be added or removed the feature or package that the developer wants.
- **Partial builds** can build or rebuild individual packages as needed being able to build and debug components individually.

2.1.4.1 Poky

Poky is the Yocto project reference and it is one of the projects under the Yocto supervision. This system is composed of tools and metadata to perform cross-compiling, using Bitbake and OpenEmbedded core with the main goal to provide all the features to build a tailored operating system. It is a combined repository of the components Bitbake, OpenEmbedded-Core, meta-yocto and documentation all together in one place.

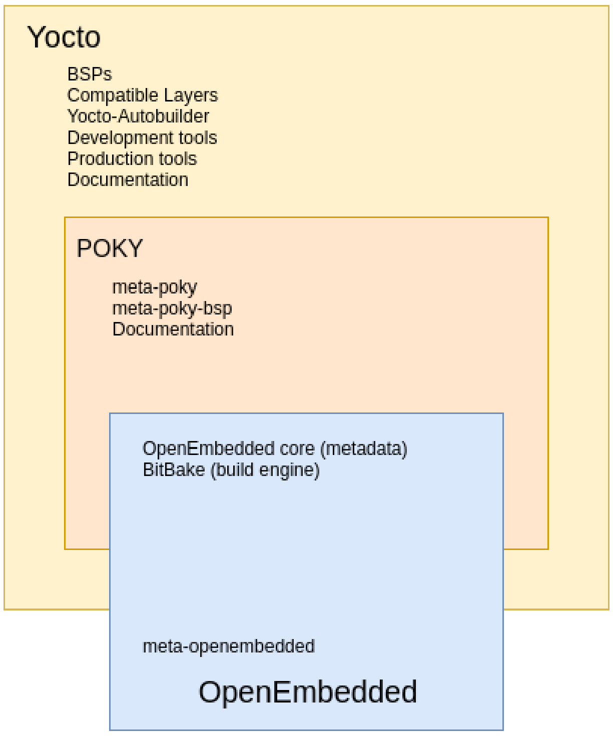


Figure 2.4: POKY overview.

Poky is a platform builder that generates filesystem images based on open source software with the possibility to boot inside a QEMU emulator adding the capacity to test the final image created in the host machine. One of the best features of poky is that every aspect of a build is controlled by metadata. Adding metadata layers is possible to extend functionality and features. These layers can provide additional software, add a new board support package or even create a new image type.

2.1.4.2 Open-Embedded

Open-Embedded is an open source build engine and a layer of metadata used by the Yocto project. OpenEmbedded tools use these recipes to fetch and patch source code, compile and link binaries to produce binary packages and create bootable images. This tool is the recommended build system of the yocto project but it is possible to use another mechanism.

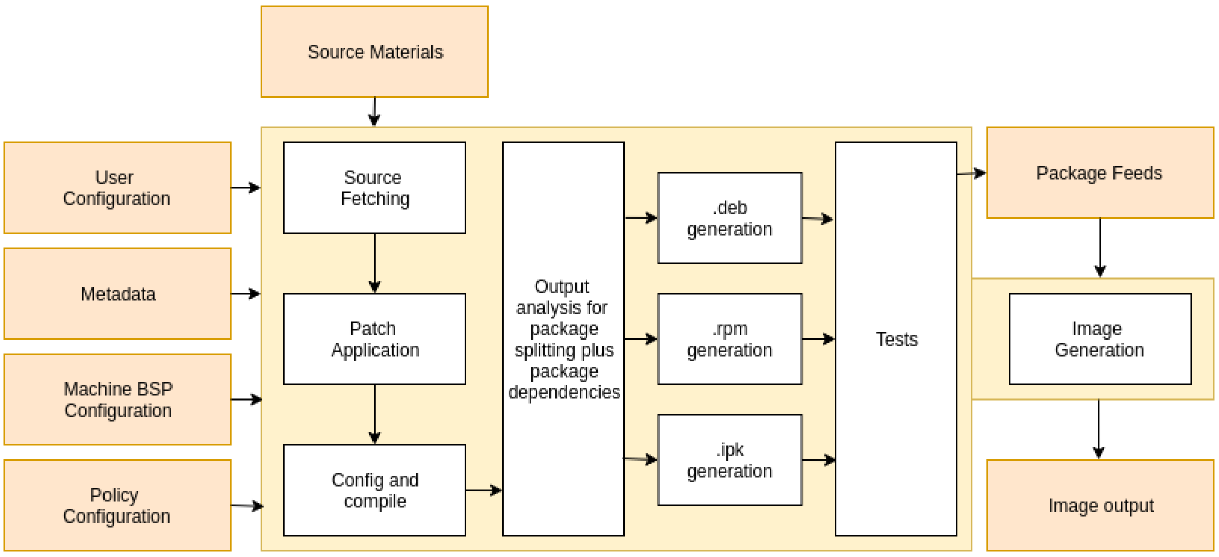


Figure 2.5: Open-Embedded overview.

The previous Figure:2.5, it represents the build overflow. Several functional areas are represented there: **User configuration** is metadata that is possible to use to control the build process. **Metadata layers** that provide software, machine and distro metadata. **Build system** is the process under the control of the bitbake. This process is represented by the yellow area and shows how to bitbake fetches source, applies patches, completes compilation, analyzes output for packages generation, creates and tests packages, generates images and generates cross-compile tools. **Package feeds** contains output packages that are used in the construction of an image produced by the build system. Finally, **images** are the final product produced by the entire workflow.

2.1.4.3 BitBake

Written in the Python language, BitBake interprets metadata, decides what task is required to run and execute those tasks. This metadata is stored in a recipe, recipe ”append” files, configurations files, underlying include files and in class files. This provides BitBake with information about what task to run and dependencies between tasks. Some requirements are handled cross-compilation, handle inter-package dependencies, support to multiple builds and target operating systems, be self-contained, handle conditional metadata, be easy to use and split metadata into layers but the main goal of this tool is flexibility and power. After this process, OpenEmbedded takes this core and builds embedded Linux software stacks using a task-oriented approach.

2.1.4.4 Metadata

Metadata is used to construct a Linux distribution and is contained in the files that OpenEmbedded parses to build an image. Metadata is composed of recipes, configuration files, and other information needed to control the built. Metadata also includes commands and data used to indicate the version of the software are used and other information about them.

- **Recipe** is the most common form of metadata and contains a list of settings and tasks for building a package. This information is used to build the binary image.
- **Configuration files** are files that hold global information about configurations. These files are used by OpenEmbedded to know what to build and what to put in the final image to support a particular platform.
- **Layer** is a collection of related recipes used to customize a build. They are hierarchical in their capacity to override previous specifications in previous layers.
- **Packages** in the context of the Yocto project are the compiled binaries made from the recipe's sources. This process is produced by Bitbake.
- **Board Support Package** is a layer that provides machine-specific configurations to a particular target architecture.

2.1.5 Shell Scripts

Shell scripts are files in which the developer can write a sequence of commands that the user needs to perform and are executed using the shell utility. The main goal of this concept is the possibility of automating a process.

Yocto makes use of shell scripts to define Yocto Project's build environment on the build host and to create the build directory which is "*build*" and is located in the source directory, using the script "*oe-init-build-env*". Later, when the build completes, this directory contains all the files created. Shell scripts are also used to make the connection between the Graphical user interface and the Yocto Project.

2.2 SO for Desktop systems

In this section, all solutions presented aim at developing operating systems for complex systems and with a high number of resources, unlike embedded systems.

2.2.1 COSMOS

Cosmos is an open-source toolkit for building an operating systems hosted at GitHub and has as pre-requisites Visual Studio, Inno and Net core. The ahead-of-time compiler named IL2CPU is useful to create a bootable operating system that can run without any support. This operating system can be booted from a USB Flash, CD-Rom or inside a virtual machine. Cosmos uses c# as primary language but many CLI languages are also available.(Figure2.6)[2]

Cosmos allows the user to create and boot an operating system in a short amount of time using visual studio. Cosmos also has an integrated debugger and can use breakpoints. Two versions of cosmos are developed: the developer kit and the user kit. The developer kit is made for users who want to work on the cosmos itself. On other hand, the user kit is designed to build an operating system through the cosmos. The user kit is better for beginners because it decreases the learning curve but moving later to the Dev Kit is advised.[3]

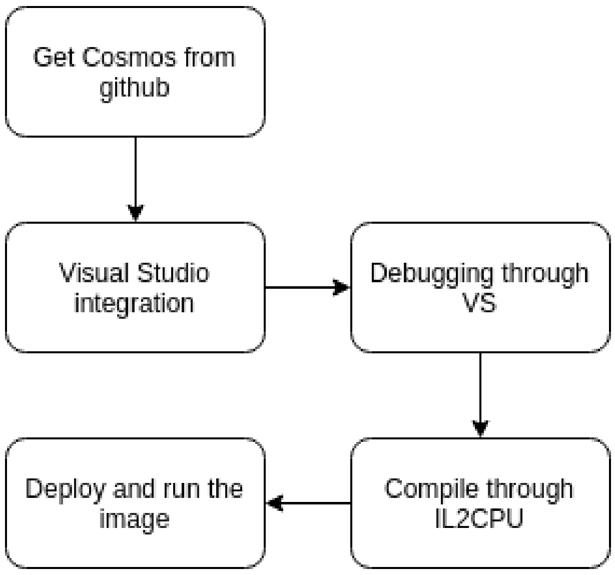


Figure 2.6: Cosmos workflow.

2.2.2 Linux Live Kit

Linux Live kit is a set of shell scripts which allows the user to create a Live Linux from an already installed Linux distribution. The operating system can be booted from a CD-ROM or a disk device. The tool is very perceptive and using it to build a live distro just require to follow these steps:

- Host with a distro installed to work with this tool.
- Check the compatibility of the advanced multi-layered unification filesystem and squashfs with the user kernel.
- Remove all unnecessary files to make the final output smaller.
- Download Linux live kit and edit .config file to configure the final operating system.
- Run the ./build script as root to create the live kit iso image.
- Run the ./bootinst.h to make a bootable device.

Linux live kit is useful to clone and backup an operating system but is not the best option for an embedded system because of the size of the output. It is a really good option to generate OS to a big amount of computers, for example, in a supermarket store.[4]

2.2.3 Desktop distros

This is not a project but a common approach to design an embedded Linux system. It starts with a desktop distribution and unneeded components are removed until the installed image fits in the requirements of the new target.(Figure 2.7) With this approach, the user has familiarity because of the similarity of the environment. This allow developers to get started more quickly. Another advantage of this method is the wide number of packages available. The number of packages available for most computer distributions is generally bigger than that available for embedded systems. The biggest disadvantage of this approach is that desktop distributions are not designed to run in low-resource systems.

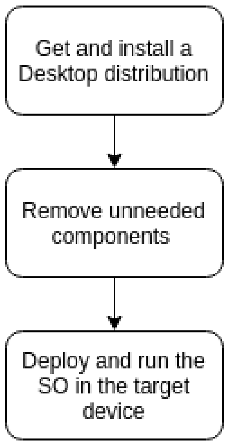

Get and install a Desktop distribution
Remove unneeded components
Deploy and run the SO in the target device

Figure 2.7: Desktop distros workflow.

2.2.4 SLAX Module Tool

Slax Module is a free ”pocket” operating system based on Debian that runs from an external media without any need for permanent installation making portability a strong factor to use this tool. It has easy modular customization and additional software can be added and removed using Slax modules. Slax provides an easy and fast way to configure and use an operating system in any device with a wide range of different filesystems. Despite its small size, Slax provides a nice graphical user interface and wise selection of pre-installed programs, such as a Web browser, Terminal, and more.[7]. Slax is based on Debian which gives the user the possibility to use a wide range of packages and applications provided by the Debian ecosystem. Regular updates are implemented by the founder and features can be requested by users. The user can add software to Slax manually instead of using ”apt” command. By downloading source codes from the internet it is possible to compile them using build-essential which provides GCC compiler and other tools necessary for that task. It is also possible to make the changes permanent if the user selects the persistent change feature during the boot. After this, it is only necessary to run the command ”save changes” to save any configuration or change in the operating system. A large number of commands are available to use and help the user to configure and change the operating system. Everything is explained on the official website described by the creator of this project.

2.2.5 Debian Live

Debian live project produces tools and frameworks to build live systems based on Debian. Official Debian Live images are produced with this tool. A live system is a preconfigured ready-to-use system in a compressed format that can be booted and used from a read-only CD or DVD. However, this live system has some disadvantages. The first is that all changes in the live system are saved in the computer RAM making it suitable only for systems with enough RAM for that. Another problem is the necessity of installation of non-free components manually if needed. [8] This project is also responsible for the maintenance of other packages that are used to build live images. A live install image provides a boot without modifying any file in the hard drive. The live image gives a wide range of choice of desktops environments(e.g GNOME, LXDE, Mate). It provides images for the two most popular architectures, 32-bit(i386) and 64-bit(amd64). The image does not contain a wide set of languages, making it necessary to recur to their installation. [9] A official live "Hybrid" ISO image is made available by the Debian live team and is suitable for writing directly to DVD or USB. This tool is a powerful tool but it is not the best solution to make a small and efficient operating system for embedded systems.

2.2.6 Remastersys/Linux Respin

Linux Respin is a fork of abandoned Remastersys and can create a distro based on Debian. It is a free and open-source software based on python and Bash. It is a simple way to create a tailored operating system. The first step is to download and install dependencies/packages needed for Respin(git and Respin). The second and last step is to run the respin and it is done. This application allows to make a clone or backup including root, other partitions and all personal data of the user and this is the first problem, it is not able to make an operating system for embedded systems.[14] This tool works in several distributions and requires a basic knowledge of Linux. A very friendly GUI makes everything easy to use and allows the user to create a bootable CD with packages and set already implemented ones very quickly. The user can choose between four actions: Backup complete system with user data, make a distributable copy to share with the creation of cdfs and iso, make a distributable copy filesystem only and make a distributable iso file only. In conclusion, Linux Respin is powerful and simple to use but it is not the solution for embedded systems.

2.3 SO for Embedded systems

Unlike the previous section, the presented tools have as main objective the development of operating systems for embedded systems.

2.3.1 Linux From Scratch

Linux from scratch is a project that provides step-by-step instructions to build a Linux system. It is a way to install a working Linux system from scratch in a compact and flexible form and grants a better understanding of the internal working of a Linux-based operating system. To keep it small and simple, a book called *Beyond Linux From Scratch* was written. The book has all the instructions to help the user to develop his OS. [5]
The user has lot of advantages of using Linux from scratch:

- The knowledge about how things work together and depend on each other and how to create a tailored image to suit the working environment.
- The capacity to build a very compact and small Linux system. It is possible to get an operating system within 5mb, which is not possible with a regular distro.
- It is extremely flexible. The user can customize and turn the operating system into whatever type of system depending on the needs.
- With the possibility to compile the entire system from the source, it is possible to apply all the security patches needed. It is not necessary to wait for someone else to provide a new binary package.[6]

However, building an LFS system is not a simple task. It requires a certain level of existing knowledge of the Unix system and the learning curve is huge for beginners. The lack of an interface makes this tool a non-friendly for the user. But putting some time and effort is possible to use this to make an operating system for embedded devices with high efficiency.

2.3.2 OpenWrt

OpenWrt is a project that allows the user to build an OS from scratch focusing on its customization and accessing all the features by Linux Kernel. Instead of a static firmware, OpenWrt provides a fully customized file-system with package management. This allows the user to use packages to customize an embedded device to support applications without any restriction. The framework provides everything needed to build an application without having to create a complete firmware image from the scratch.

This project aims to build and customize the software for embedded devices, especially wireless routers. Many algorithms are used to make it stable and reliable for long periods, to reduce lag and increase the network transfer rate. Older devices are still upgraded and supported by OpenWrt long after the manufacturer stopped the renovations. In the field of security, OpenWrt is an open-source software without any hidden backdoors, the regular amount of updates closed any vulnerabilities after they were discovered and the standard configuration is very conservative which permits connecting to internet without exposing information.

As a consequence of being a Linux-based system, OpenWRT grants full control over the device functions. The configuration is made through a command line or web-based user interface and all the information is saved in a file text. OpenWrt is one of the most used and with most information available. This makes one of the best options to make a Linux-based system form the scratch.[10]

2.3.3 KIWI/SUSE Studio express

KIWI is an application for making a wide range of images for Linux devices. It is developed by openSUSE Project and is a command-line tool having no graphical user interface that allows the developer to configure, build and deploy an operating system images in a variety of formats. There are two main steps to use KIWI:

- **Preparation** - Install the required packages from a software package manager, create an image description file and apply customization.
- **Creation** - The image creation process does not need user interaction but can be modified by changing the images.sh script that is called during the creation process.[11]

With no graphical user interface has a steep learning curve. Taking this in consideration, it is possible to confirm the importance of a graphical user interface to help a beginner to build an operating system. To make the system more perceptive and increase the target audience, the SUSE project created a web-based user interface. Having a browser is the only requirement of this application, not need any additional software. It allows to boot, configure and test a new operating system in a browser window without downloading anything. [12] SUSE Studio express is the final product of a merged OBS and the old SUSE Studio. This combined solution will give to the user:

- Collaboration on image building
- Open development

- Support for additional architectures
- Support for additional number of toolchains

It is also possible to import the existing kiwi files from SUSE Studio and export them into SUSE Studio express.[13]

2.3.4 Yocto

Yocto Project is an open source project whose goal is to produce tools and processes that enable the creation of an operating Linux-based system distribution for embedded systems. These tools allow the abstraction of the hardware layer improving and facilitating the job of customized Linux systems developers. A key part is the OpenEmbedded build system, which allows developers to create their own Linux distribution that is specific to their environment and their final product. The Yocto Project provides a reference implementation called Poky, which contains the OpenEmbedded build system plus a large set of recipes, arranged in a hierarchical system of layers, that can be used as a fully functional template for a customized embedded operating system as well as building Linux systems. There is also an ability to generate a toolchain for cross-compilation and a software development kit (SDK)tailored to their distribution. Thousands of developers worldwide have discovered that the Yocto Project provides advantages in both systems and applications development, archival and management benefits, and customizations used for speed, footprint, and memory utilization. The project is a standard when it comes to delivering embedded software stacks. The project allows software customizations and builds an interchange for multiple hardware platforms as well as software stacks that can be maintained and scaled.[15]

2.3.5 Toaster

Toaster is a web interface for the yocto project and for this reason is the most similar. The interface allows the user a quick configuration and run of the build through the yocto project. It is a good way to use yocto projects without a learning curve.

The toaster has two operational modes: Analysis and Build mode. in Analysis mode the build can be recorded and statistics are available. Using this mode, the user can use the bitbake directly to build an image. To use the analysis mode, the user needs to start the toaster and then to initiate the build using the bitbake command from the shell. Toaster must be started before the build to collect build data. In this mode, the following features are available:

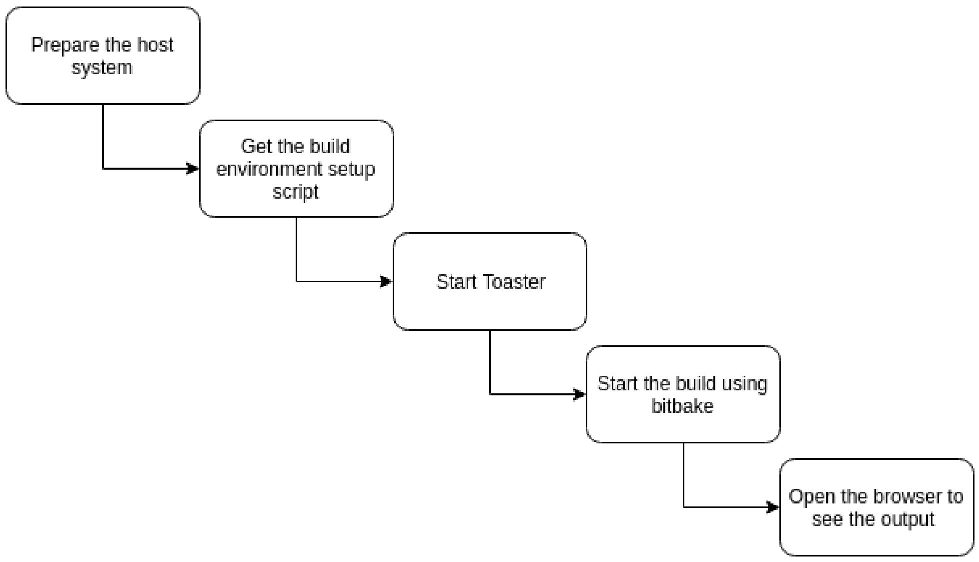

Prepare the host system
Get the build environment setup script
Start Toaster
Start the build using bitbake
Open the browser to see the output

Figure 2.8: Analysis mode workflow.

- See the final build, recipes and packages and what packages were installed into the final image.
- Browse the directory structure of the image
- See the content of all configuration files
- Debug the configuration build
- See information about the tasks executed
- See dependency between recipes, packages and tasks
- Information about performance

In build mode, the build configuration and execution are handled by Toaster. In this mode, all the interaction happens through Toaster and there is no need to run any shell command. Build mode has the same features of Analysis mode plus the following:

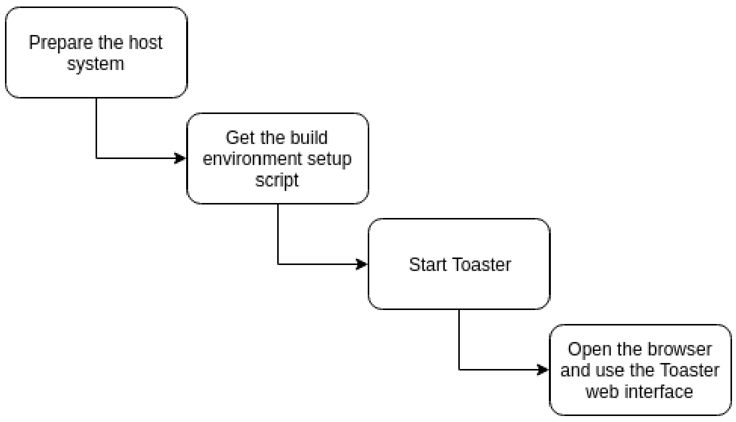

Prepare the host system
Get the build environment setup script
Start Toaster
Open the browser and use the Toaster web interface

Figure 2.9: Build mode workflow.

- Look for new layers in the repository of yocto
- Use user own layers for building
- Add and remove layers from the configuration
- Set configuration variables
- Select a target to build the image
- Start the build

This mode is, even more, user-friendly because everything is made through the web interface. The toaster can be set to run as a local machine or as a shared hosted service. Both modes are available in both configurations. In the local configuration, all the components are in the host machine. In other hand, when the Toaster is set to run as a hosted service, its components can be spread across several machines.[16]

2.3.6 Buildroot

Buildroot is an embedded Linux distribution, being currently one of the most widely used. It is not a new project and has been around for many years, firstly under the name of uClinux. Its first goal was to port the Linux kernel to processors without the MMU (memory management unit) but once it was released, the developer community quickly sprang out extending to newer kernels and different microprocessor architectures making it expand beyond the MMU-less systems. Eventually, it evolved into one of the most advanced and easy to use embedded Linux distributors. Taking a look at Buildroot, it is a tool that simplifies and automates the process of building a complete and bootable Linux-based environment for embedded systems. It is a set of makefiles and patches that, through the usage of cross-compilation allows the building for multiple target platforms on a single Linux-based development system. Buildroot can generate a cross-compilation toolchain, a root filesystem, a Linux kernel image, and a boot-loader for the user target and it can be used for any combination of these options, independently and is focused on simplicity and minimalism.(Figure2.10) [17] Buildroot builds all components from source but does not support on-target package management.

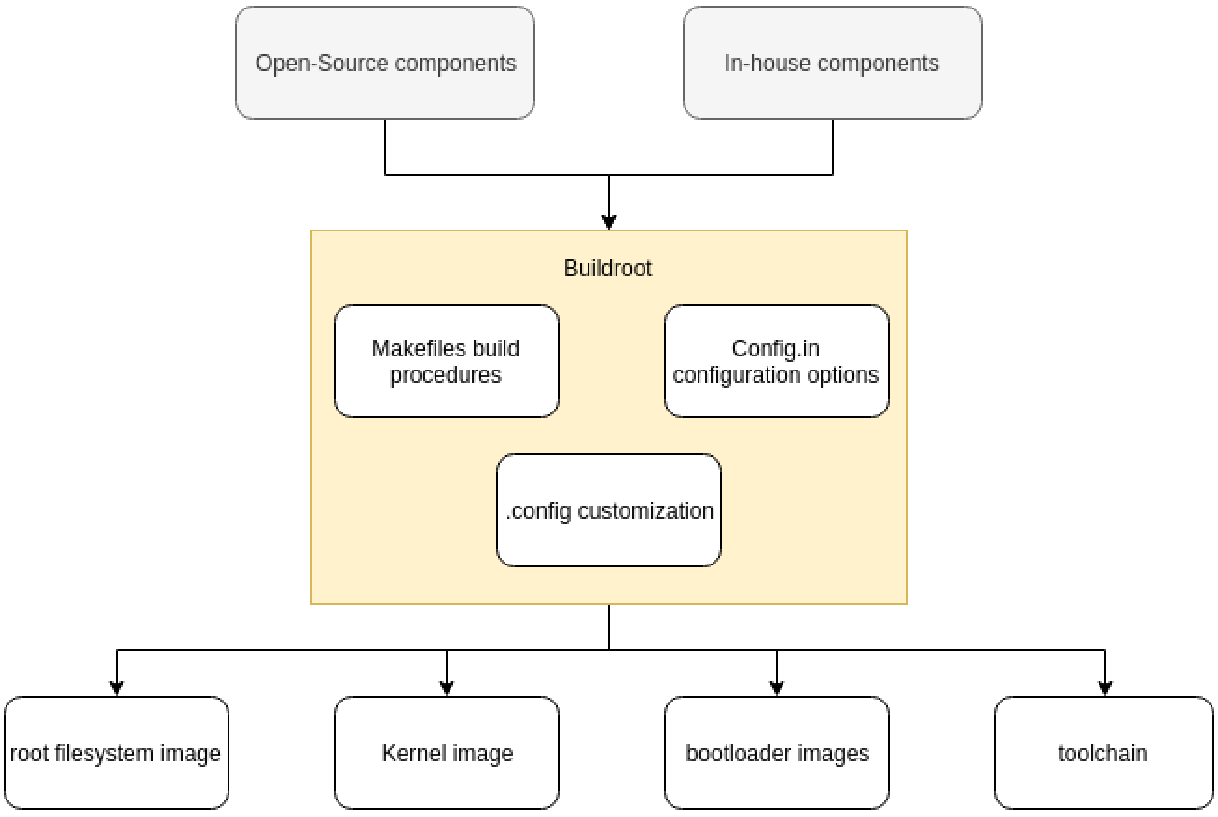


Figure 2.10: Buildroot workflow.

Buildroot output consist mainly in three components:

- The root filesystem image and any other auxiliary files needed to deploy Linux to the target platform
- The kernel, boot-loader, and kernel modules appropriate for the target hardware
- The toolchain used to build all the target binaries.

Buildroot has as advantaged the focus on simplicity is an easy to use tool. The core build is in make file and is short enough to allow the user to understand the entire system making process. Taking this into account, it is possible to expand via scripting editing the scripts. Buildroot users make files and recur to the kconfig language which is open source making it possible to find help online. However, the most important feature is the capacity to produce the smallest possible image. The focus on simplicity and minimal enabled build options imply that the user must do significant customization to configure a buildroot for a simple machine. The machine is said to be simple because all configurations options are stored in a single file which means that when using multiple platforms it is necessary to make a configuration for each one of them. Another problem is the time it takes to make a simple change to the system. This change requires a full rebuild of all packages turning a possible fast build in a long build while the user changes configurations. [17]

2.4 Conclusion

As it was possible to verify in the products presented previously, there is a great variety of mechanisms to assist in the creation of a tailored operating system. A wide range of options can be found with functionalities that provide the user with an opportunity to make an operating system in short amount of time. The problem is that this tools are not made and optimized for embedded systems but operating systems that require large resources are created by them.

On the other hand, it is possible to find systems that have focus on embedded devices having the goal to create tailored OS that do not require a large amount of resources. However, it is not a simple task that need a steer learning curve of Linux and embedded systems to use them.

Taking this into account, the application must have the possibility to create a small and compact operating system through a graphical user interface.

Chapter 3

System Specifications

After having the theoretical knowledge of the different technologies and aspects of this dissertation and having an overview of the different tools on the market, it is possible to define the components by which the system is composed.

That said, it is possible to present the design and specifications of the system and its architecture to demonstrate the main purpose of the developed application, as well as its features and its compatible hardware. Everything described above will be presented in the following chapter.

3.1 System Requirements

To design and conceive the final system, it is necessary to define all requirements and restrictions in advance. It is crucial to fulfill them in the system development process. With all the basic concepts previously presented, it will be possible to easily understand all the functional requirements and restrictions presented below.

- **Create embedded images for kernel, bootloader, and root filesystem.**
  The system must be able to create all images for an embedded system making its usage possible. Aspects such as output format and package format must be able to be changed by the user.

- **Tailor the embedded OS Image**
  The user will need to be able to customize operating system features to fulfill the system requirements.

- **Have a minimal image as a base for the operating system**
  All modifications made by the developer must be made on a base image with the basic and necessary features to make possible a normal operation of the embedded system.

- **Able to support the desired hardware and architecture.**
  Inside of the initially supported architecture, the system will need to be able to support the hardware and architecture specifically chosen by the user.

- **Be user friendly**
  The system must be able to create an abstraction layer between the user and the tools needed to create an embedded system. The user will only have to choose the necessary features for the final system through a graphical interface. The whole process behind this development will have to be hidden to the user.

Non Functional Requirements are essential to every program since they define the performance boundaries for the application. They ensure the usability and effectiveness of the entire system. With this in mind, the system will have as non-functional requirements:

- **Usability**
  The system must be user-friendly.

- **Portability**
  The application must be able to run on a large number of different operating systems.

- **Scalability**
  Although the main focus is Raspberry Pi, with simple system changes and modifications will have to be possible to adapt to another desired architecture.

- **Reliability**
  Through the process of obtaining the Yocto tool, the system must be updated.

- **Security**
  All aspects related to the security involved in the operating system is ensured by yocto. Any other aspect of security is out of scope of this dissertation.

3.2 System Architecture

As mentioned in previous chapters, the main goal of this dissertation is to create tailored Operating Systems for embedded environments that will later be used by developers to a specific project with specifics requirements. Taking into account all the specifications and requirements of this dissertation the study that was previously made in chapter 2, it is possible to define the system components. The system is divided into two major layers: the hardware layer, and the software layer. The full system stack, containing all the different layer divisions and elements are represented in the next figure: 3.1.

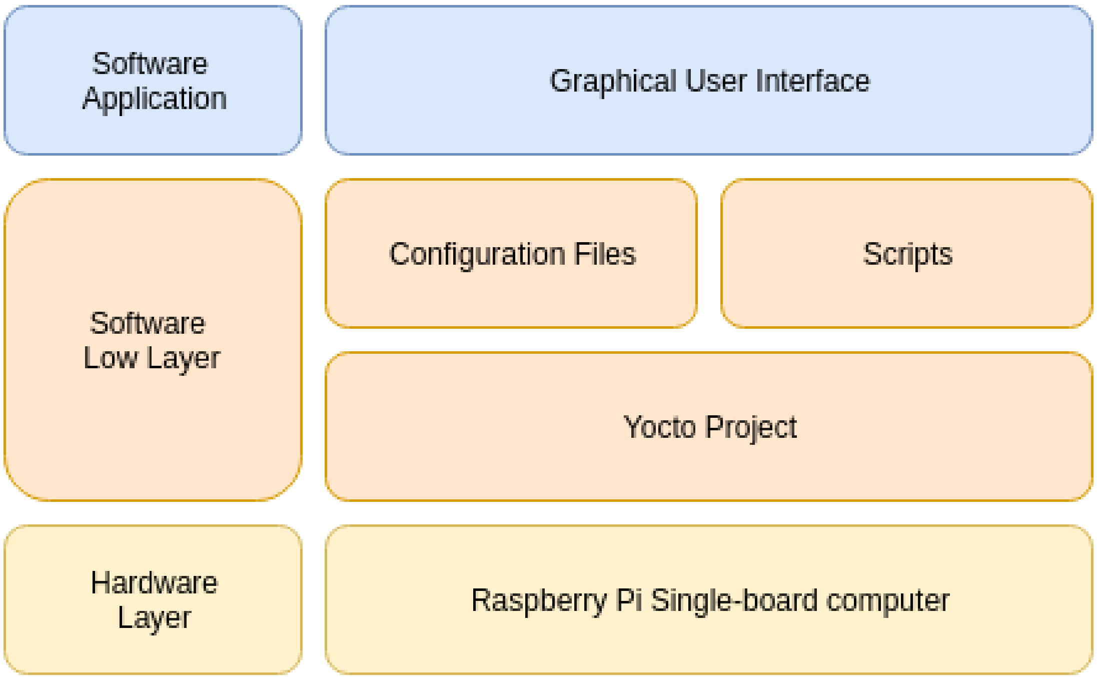


Figure 3.1: System stack, containing both the hardware and software layers.

The target platform is the main and only component of the Hardware layer. The choice of this board was studied in order to get the most used platform, increasing the importance and of this dissertation. Adding new single-board families can easily be done by adding the corresponding BSP.

The software layer can be divided into two sub-layers: the software low layer, and the application layer. The software low layer is responsible for the entire tailored image creation process using the Yocto Project tool. This layer is also responsible for communication with the software application layer. This is the last layer, where the user can make the customizing choices they want for their operating system in a user-friendly manner.

3.3 Hardware Specifications

After specifying all the system requirements and having defined the system stack and their different layers it is then possible to select the target platform to be used by this dissertation. This section is composed of the description and specifications of the target platform to be used.

3.3.1 RaspberryPi

After specifying the system requirements and architecture it is necessary define the hardware used in this dissertation. This hardware is not strictly necessary for the functioning of the concept of this dissertation, however, is required to be able to use the developed software. The Raspberry Pi is a series of small single-board computers and several generations of Raspberry Pis have been released, however, even though all versions are fully supported, three models will be further studied.

3.3.1.1 Raspberry Pi 3 model B

The Raspberry Pi 3 Model B is best known to the developer of these families. It can be used for many applications and maintains the format and design of the old platforms of this model. It has a Broadcom BCM2837 SoC with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor making this processor ten times faster than the one from the first generation and approximately 80% faster than the Raspberry Pi 2. It also has onboard 802.11n WiFi, Bluetooth and USB boot capabilities. With a small size (65x30mm), with Ethernet and wireless connection is very powerful for IoT systems. The 40-pin general-purpose input/output, it is useful to connect the platform to a large number of I/O devices. Some other features are presented in the list below:

- 1GB Low-Power DDR SDRAM;
- 4 USB 2.0 ports;
- 15-pin MIPI camera interface;
- High-Definition Multimedia Interface and Mobile industry processor interface display;
- Analog 3.5mm jack;
- MicroSDHC slot with USB Boot Mode;
- -20°C to +60°C temperature range.

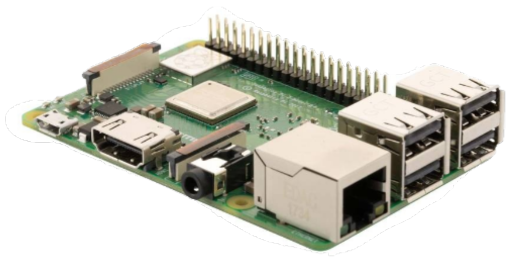


Figure 3.2: Raspberry Pi 3.

3.3.1.2 Raspberry Pi Compute Module 3

The Compute Module 3 is a Raspberry Pi in a format that allows for greater flexibility and customization of peripherals making this system suitable for an industrial application. With the same bases features as a Raspberry Pi3 (CPU and memory) and a 4GB eMMC flash memory in place of the memory card slot makes this platform the perfect choice for embedded systems. With the ability to achieve the performance of a Raspberry Pi but without all the unwanted peripherals it gives tremendous flexibility when designing new products. And since this board is fully customizable due to its flexibility, it is chosen as the ideal board for the entire final test of this dissertation. Some other features are presented in the list below:

- Broadcom BCM2837 quad core Cortex A53 processor @ 1.2 GHz with Videocore IV graphics processing unit;
- 1GB LPDDR2;
- 4GB eMMC flash;
- 48x GPIO;
- 2 I2C, Serial Peripheral Interface and Universal Asynchronous Receiver/Transmitter;
- -25°C to +80°C temperature range.

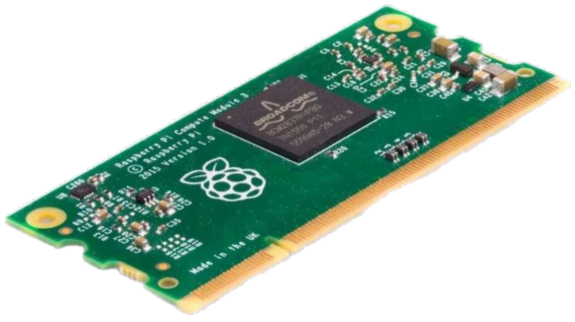


Figure 3.3: Raspberry Compute Module 3.

3.4 Software Specifications

After presenting all the hardware that will be used in this dissertation it is now presented the software that will be used later in the implementation. Unlike all the tools showed in the ”state of the art” previously, this chapter has as main goal to help understand the dissertation concept or basic concepts used. This software has as role of being the implementation process support platform.

3.4.1 QT Creator

QT is designed for the development of applications and graphical user interfaces on various desktop and mobile operating systems. Due to its easy use it is the ideal tool to be used. The easiest way to get started is simply to download it, which already contains all the libraries, examples, documentation and all the necessary development tools, such as QtCreator.

Qt Creator is an integrated development environment (IDE) and the version that was used is 5.12.3. This version provides the tools for designing and developing an application for multiple desktop, embedded, and mobile device platforms with the Qt application framework. It includes:

- **Managing projects:** To be able to build and run applications, QT creator needs information. All this information is stored in the project settings. To this end, the user is assisted by QT Creator which guides the user step by step through the process creation, producing the necessary files depending on the choices made.

- **Designing user interfaces:** QT Creator provides all the necessary tools for developing a GUI. With a Widget-based application, make it possible to create a modern and intuitive application using the Qt signals and slots mechanism that allows the user to easily assign behavior to graphical elements.

- **Coding:** The code editor is a key component of QT Creator. Making it possible to write, edit and navigate the developed code easily and intuitively. It understands the languages as code and not just a plain text. This gives the user useful features, such as semantic highlighting, checking code syntax, code completion, and refactoring actions.

- **Building and Running:** Qt Creator is integrated with cross-platform systems for build automation: qmake, Qbs, CMake, and Autotools. Besides, it is possible to import projects as generic projects and fully control the steps and commands used to build the project.

- **Testing:** Qt Creator is integrated to several external native debuggers: GNU Symbolic Debugger (GDB), Microsoft Console Debugger (CDB), and internal JavaScript debugger. In the Debug mode, it is possible to inspect the state of the application while debugging.

The reason for opting for this software for the dissertation development is for all the characteristics mentioned above. However, other more specific features were also important to make this choice. One of these features is the ”QProcess” library which provides the ability to create processes. Through this processes, it is possible to interact with the operating system and the Yocto Project. The possibility of using Signals and slots for communication between the graphical interface, code and the library ”QProcess” was essential too.

3.4.2 Docker

Docker is an open-source tool to make the process of creating, developing and running easier, using containers. A Container allows the developer to merge an application with all the parts it needs for its operation, such as libraries and other dependencies. By doing this, thanks to the container, the developer can ensure the normal operation of the application on different operating systems with different settings of the machine where the application was developed and tested.

With this in mind, the developer can focus on writing code without having to worry about the system that will run the application.

Containers and virtual machines have similar resource isolation and allocation benefits but they work in a different way. Containers virtualize the operating system instead of hardware, are more portable, efficient and will create an abstraction layer that contains all the files needed to run an application. On the other hand, Virtual machines create an abstraction of physical hardware and contain a complete operating system running. This difference is represented in the following image 3.4:

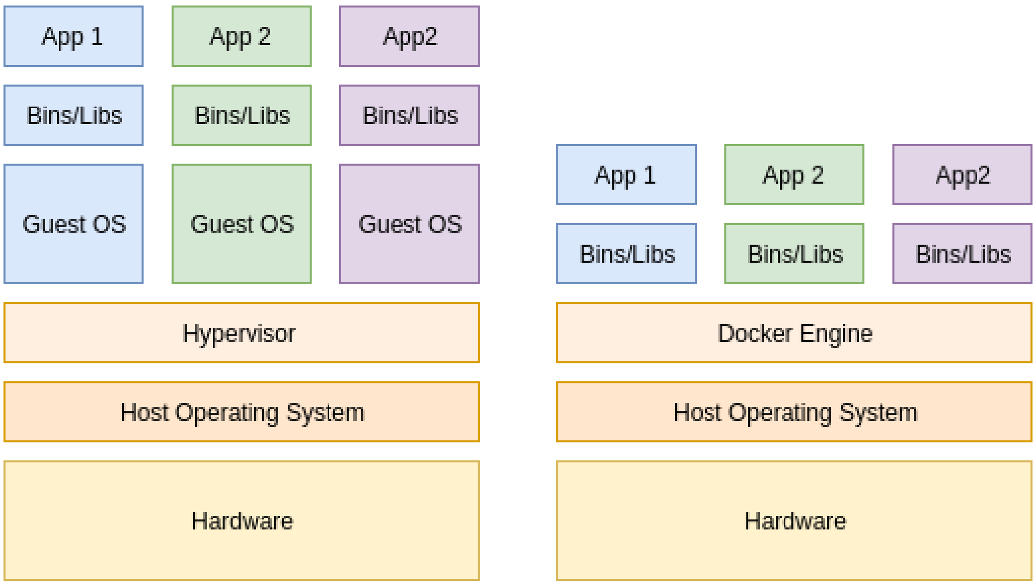


Figure 3.4: Virtual Machine vs Docker.

Using the docker ensures the **portability** constraint, allowing the final design to be run on any platform or environment. This tool also isolates the project from any external problem as compatibility. Under these circumstances increasing its **reliability**.

3.5 Conclusion

This chapter specifies all the hardware and software that was used. Putting together the knowledge from chapter 2, it is then possible to understand the implementation more easily. Once chosen the support tools for application development and the target output platform, it is possible to start the software development. This implementation will be fully represented in the following chapter.

Chapter 4

Implementation

After defining the system requirements and constraints and representing the hardware and software components, it is possible to proceed to the implementation of a system that fulfills the objectives of this Dissertation. This chapter will describe all the development done and is divided into three different sections.

The first section aims to create a minimal configuration for OS Image by creating all the necessary configuration files for an image that is only able to boot. This image will be the starting point for all user customizing later, turning the system modular and easily customizable.

The second section represents the implementation of the graphical application and can be divided into 3 subsections. The first is to make the host system fully capable of running the application smoothly by installing all the requirements and packages to run Yocto. In the second subsection, the GUI is implemented with all possibilities of customization. Finally, in the last subsection, all communication between the graphical interface and the Yocto Project is implemented.

The third section integrates the entire project with Docker to meet all portability requirements.

4.1 Base image

A small image just capable of allowing a device to boot is the summary of the purpose of creating this minimalist image. To be able to boot the system, the base image must have the necessary base packages for it. With this in mind the *"core-image-base"* **??** is used to get the basic, however is added necessary packages. This image does not include DEV packages and if needed, needs to be added to *"local.conf"* file.

```
LICENSE = "MIT"
inherit core-image
IMAGE_INSTALL += " \
kernel-modules \
"
COMPATIBLE_MACHINE = "^rpi\$"
```

Listing 4.1: Core-image-base.

With this configuration, the system is only able to boot and all desired characteristics must be added later by the developed graphical application. With this in mind, it is possible to state that the final output is the combination of the minimalist initial image with all the features added in several files. These files include *"local.conf"*, *"bblayers.conf"*, *"config.txt"* and *"interface.txt"*.

4.2 Graphical user interface

4.2.1 Initial configuration of the system

To be able to use all the tools and features correctly, it is necessary to ensure that all software packages and dependencies are correctly installed. A log file has been deployed to save the last successful boot state. If the system has an unexpected shutdown, this feature makes it possible to boot the system from the last successful state. With this control system, it is then possible to request the user password only when it is necessary to execute commands that require sudo permissions. When the system boots for the first time, the status variable is always 0.

To perform this process of startup and preparation of the host system was implemented a script with a state machine. Communication with this script is established using the *QProcess* function in QT. This function sends the parameters that will trigger the initialization of the desired function.

The first action is to request the user for root permission using *QInputDialog*. This action gets the user password that is sent to the script as a parameter using *QProcess*.

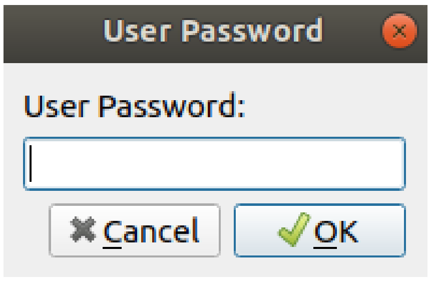


Figure 4.1: QInputDialog.

Once the password has been obtained and sent to the script as a parameter, it is then possible to initialize the installation of all the necessary dependencies to run the Yocto Project. This process is then performed by the following script command:

```
echo $UserPassword | sudo -S -k apt-get install
```

Listing 4.2: Password request

When the process stops, a signal is sent that triggers "*readAllStandardError()*". If the output has an error means the password is wrong and the user is asked to re-enter the password through a new *QInputDialog*. Otherwise, if the password is correct and no error was found, all dependencies will be installed and the next step it will be ready to run.

After the password has been accepted and all the necessary dependencies have been installed, it is now needed to download all content necessary to use the Yocto Project tool. This content consists of several layers, each with the required files for later use in the process of building the final image. This script consists of the following steps:

- **Step 1:** Installing required packages

  To run the Yocto tool the installation of the following packages is extremely necessary.

  ```
  sudo -S -k apt install git
  ```

  and

  ```
  sudo -S -k apt-get install gawk wget git-core diffstat unzip texinfo
      gcc-multilib build-essential chrpath socat cpio python python3
      python3-pip python3-pexpect xz-utils debianutils iputils-ping
      libsdl1.2-dev xterm
  ```

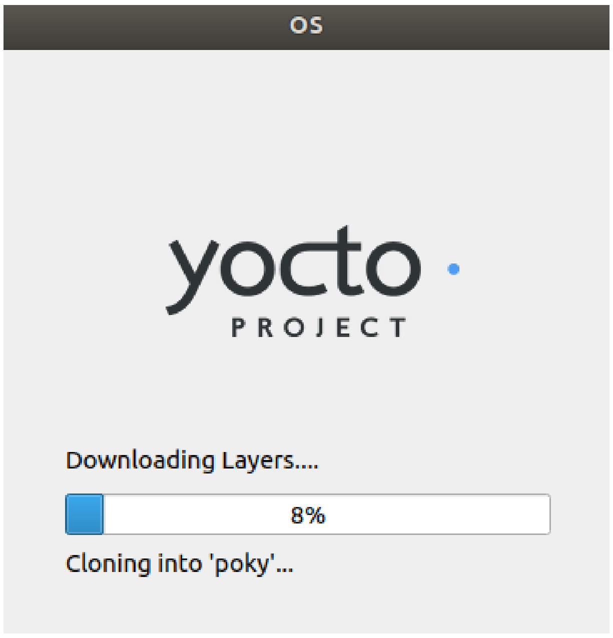
Listing 4.3: Required packages for Yocto

If any of these packages are already installed on the host system, the script starts the installation of the following package on the list. After completing the installation of all requirements, the state variable is changed in the log file so that at the next system startup the password for the installation of required packages will not be asked.

- **Step 2:** Download the first layer:

  At this stage of preparation, it is possible to spot the first security mechanism to make the system more viable. If the script is in this state, it means that if exist, all the existing content that would be downloaded in the future may be corrupted. To protect the user, all existing files are deleted and downloaded from zero.

  After completing the download of each layer, the status variable is changed in the log file. This system turns possible to only download each layer once if the system is rebooted. If the system shuts down in the middle of a download, the previously mentioned protection mechanism works by deleting all existing files for that layer and starting the download from zero.



Downloading Layers....

Cloning into 'poky'...

Figure 4.2: Download GUI.

To make it possible for the user to see what step is being performed at that particular time, a GUI has been created, with a progress bar that is incremented at the end of each complete step 4.2. When all the necessary steps are completed, the status variable is changed to 100. Proper GUI update is performed, the process is killed and a new GUI appears to the user. This new GUI will be where will be chosen and made all the customization of the new image.

It is possible to observe the flowchart of the script operation that is responsible for the preparation of the host system fig: 4.3.

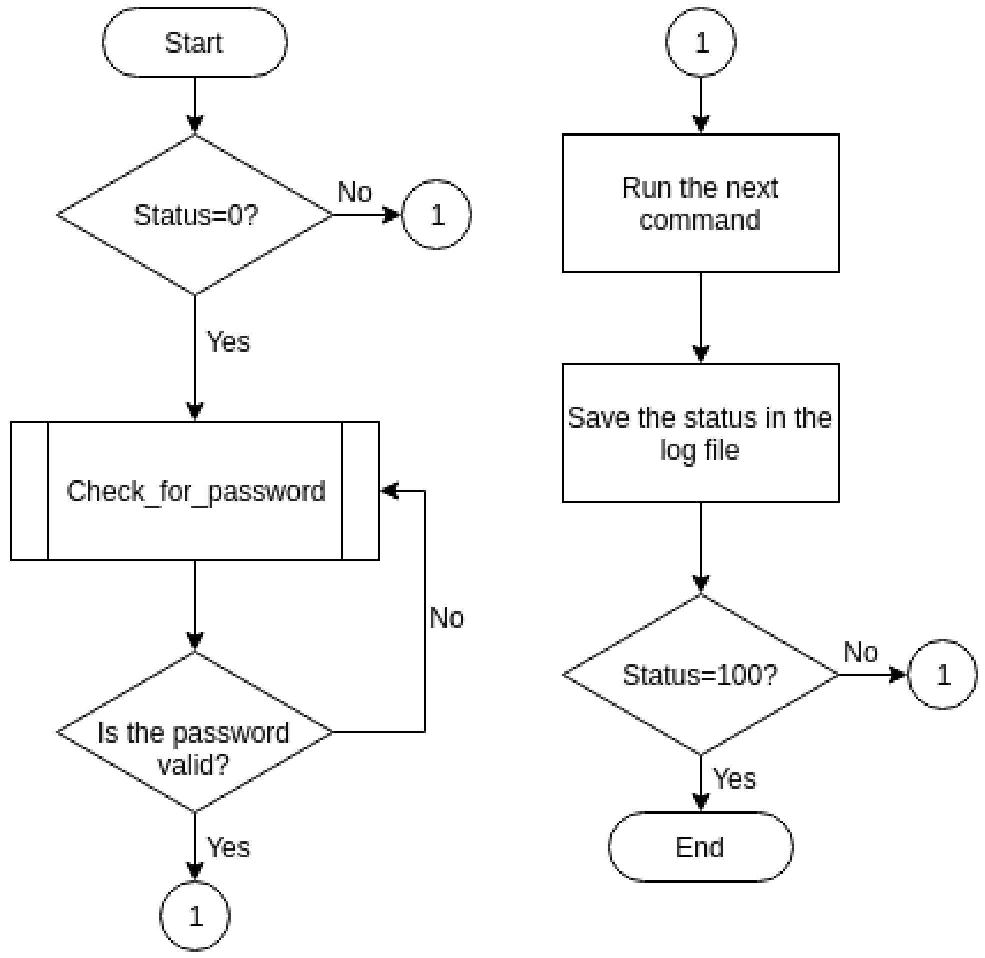


Figure 4.3: System initialization flowchart.

To run commands that require root permissions the user password is essential fig: 4.4. This requires using *QProcess* and script as mentioned above. The system recursively prompts the user for the password until it is correct. After assigned root permissions to the process, it is then possible to perform the necessary system preparation tasks. The password is only required at this stage of the process, so the status is stored in the log file and the next execution, the password is no longer required.

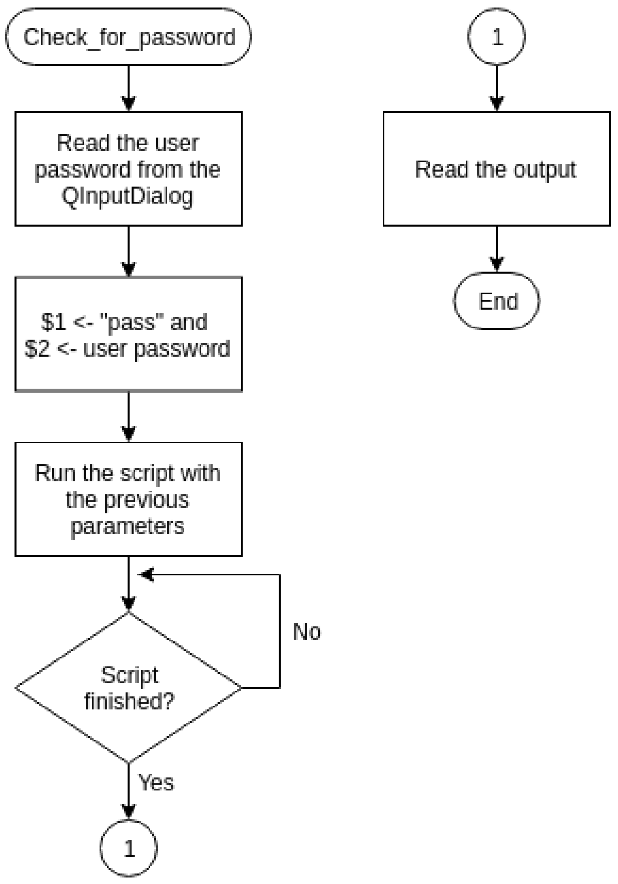

Check_for_password
Read the user password from the QInputDialog
$1 <- "pass" and $2 <- user password
Run the script with the previous parameters
Script finished?
No
Yes
1
1
Read the output
End

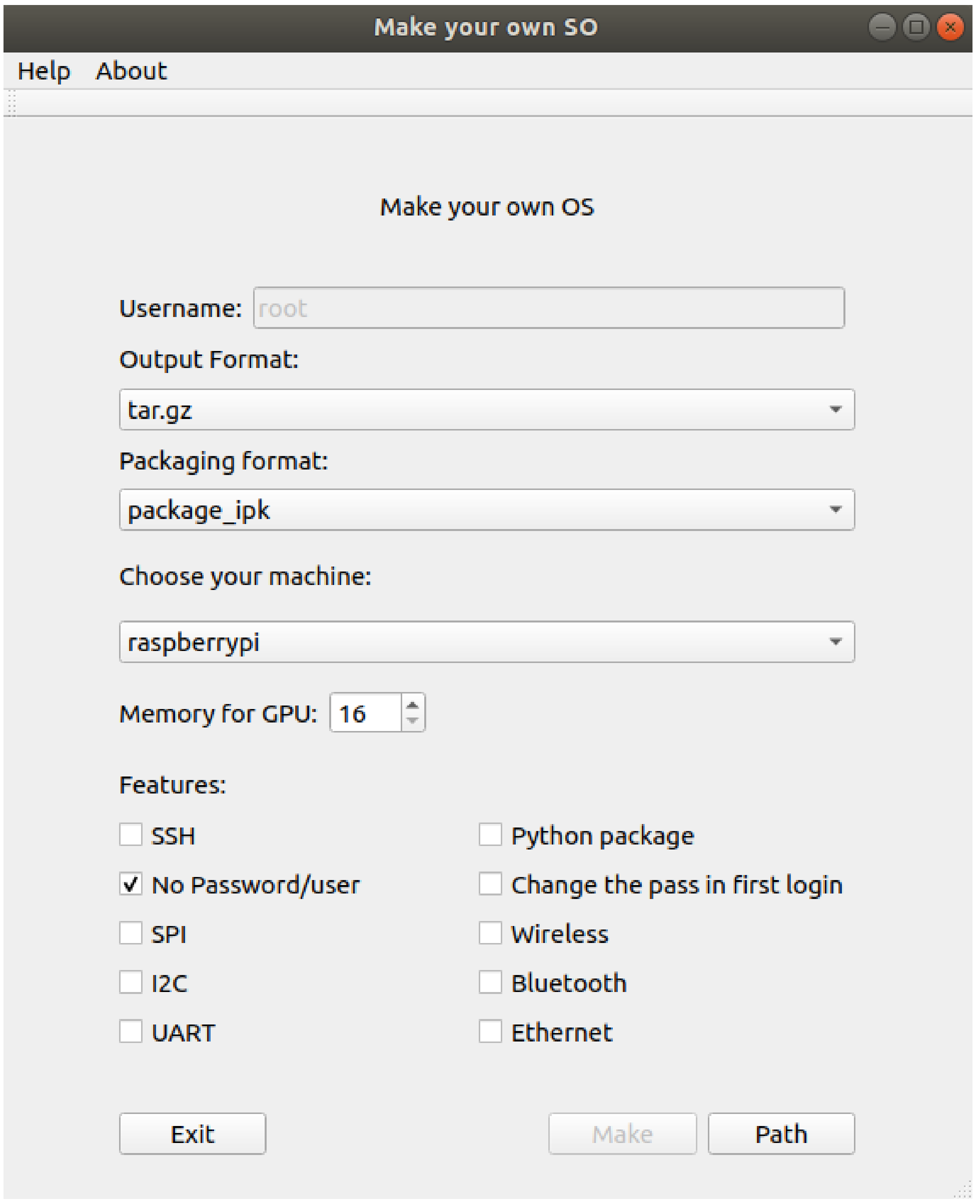
Figure 4.4: Password request flowchart.

After completing all the steps in this script, the system is now ready to accommodate the user's customization. The working directory where all the files for Yocto operation are is now ready for use and it is possible to start the customization.

4.2.2 GUI development

After complete the system host preparation, the state variable is updated to 100 meaning that the system initialization and preparation has been completed. The first step taken after preparing the host system is preparing the script provided by Yocto to build the new image. To this script is added the content needed for complete compatibility with the Raspberry Pi family platform. The next step is the development of the graphical user interface.

This is composed of elements provided by qt, such as:QLineEdit, QLabel , QComboBox , QPushButton , QCheckBox , QSpinBox , QMenuBar , QToolBar and QStatusBar. All of these mechanisms are used for intuitive communication with the user while choosing the desired characteristics for the final image.



Make your own OS

Username: root

Output Format:

tar.gz

Packaging format:

package_ipk

Choose your machine:

raspberrypi

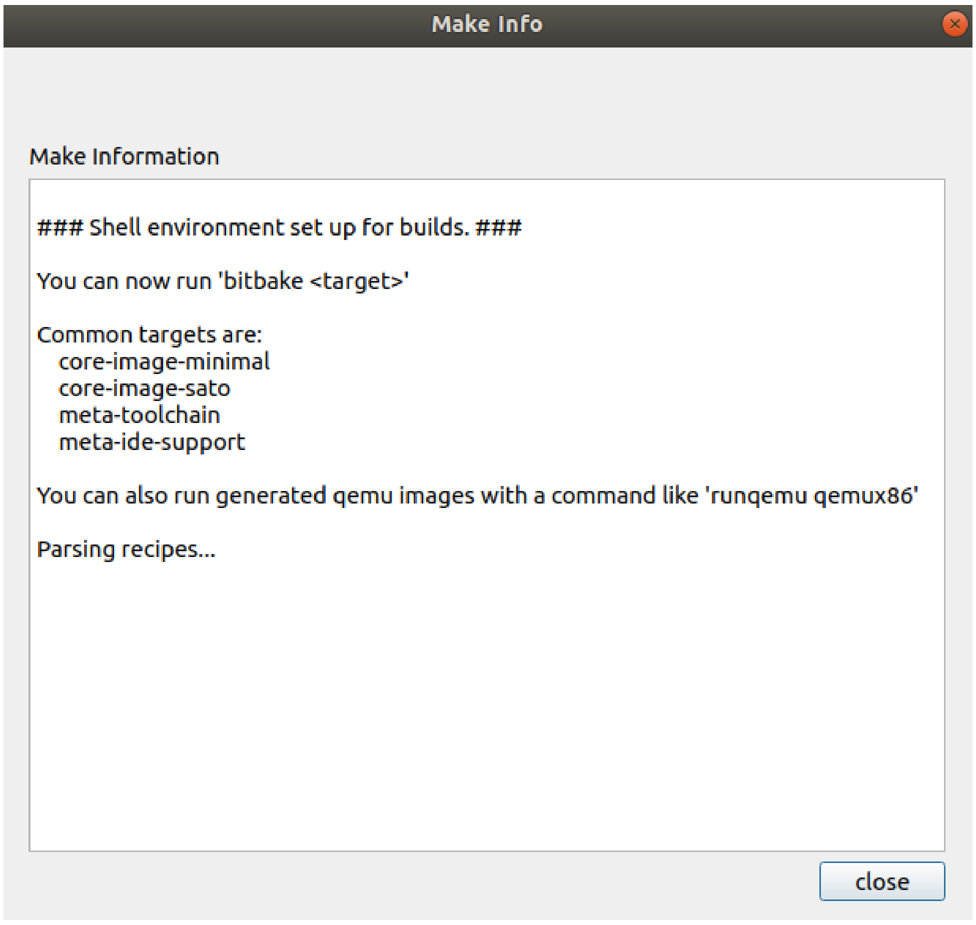
Memory for GPU: 16

Features:

- [ ] SSH
- [✓] No Password/user
- [ ] SPI
- [ ] I2C
- [ ] UART
- [ ] Python package
- [ ] Change the pass in first login
- [ ] Wireless
- [ ] Bluetooth
- [ ] Ethernet

Figure 4.5: Main GUI.

In the figure 4.5, it is possible to see the entire constitution of the GUI. This is the main interface, however, others were developed. The window that gives the user all the information about system preparation is one of them. Another important window is the window that provides the system build information to the user through a *QTextBrowser* 4.6.



```
### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'

Parsing recipes...
```

Figure 4.6: Make windows information.

Other secondary windows have also been developed, among them is possible to highlight, the window that asks the user to enter the password mentioned above. Another example is the usage of these windows to prompt the user to enter the IP address used in the SSH communication if it is selected for the final image 4.7 or the password desired by the user for the final operating system. Both of these last three windows use *QInputDialog*.

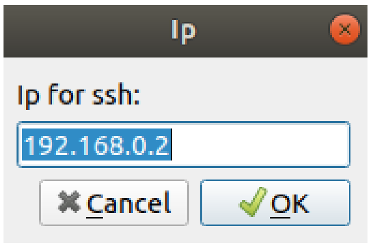


Figure 4.7: Ip for ssh windows.

In the next section will be implemented the ability to choose all features through this GUI.

4.2.3 Customization and features

All choices made in the GUI by the user will have a practical effect on development, from adding, removing or editing the contents of some of the custom files. The customization workflow can be observed in the following figure 4.8:

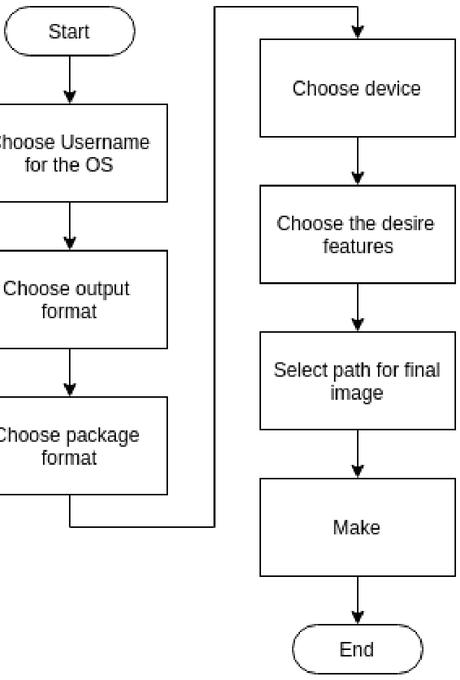
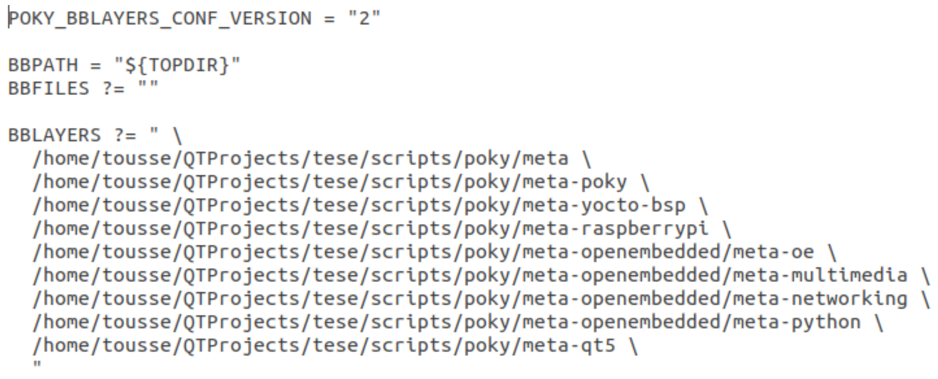

Start
Choose Username for the OS
Choose output format
Choose package format
Choose device
Choose the desire features
Select path for final image
Make
End

Figure 4.8: Customization flowchart.

With application startup, there is a file change associated with it, to add all the features always needed regardless of the custom. In this process, it is necessary to add to the *"bblayers.conf"* 4.9 file all the layers that are intended to be used later.



```
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /home/tousse/QTProjects/tese/scripts/poky/meta \
  /home/tousse/QTProjects/tese/scripts/poky/meta-poky \
  /home/tousse/QTProjects/tese/scripts/poky/meta-yocto-bsp \
  /home/tousse/QTProjects/tese/scripts/poky/meta-raspberrypi \
  /home/tousse/QTProjects/tese/scripts/poky/meta-openembedded/meta-oe \
  /home/tousse/QTProjects/tese/scripts/poky/meta-openembedded/meta-multimedia \
  /home/tousse/QTProjects/tese/scripts/poky/meta-openembedded/meta-networking \
  /home/tousse/QTProjects/tese/scripts/poky/meta-openembedded/meta-python \
  /home/tousse/QTProjects/tese/scripts/poky/meta-qt5 \
  "
```

Figure 4.9: Blayers.conf file.

```
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
     /scripts/poky/meta \
     /scripts/poky/meta-poky \
     /scripts/poky/meta-yocto-bsp \
     /scripts/poky/meta-raspberrypi \
     /scripts/poky/meta-openembedded/meta-oe \
     /scripts/poky/meta-openembedded/meta-multimedia \
     /scripts/poky/meta-openembedded/meta-networking \
     /scripts/poky/meta-openembedded/meta-python \
     /scripts/poky/meta-qt5 \"
```

Listing 4.4: Blayers.conf file.

Before the user can initialize the customization, it is necessary to add settings to the file where later most of the customizations will be added. This file is "local.conf", and the following is added:

```
DISTRO = "poky"
DISTRO_FEATURES_remove = " x11 "
IMAGE_INSTALL_append = " ${CORE_IMAGE_BASE_INSTALL}"
USER_CLASSES ?= "buildstats image-mklibs image-prelink"
SDKMACHINE = "x86_64"
PACKAGECONFIG_append_pn-qemu-native = "sdl"
PACKAGECONFIG_append_pn-nativesdk-qemu = " sdl"
PATCHRESOLVE = "noop"
CONF_VERSION = "1"
```

Listing 4.5: local.conf file.

These previously changes make the system ready to receive customization by the user.

4.2.3.1 Username and Password

The username choose by the user will be the username in the built operating system. If no changes are made by the user to it, it remains ”root”. If no password is added when ”No Password” is deselected, the default password is root. To do this change it is only necessary to add:

```
INHERIT += "extrausers"
EXTRA_USERS_PARAMS = "usermod -P password username"
```

Listing 4.6: Password change.

to the *”local.conf”*. However, it is possible for the user not to have a password on the system at startup and this is the standard option. To do so it is only needed to add the ”debug-tweaks” command to the *”local.conf”* file.

4.2.3.2 SSH protocol

Initially, ssh is unable and needs to be activated by the user if desired. This toggle from off to on trigger a series of events. The first is the addition of the packages needed to use this protocol to the ”local.conf” file by adding the following:

```
"ssh-server-openssh ssh-server-dropbear"
```

After that, ssh is active but not configured. The desired IP is requested to the user to later establish communication via ssh with the target platform. This IP is added to the file ”interface.txt” as follows:

```
iface eth0 inet static
address User_IP
broadcast 192.168.0.255
netmask 255.255.255.0
```

Listing 4.7: Interface for ethernet change.

After these configurations are completed it is possible to use the SSH protocol for communication with the target platform. If it is no longer desired by the user, it is only necessary to uncheck the checkbox and all settings are deleted.

4.2.3.3 Machine

This tool targets the two previously mentioned platforms and all tests will be performed on them. However, it has the capacity and is compatible with the entire Raspberry Pi family of platforms. For this, a Combo Box is used, where the user has available all compatible and BSP provided by the raspberry layer. After choosing the desired platform, ” machine =”name_of_the_machine” ” is added to the *”local.conf”* file.

4.2.3.4 Output and Package format

As a choice of output format, the user can decide between ”tar.gz” or ”rpi-sdimg”.The only difference will be in the size of the output and how the operating system will be flashed on the target platform. As an option of choice for the package format the user can decide between:

- package_deb, provides support for creating packages that use the Debian file format.
- package_rpm, provides support for creating packages that use the RPM file format.
- package_ipk, provides support for creating packages that use the IPK file format.
- package_tar, provides support for creating tarballs.

4.2.3.5 Make

All the other features are added with the same process to the final image, adding information to the ”local.conf” file. After all the customization, the tool can then start the making process. Using QProcess to communicate with the script it is then possible to run the following part of the script:

```
if [ $1 = "bitbake" ]
then
     cd poky
     source oe-init-build-env $2
     cd $2
     bitbake core-image-base
fi
```

Listing 4.8: Bitbake.

Once the bitbake process is complete, the final image is generated and can then be copied to the final path previously chosen by the user.

4.3 Integration with docker

4.3.1 Host configuration

To be able to execute a container, the host system must be prepared with all the necessary resources by the docker. The preparation of this system depends on the operating system used. For **Ubuntu** the user must update the base system packages to ensure that Ubuntu downloads the most up-to-date docker. After the upgrade, the user only needs to install Docker. For **windows** the end goal is the same but the way to go is different. There are two possible paths, the Docker HUB graphical installation that is only compatible with Windows Pro. This GUI is a native Windows application that provides an easy-to-use development environment for building, shipping, and running dockerized apps. The other way is to install the Docker toolbox which is designed for older Mac and Windows systems. With this tool, a Docker client, Docker toolbox management and an Oracle VM Virtual Box are added to the host system. To make it possible for the container to work, it is necessary to install a tool so that the GUI can be displayed on the host system through "X Server". This step is not necessary to perform on the Linux system because it already has an X Server.

4.3.2 Docker File

Once the host system has been prepared, it is now necessary to configure the containers so that they have everything necessary for the execution of the tool. To achieve this configuration it is necessary to create a file called "**Dockerfile**". This file will download and install all required packages. After all installed packages, the GUI will be downloaded from Github. With the GUI running inside of a container, all necessary permissions are given to the downloaded files for the correct functioning of the tool:

```
FROM ubuntu:latest


RUN apt-get update -y; apt-get install cmake -y; apt-get install sudo -y;
    apt-get install -y locales;
```

```
EXPOSE 22
WORKDIR /root
RUN git clone https://github.com/letousse/YoGui_withoutmake.git
RUN make -C YoGui_withoutmake

RUN chmod 777 YoGui_withoutmake/scripts/poky_script.sh

RUN locale-gen en_US.UTF-8
RUN export LC_ALL=en_US.UTF-8
RUN export LANG=en_US.UTF-8

USER root

CMD ["/bin/bash", "/YoGui_withoutmake/tese"]
```

Listing 4.9: Dockerfile

To make this file easy to use, two scripts were implemented. With the host system prepared as mentioned above, the user only needs to run these scripts and everything will be prepared in an automated way. Each script is intended for two different operating systems: Ubuntu and windows.

4.3.3 Linux execution

To run a container on the Linux system the following script was developed:

```
#!/bin/bash
sudo docker stop $(docker ps -a -q)
sudo docker rm $(docker ps -a -q)
sudo docker system prune -a
sudo docker build -t yupi_v1 .
xhost +local:docker
sudo docker run -ti --rm   -e DISPLAY=$DISPLAY -v
    /tmp/.X11-unix:/tmp/.X11-unix -v /home:/new yupi_v1
```

Listing 4.10: Script used on Linux

This script has as its main function the deletion of all existing images or containers to create a new one from scratch. Performing this step, the user can then build the dockerfile mentioned above to create the image with all the necessary content. It is then now possible to run this image. Some parameters are added to the "*run*" command to send the output to a host system and mount user partitions in the container.

4.3.4 Windows execution

In the windows operating system, few differences exist in the script execution workflow. The biggest difference is that windows do not have a native X server. To do so, it is necessary to send the host system IP to the container to be able to send the GUI to it. Everything else is done the same way, just having to adapt the commands to make them run on Windows. In the following code snippet is the script used for the windows operating system:

```
docker stop $(docker ps -a -q)
docker rm $(docker ps -a -q)
docker system prune -a
docker build -t yupi_v1 .
set-variable -name DISPLAY -value 192.168.56.1:0.0
docker run -ti --rm    -e DISPLAY=$DISPLAY    -v
   /tmp/.X11-unix:/tmp/.X11-unix -v /c:/new yupi_v1
```

Listing 4.11: Script used on Windows

4.4 Conclusion

All decisions relevant to the creation of the minimalist image, graphical interface and docker integration are described and presented in this chapter. All these decisions were decisive and important to achieve the final product.

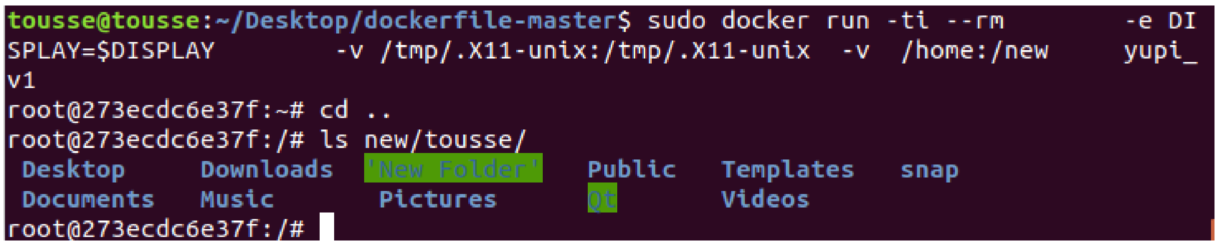
Chapter 5

Tests and Results

To demonstrate the correct functioning of the entire application, modular tests were run. The first test performed is the minimal image that will serve as the basis for the whole process. After this first test, each feature was tested individually. Finally, to show how this tool works in the real world, an image capable of running a gateway was created and tested.

5.1 Docker

To test the correct implementation of the tool implemented with docker, a container was created and the basic Linux command executed. After successfully running these tests, the final test can be performed. This final test is based on running the GUI inside a container. In the following image, it is possible to see the command where the host system is mounted on the container to be possible to save the final image on the host system instead of inside the container.



```
tousse@tousse:~/Desktop/dockerfile-master$ sudo docker run -ti --rm        -e DI
SPLAY=$DISPLAY        -v /tmp/.X11-unix:/tmp/.X11-unix  -v  /home:/new     yupi_
v1
root@273ecdc6e37f:~# cd ..
root@273ecdc6e37f:/# ls new/tousse/
 Desktop      Downloads  'New Folder'   Public    Templates   snap
 Documents    Music       Pictures      Qt        Videos
root@273ecdc6e37f:/# 
```

Figure 5.1: Docker.

5.2 Minimal image

To be able to create the minimal image, the ”core-image-minimal” which is available in Yocto, was taken as a starting point. This minimal recipe provides an image with the minimum number of packages to be a bootable image for a given platform from a stock Yocto Project distribution. This is the fastest complete build, and it is also a good place to start to create a custom image. It is now possible to remove the remaining packages that will not be needed for the initial image. After this procedure, it is possible to reduce the image to a size of 26.7mb with the format of ”tar.gz” if chosen by the user. The kernel version is 4.14.112 and the busybox version is 1.29.3.

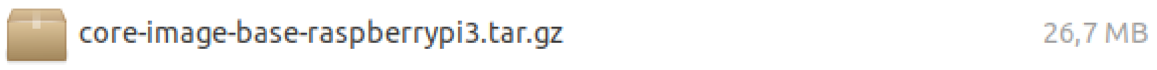

```
core-image-base-raspberrypi3.tar.gz                    26,7 MB
```

Figure 5.2: Minimal image.

The time required for the system to fully boot and be ready for use is 4.8 seconds as shown in the following image.

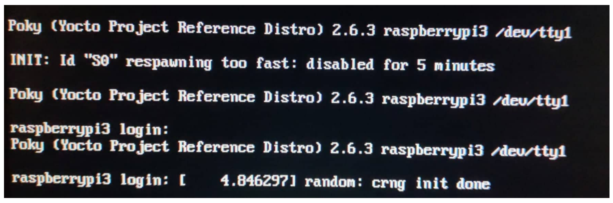

```
Poky (Yocto Project Reference Distro) 2.6.3 raspberrypi3 /dev/tty1

INIT: Id "S0" respawning too fast: disabled for 5 minutes

Poky (Yocto Project Reference Distro) 2.6.3 raspberrypi3 /dev/tty1

raspberrypi3 login:
Poky (Yocto Project Reference Distro) 2.6.3 raspberrypi3 /dev/tty1

raspberrypi3 login: [    4.846297] random: crng init done
```

Figure 5.3: Boot time.

After obtaining the settings for this minimal image it is possible to add the remaining required and desired features. Subsequently, a test of the functioning of each of these features will be performed individually.

5.3 Features test

5.3.1 Username and password

If nothing is changed by the user, then an image is created with no password and the default username of ”root”.

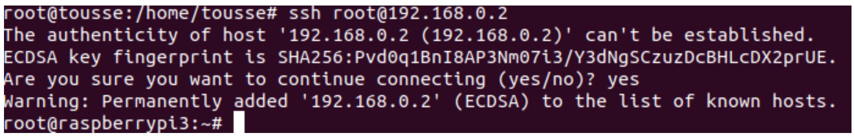

```
root@tousse:/home/tousse# ssh root@192.168.0.2
The authenticity of host '192.168.0.2 (192.168.0.2)' can't be established.
ECDSA key fingerprint is SHA256:Pvd0q1BnI8AP3Nm07i3/Y3dNgSCzuzDcBHLcDX2prUE.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.2' (ECDSA) to the list of known hosts.
root@raspberrypi3:~#
```

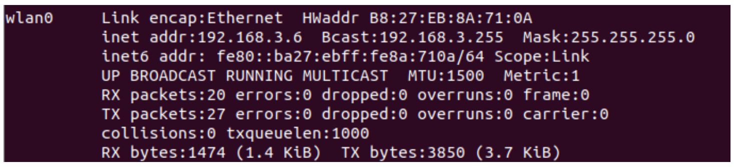
Figure 5.4: Password default.

Now it is possible to unable ”No Password / User” checkbox and a pop-up windows appears requesting the new password from the user. The option ”Username” is unlocked and it is now possible for the user to enter the new username. If not entered, ”root” is set by default for both.

It is also possible to change the password after the first login, you only need to check-in ”Change the pass in the first login”. After the first boot, the user will be asked to enter a new password that will then be the password associated with that username.

5.3.2 Wlan

To perform a test that can confirm the correct operation of wireless network, wla0 was activated and configured with DHCP. The Dynamic Host Configuration Protocol is a network management protocol used on UDP/IP networks whereby a DHCP server dynamically assigns an IP address and other network configuration parameters to each device on a network so they can communicate with other IP networks. In the following figure, it is possible to verify that Wlan is active and with has an assigned IP, making possible the connection via Wlan to the internet.

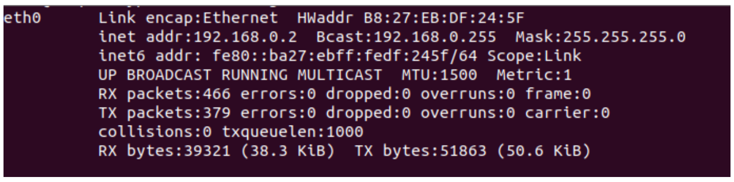


```
wlan0     Link encap:Ethernet  HWaddr B8:27:EB:8A:71:0A
          inet addr:192.168.3.6  Bcast:192.168.3.255  Mask:255.255.255.0
          inet6 addr: fe80::ba27:ebff:fe8a:710a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:20 errors:0 dropped:0 overruns:0 frame:0
          TX packets:27 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1474 (1.4 KiB)  TX bytes:3850 (3.7 KiB)
```

Figure 5.5: Wlan information.

5.3.3 Ethernet

In this case, the test operation is extremely similar to the test of Wlan. The major difference is that in this case, it is not necessary to configure the file ”wpasupplicant.conf” to make the communication with a wireless network possible. It is only necessary to configure eth0 with the DHCP protocol and make the physical connection. In both tests (Wlan0 and eth0) a ping to ”google.com” was performed to confirm the internet connection.

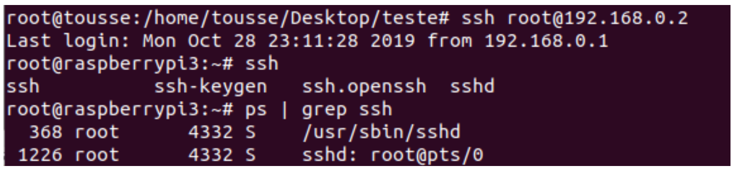


```
eth0      Link encap:Ethernet  HWaddr B8:27:EB:DF:24:5F
          inet addr:192.168.0.2  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::ba27:ebff:fedf:245f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:466 errors:0 dropped:0 overruns:0 frame:0
          TX packets:379 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:39321 (38.3 KiB)  TX bytes:51863 (50.6 KiB)
```

Figure 5.6: Ethernet information.

5.3.4 SSH

To perform this test it is necessary to assign a static IP to the eth0 configuration instead of the DHCP protocol previously used to configure ethernet. This hardware conflict brings a restriction and it is only possible to have ssh active if ethernet is not configured to establish an internet connection.

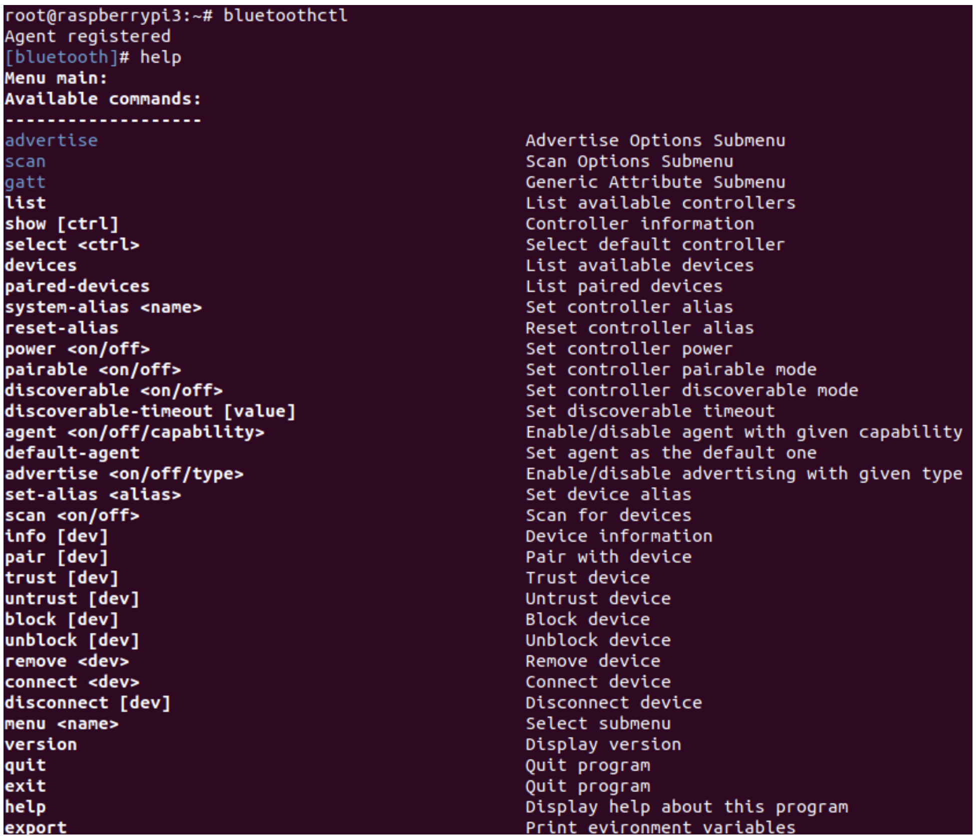


```
root@tousse:/home/tousse/Desktop/teste# ssh root@192.168.0.2
Last login: Mon Oct 28 23:11:28 2019 from 192.168.0.1
root@raspberrypi3:~# ssh
ssh           ssh-keygen   ssh.openssh  sshd
root@raspberrypi3:~# ps | grep ssh
  368 root      4332 S    /usr/sbin/sshd
 1226 root      4332 S    sshd: root@pts/0
```

Figure 5.7: SSH connection.

5.3.5 Bluetooth

To perform the Bluetooth test, it must be previously activated on the GUI. With this active, it is possible to access an interface as shown in the following image. Through a text menu, it is possible to perform all Bluetooth communications and pairings.



```
root@raspberrypi3:~# bluetoothctl
Agent registered
[bluetooth]# help
Menu main:
Available commands:
-------------------
advertise                                         Advertise Options Submenu
scan                                              Scan Options Submenu
gatt                                              Generic Attribute Submenu
list                                              List available controllers
show [ctrl]                                       Controller information
select <ctrl>                                     Select default controller
devices                                           List available devices
paired-devices                                    List paired devices
system-alias <name>                               Set controller alias
reset-alias                                       Reset controller alias
power <on/off>                                    Set controller power
pairable <on/off>                                 Set controller pairable mode
discoverable <on/off>                             Set controller discoverable mode
discoverable-timeout [value]                      Set discoverable timeout
agent <on/off/capability>                         Enable/disable agent with given capability
default-agent                                     Set agent as the default one
advertise <on/off/type>                           Enable/disable advertising with given type
set-alias <alias>                                 Set device alias
scan <on/off>                                     Scan for devices
info [dev]                                        Device information
pair [dev]                                        Pair with device
trust [dev]                                       Trust device
untrust [dev]                                     Untrust device
block [dev]                                       Block device
unblock [dev]                                     Unblock device
remove <dev>                                      Remove device
connect <dev>                                     Connect device
disconnect [dev]                                  Disconnect device
menu <name>                                       Select submenu
version                                           Display version
quit                                              Quit program
exit                                              Quit program
help                                              Display help about this program
export                                            Print evironment variables
```

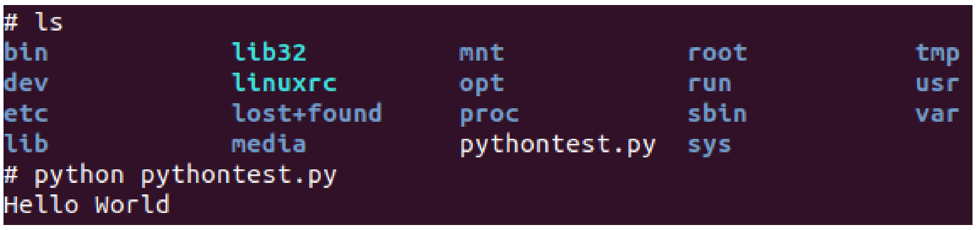
Figure 5.8: Bluetooth interface.

5.3.6 Python

A simple python script was executed on the system to print a simple message.

```
print("hello world")
```

Although simple, the executed python script confirms the existence of python libraries and the possibility of running python with more complexity.

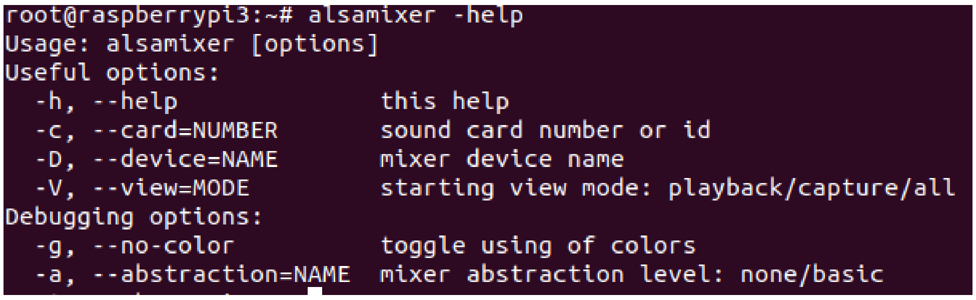


```
# ls
bin             lib32           mnt             root            tmp
dev             linuxrc         opt             run             usr
etc             lost+found      proc            sbin            var
lib             media           pythontest.py   sys
# python pythontest.py
Hello World
```

Figure 5.9: Python script.

5.3.7 ALSA

Like Bluetooth, the Advanced Linux Sound Architecture is part of the Linux kernel which provides an interface for sound card device drivers and as seen in the following image the API is active on the system and ready for use by the user.



```
root@raspberrypi3:~# alsamixer -help
Usage: alsamixer [options]
Useful options:
  -h, --help              this help
  -c, --card=NUMBER       sound card number or id
  -D, --device=NAME       mixer device name
  -V, --view=MODE         starting view mode: playback/capture/all
Debugging options:
  -g, --no-color          toggle using of colors
  -a, --abstraction=NAME  mixer abstraction level: none/basic
```

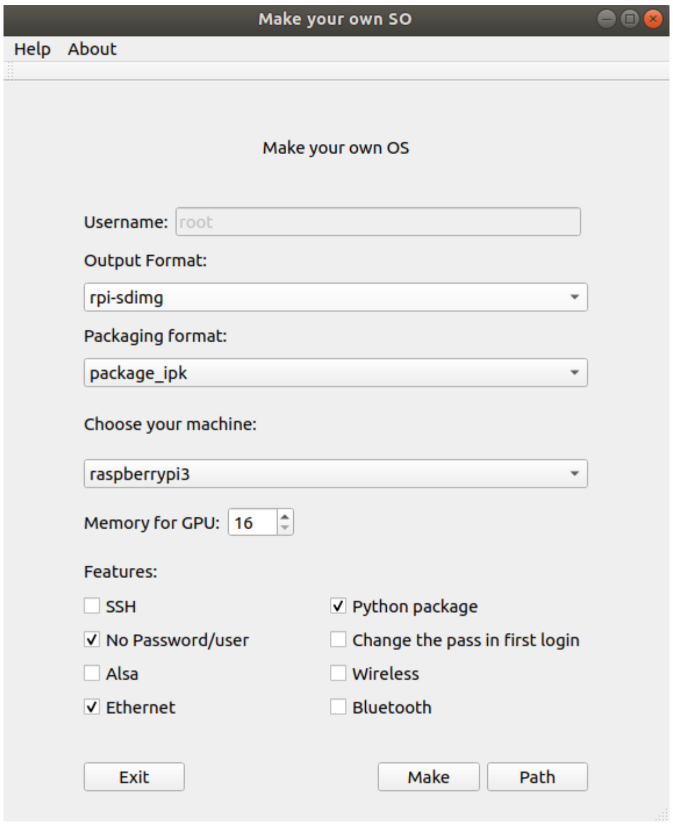
Figure 5.10: ALSA interface.

5.4 IoT test

To perform a real-world test, an image was created capable of running a gateway with the functions of:

- Forwarding JSON messages from the coordinating node to the web microservices.
- Temporary storage of messages in case of lack of internet connection or unavailability microservices.

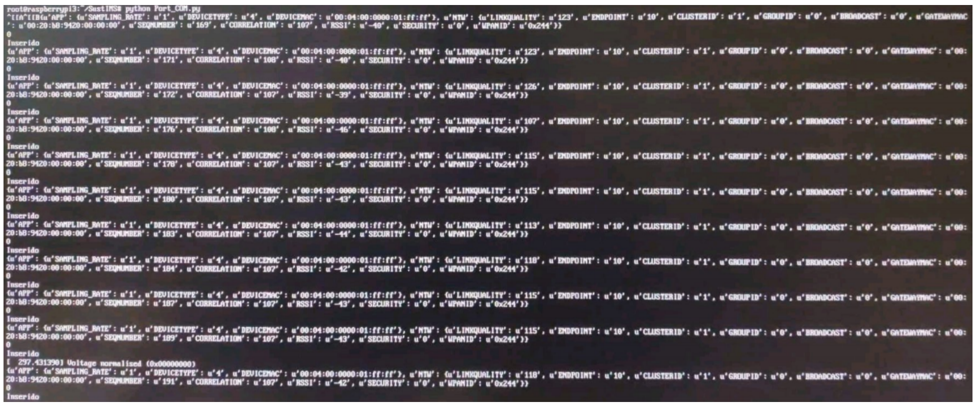
To be possible perform the previous test is required an internet connection, in this case via Ethernet and python libraries.



```
Make your own SO
Help   About

Make your own OS

Username: root
Output Format:
rpi-sdimg
Packaging format:
package_ipk
Choose your machine:
raspberrypi3
Memory for GPU: 16
Features:
[ ] SSH                      [✓] Python package
[✓] No Password/user         [ ] Change the pass in first login
[ ] Alsa                     [ ] Wireless
[✓] Ethernet                 [ ] Bluetooth

Exit                         Make   Path
```

Figure 5.11: Gui to perform the IoT test.

Having this in mind, the final image was created with all these features using the graphical interface. In the following image, it is possible to see the gateway running and sending the packages.



```
root@raspberrypi3:~/SustIMS# python Port_COM.py
^[[A^[[B{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'123', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC
': u'00:20:b8:9420:00:00:00', u'SEQNUMBER': u'169', u'CORRELATION': u'107', u'RSSI': u'-40', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'123', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'171', u'CORRELATION': u'108', u'RSSI': u'-40', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'126', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'172', u'CORRELATION': u'107', u'RSSI': u'-39', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'107', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'176', u'CORRELATION': u'108', u'RSSI': u'-46', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'115', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'178', u'CORRELATION': u'107', u'RSSI': u'-43', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'115', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'180', u'CORRELATION': u'107', u'RSSI': u'-43', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'113', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'183', u'CORRELATION': u'107', u'RSSI': u'-44', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'118', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'184', u'CORRELATION': u'107', u'RSSI': u'-42', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'115', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'187', u'CORRELATION': u'107', u'RSSI': u'-43', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'115', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'189', u'CORRELATION': u'107', u'RSSI': u'-43', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
[  297.431390] Voltage normalised (0x00000000)
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'118', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
20:b8:9420:00:00:00', u'SEQNUMBER': u'191', u'CORRELATION': u'107', u'RSSI': u'-42', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
```

Figure 5.12: Final test being performed.

Chapter 6

Conclusions and Future Work

After the implementation of the whole system, conclusions can be made from the developed work and results obtained. In this chapter, the main goal is to make an overview of the results obtained and what can be concluded from them. Possible improvements will also be suggested for future implementation.

6.1 Conclusion

Unlike more complex systems such as desktops or smartphones, embedded systems are designed to perform a reduced number of tasks. It is then possible to reduce the size of the system by reducing the number of features not needed to perform the main goal of all systems. Having this in mind, this tool is extremely useful and effective in creating these most specialized systems with few functions to be performed without requiring in-depth knowledge of them.

It is then possible to conclude based on the tests, it is extremely easy to create an image with the desired features, just using the GUI. A good example of this effectiveness is the creation of an image where python libraries and internet connection are required. It was just necessary to run the application, choose the necessary packages and build the OS image. It is possible to accomplish the same goal with a system with all features active, however, it would be necessary to use an image with a larger size, a longer boot time and a huge amount of unnecessary packages and modules to perform these functions.

The integration of this tool with Docker has brought portability to the system. Yocto has as restriction the need to use Linux operating systems on the host system. Because of the operation of Docker, this tool increased its compatibility with other operating systems besides Linux. Because of its ability to operate on operating systems such as Windows or MacOS, docker has made this tool available to a larger number of users. On the other hand, using the docker is no longer up to the user to configure the host system and this responsibility now belongs to Docker via the Dockerfile making this process fully automated. As one of the main goals for this project, the user will just have to run a script, configure the desired GUI and wait for an operating system ready to use on the desired target.

6.2 Future work

In future work, the support for more target families than raspberry pi should be implemented. To make this feature even more advantageous, a database should be created with all available layers for different machines. With the database ready to use, tow fields should be implemented int the graphical application allowing the user to choose between: the family of targets where the board belong and the specific target. This option will make it possible to use the tool with a large number of hardware.

Another future implementation is due to the huge amount of packages the image can have. To overcome this challenge, a database would have to be implemented to create the possibility for the user to search for less common packages. In the database, it would have to be possible to find the machines compatible with the package, dependencies to use them and where they would have to be added in the configuration files. The yocto project has a database with all these packages, called ”index” but the compatibility information is needed and the necessary configuration file change is missing from this database. In summary, it would be necessary, starting from the database provided by yocto, to add extra information to make its use possible in an automated way.

Finally, implementing the flash memory functionality directly flash the image created to have an automated start-to-finish process. With the implementation of this feature, the process would be entirely automated by the tool, removing any workload to the user to create the desired image.

References

[1] Daniel P. Bovet and Marco Cesati. *Understanding Linux Kernel*. 2006.

[2] Chad Z. Hower aka Kudzu. *Documentation*. 2007. URL: `gocosmos.org/docs/`.

[3] Ssarthak598. *Make a Simple Operating System*. URL: `nstructables.com/id/Make-A-Simple-Operating-System/`.

[4] Tomas Matejicek. *Linux Live Kit*. 2019. URL: `linux-live.org/#home`.

[5] Gerard Beekmans. *Linux From Scratch Version 8.4*. 2019.

[6] Gerard Beekmans. *Linux From Scratch*. 2016. URL: `wiki.linuxfromscratch.org`.

[7] Tomas Matejicek. *Slax, Pocket operating system*. 2019. URL: `slax.org`.

[8] Debian Installer team. *Debian GNU/Linux Installation Guide*. In: *Debian GNU/Linux Installation Guide*. June 23, 2019.

[9] Devian lite team. *Devian Wiki*. July 7, 2019. URL: `debian.org`.

[10] Hauke. *OpenWRT Wireless Freedom*. July 2, 2019. URL: `openwrt.org`.

[11] Marcus Schafer. *openSUSE-KIWI Image System*. Apr. 17, 2016.

[12] *Studio Express*. Feb. 18, 2018. URL: `en.opensuse.org/Studio_Express`.

[13] Andreas Jaeger. *SUSE Studio online + Open Build Service = SUSE Studio Express*. Sept. 22, 2017. URL: `.suse.com/c/suse-studio-online-open-build-service-suse-studio-express/`.

[14] Marcia Wilbur. *LINUX RESPIN*. 2018. URL: `linuxrespin.org`.

[15] Scott Rifenbark. *Yocto project Mega-Manual*. 2019.

[16] Scott Rifenbark. *Toaster User Manual*. 2015.

[17] Drew Moseley. *Tools for building embedded Linux systems*. June 15, 2018. URL: `opensource.com/article/18/6/embedded-linux-build-tools`.

[18] Buildroot developers. *The Buildroot user manual*. Aug. 6, 2019.

[19] Sriram Neelakandan P. Raghavan Amol Lad. *EMBEDDED LINUX SYSTEM DESIGN AND DEVELOPMENT*. 2006.

[20] The Linux Information Project. *Kernel Definition*. 2005. URL: `linfo.org`.

[21] Shantanu Tushar. *Linux Shell Scripting Cookbook*. 2013.

[22] Wolfgang Mauerer. *Linux Kernel Architecture*. Ed. by Inc. Wiley Publishing.

[23] Chris Larson Richard Purdie and Phil Blundell. "BitBake User Manual". In: *BitBake Community* (2019).

[24] Patrick Bitterling. "Operating System Kernels". In: ().

[25] Luca Ceresol. "Buildroot vs Yocto:" in: *Differences for Your Daily Job* (2018).

[26] Scott Rifenbark. "Yocto Project Development Manual". In: *Yocto* (2014).

[27] Karim Yaghmour. Embedded Linux. In: *Embedded Linux*. 2012.

[28] Gilad Ben-Yossef and Philippe Gerum Karim Yaghmour Jon Jason Masters. *Building Embedded Linux Systems*. 2008.

[29] Fabrizio Castro. "How to Survive Embedded Linux". In: *The Embedded Linux Development Process* (2018).

[30] Greg Kroah-Hartman. *LINUX KERNEL IN A NUTSHELL*. 2006.

[31] meta-raspberrypi contributors. "meta-raspberrypi Documentation". In: (2019).

[32] Tomas Frydrych. *poky-ref-manual*. 2011.

[33] Elizabeth Flanagan. *The Architecture of Open Source Applications (Volume 2): The Yocto Project*. 2018.

[34] Intel Corporation. *The Yocto Project Overview and Update*. 2012.

[35] *What is Operating System, Kernel and Types of kernels*. URL: `http://www.go4expert.com/articles/operating-kernel-types-kernels-t24793/`.

[36] JUSTIN GARRISON. *What is the Linux Kernel and What Does It Do?* 2017. URL: `https://www.howtogeek.com/howto/31632/what-is-the-linux-kernel-and-what-does-it-do/`.

[37] O.S. Systems. *What the Yocto Project is*. 2017. URL: `https://www.ossystems.com.br/blog/what-the-yocto-project-is`.

[38] Drew Moseley. *Why the Yocto Project for my IoT Project?* 2017.