Creating tailored OS images for embedded systems using Buildroot

Vladyslav Reznikov

UMinho | 2019

Universidade do Minho
Escola de Engenharia

Vladyslav Reznikov

Creating tailored OS images for embedded systems using Buildroot

dezembro de 2019

Universidade do Minho
Escola de Engenharia

Vladyslav Reznikov

Creating tailored OS images for embedded systems using Buildroot

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Cabral

dezembro de 2019

Acknowledgements

I would like to thank Professor Jorge Cabral for giving me the chance to develop this project, and for all the guidance throughout the development of this work. Also i would like to thank Professor Matjaz Colnaric from Univerza v Mariboru who guided me during my Erasmus+ mobility in the first semester where the theoretical research for this Dissertation was made. Thank you everyone from ESRG for the help and support.

To my family for always being my side no matter the choice i made. To my mom and my little brother, for being my inspiration. A huge thanks to my Dad for the incentive. To my grandmother for trying to help and understand what i was developing although not having any bases in the area. To my stepmother and my sister for being cheerful with every minor improvement of mine.

A huge thanks to Rafael for being my partner and part of this journey since the very beginning, and never leaving me behind. A special thanks to Rafaela for being the best friend i could have during this past times and always supporting me no matter what. To Miguel for all the friendship and support. To Afonso and João for never letting me quit. To my Royal Pomba friends for never letting me down. To my friends from OPUM DEI for refreshing my mind. To all of my colleagues and friends from NEEEICUM and AAUM for constantly teaching me new things. To my friends from Erasmus for showing me that it doesn't matter where you are from to achieve great things.

A big thank you to Beatriz for all the patience and care given through these last few months. For being the best person i could've met and always guiding me to the right direction. Without your love and support none of this would be possible.

Thank you to everyone who has crossed paths with me and contributed to making me the person i am today.

STATEMENT OF INTEGRITY

Resumo

Criação de SO customizáveis para um ambiente embebido utilizando o Buildroot

A evolução dos sistemas embebidos tem sido cada vez mais notória durante os últimos anos, embarcando nas mais diversas áreas e necessidades dos seres humanos. Hoje em dia, as tecnologias embebidas são responsáveis pelas capacidades sensoriais de uma grande maioria de dispositivos modernos, quer a nível do consumidor, quer a nível industrial. De modo a operar no ambiente embebido, o dispositivo tem de possuir um sistema operativo adequado. Estes sistemas operativos diferenciam-se dos SO regulares, pois garantem que o sistema opere confiável e eficientemente através da manipulação de recursos hardware e software. De modo a facilitar o desenvolvimento, uma camada de abstração também é fornecida. O hardware que corre num sistema embebido é, normalmente, limitado nos seus recursos como a RAM (Random Access Memory) e a ROM (Read-only Memory), fazendo com que estes sistemas sejam desenhados com o propósito de garantir a sua eficiência. Desta troca advém, naturalmente, uma perda de outras funcionalidades.

De modo a escolher quais são as funções do sistema que devem prevalecer para que este opere corretamente, é necessário customizar o sistema operativo. Atualmente, existem ferramentas que permitem construir imagens personalizadas para os sistemas operativos, como é o caso do Buildroot. O Buildroot permite que o sistema seja construído apenas com os recursos necessários para o cumprimento da finalidade de um sistema embebido, fazendo com que este sistema seja mais compacto e determinístico. A personalização da imagem é feita através de um menu de texto , que tem como base a manipulação de ficheiros de configuração e *shell scripts*.

O menu de texto disponiblizado pelo Buildroot tem uma vasta coleção de ferramentas disponíveis que são adicionadas a um ficheiro de configurações gerais, mediante a arquitetura desejada. A imagem do SO final, apesar de ser adequada para o sistema embebido em questão, não está configurada de uma forma minimalística. O tópico desta dissertação é entender a funcionalidade do *back end* do Buildroot de modo a criar uma ferramente que permite a criação de imagens com conteúdo mínimo através de um menu gráfico de fácil compreensão.

Palavras-chave: Buildroot, Compute Module, Raspberry Pi Sistema Embebido, Sistema Operativo.

Abstract

Creating tailored OS images for embedded environment using Buildroot

The embedded systems progression is noticeable throughout the last years. Today, embedded technologies are responsible for the intellectual capabilities of most modern devices, both consumer and industrial. To operate in an embedded environment, a device must own an embedded operating system. This OS differentiates itself from a regular OS by insuring it operates in an efficient and reliable manner by managing hardware and software resources, providing an abstraction layer to simplify the process of developing higher layers of software. The hardware running an embedded system can be very limited in resources such as RAM and ROM, making these systems designed for resource efficiency that comes at the cost of losing some functionalities. Hence, the operating system must be tailored in order to achieve desired operations under these circumstances.

The customization and build of the image can be done with Buildroot tool, which allows the user to build an image only with needed features and packages, making the system more compact and deterministic. The customization is done through a front end menu interface which back end manipulates configuration files and shell scripts.

The configuration through menu interface has an extensive range of available features that are built on a template file with additional general configurations. The final images, although being suitable for embedded devices, are not minimally tailored. This Dissertation understands the back end functionality of Buildroot in order to create a tool that creates minimalistic images for embedded usage based on a minimal default image and the configuration is done through a perceptive GUI, running in all type of environments.

Keywords: Buildroot, Computed Module, Embedded Systems, Operating System, Raspberry Pi.

Table of Contents

List of Figures

List of Listings

Acronyms List

BSP	Board Support Package.
CD	Compact Disk.
CDFS	Compact Disk File System.
CM	Compute Module.
COSMOS	C# Open Source Managed Operating System.
CPU	Central Processing Unit.
DHCP	Dynamic Host Configuration Protocol.
DIY	Do It Yourself.
FS	File System.
GNU	GNU's Not Unix.
GPIO	General-purpose input/output.
GUI	Graphical User Interface.
HAL	Hardware Abstraction Layer.
IoT	Internet of Things.
IPC	Inter Process Communication.
MB	Mega byte.
MOC	Meta-Object Compiler.

OS	Operating System.
POSIX	Portable Operating System Interface.
QEMU	Quick Emulator.
RAM	Random Access Memory.
ROM	Read-only Memory.
Rpi	Raspberry Pi.
SD	Secure Digital.
SDK	Software Development Kit.
SoC	System on Chip.
USB	Universal Serial Bus.
VM	Virtual Machine.

Chapter 1

Introduction

Embedded systems are everywhere around us. As the world has become more technological, embedded system also followed the growth and became indispensables. Nowadays we use embedded systems in our smartphones, cars and at home without even knowing we are using them. Due to real-time constraints, these systems were initially used in time-critical application domains where failures could lead to huge consequences. However, as the cost for processing power and memory decreased alongside the ability to design low cost SoCs (System on Chip), the range of application environments for embedded systems enlarged. Their dependence and growth led to new techniques and to the emergence of a strong industry that develop and use this kind of systems.[1]

Embedded systems , as the name suggests, are systems that are embedded in a larger one by having a dedicated function. An embedded system can be defined as any computer system contained within a product that is not identified as a computer. As any electronic system, an embedded system requires a hardware platform to run, which is usually a microcontroller. On the other side, there must be a specific software written for embedded systems to perform a particular function. It is possible to conclude that these systems are a combination of hardware and software that are usually developed simultaneously since the software components can take advantage of special hardware features in order to improve performance and the hardware components, on the other side, can simplify the design of a module by implementing certain functionality in software.

By comparing embedded systems to generic ones, some major differences can be observed. Embedded systems focus essentially on cost, power consumption and predictability as generic systems only consider the cost and average speed. In order to achieve predictability and to not waste any computing power, the run-time of an embedded system must be fixed while generic systems follow the faster the better approach. Embedded system's application is known during the design time because the system is being designed for a specific function while generic systems have a wide range of applications as we can observe in any desktop.

In order to make embedded devices smarter and capable of doing multiple jobs in less time, the need for an OS (Operating System) emerged. An Operating System supports enough memory for multi tasking, can be reusable and is also stable with more software updates. Nevertheless, General-Purpose Operating Systems are not suitable for an embedded environment, creating the need for an Embedded Operating System.

1.1 Contextualization

Embedded systems continuous growth is noticeable. As the production cost of SoC decreases, the number of applications for embedded systems flourishes alongside the amount of embedded engineers.

The embedded systems are known for being an integration between the hardware and software. This way, it is possible to discern two different parts, the Embedded Hardware - which is a combination of the microchip and several electronic components, and the Embedded Software - which is able to perform operations and tasks on the hardware. The individual existence of each of this elements does not bring any advantage to the world. The structure of and embedded system can be seen in the Figure 1.1

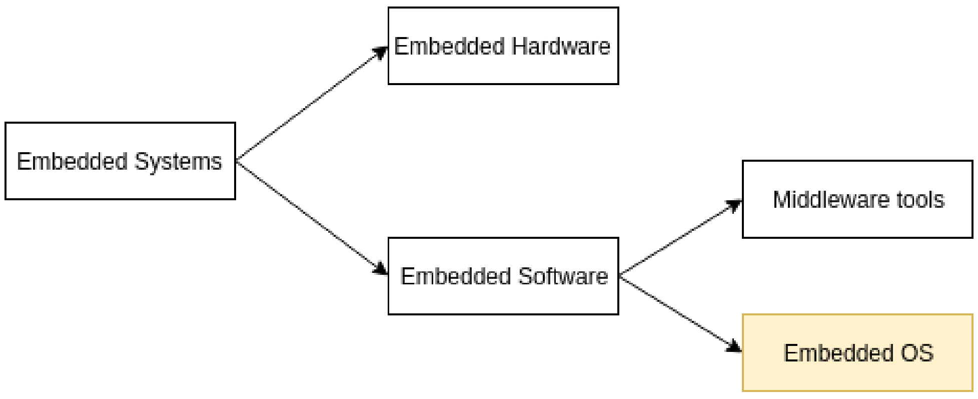


Figure 1.1: Embedded Systems Elements.

The embedded systems' appliance in the market has grown to multi-core technologies, making a single chip behave like it has a logic of 2 or more processors, driving the electronic devices to be smarter and urging to create connectivity between them with the emergence of the IoT (Internet of Things). Furthermore, industries like the automotive require an embedded graphic systems.[2] This led to the requirement for a proficient software modified for the target's purpose.

Embedded developers bump into loads of obstacles in order to develop an embedded system. While desktop developers can safely ignore everything that happens before the first line of application code in *main()* or similar, the embedded developers have plenty of issues to overpass before starting the code. Setting up the board specific devices, initiating the processor work, configuring stack pointers and addressing RAM (Random Access Memory) are just some of them. Happily, the market has tools to automate this process by generating an image for the Embedded OS depending on user's choice through a text based interface.

Using Buildroot it is possible to configure the necessary features for a Linux-based OS and generate all the necessary files to run it on the desired architecture. This tool, although simplifying the process, still requires some knowledge in order to choose from a vast variety of available packages to suit the final OS image to fulfill the needs of the embedded system. The modifications on kernel's level require even a steeper learning curve as the majority of OS images are built upon a default kernel configuration which is still overweight.

By abstracting the user from all the necessary knowledge and Buildroot's operations it is possible do increase the range of developers using embedded systems and making the embedded developers to save time when creating OS specific to their needs. Building the OS features upon a lighter default image and having a minimal configuration for kernel makes the final OS more specific and unloads the processor.

1.2 Motivation

One of the demands of the Embedded Systems master course was to develop a project on a Raspberry Pi board recurring to the usage of an Embedded Operating System. In order to suit the needs of the project, the OS image was created with the assistance of Buildroot. After studying the basis of Buildroot, an image was generated for Raspberry Pi 3 architecture by following simple steps on what to choose in the text based configuration menu. By the end it was known how the Embedded Systems work and what is their role as well how to develop a project with real time constraints. Every part made itself clear except what comes before the beginning of the project and how can the Operating Systems be customized in the embedded environment.

First of all it was important to understand that the OS generated by Buildroot were based on Linux. It is not possible to install and run Linux on embedded devices for the reason that Embedded Systems:

- may be based on different processors and support different peripherals;
- boot from flash instead of hard drives;
- don't support BIOS;
- are resource constrained and hence can't run bulky OS;
- are low power devices;
- require real time performance.

Therefore, Embedded Linux was created to satisfy these requirements. In order to understand how it works and to tailor it for a specific need, a deep knowledge about Kernel is indispensable. Running the generated Linux-based OS on a platform is only possible by having a BSP (Board Support Package) that contains all the necessary drivers for the OS to function in a particular hardware environment.

Despite all the hard work, the embedded systems are clearly conquering the market and it is no longer possible to imagine a world where the embedded systems do not exist. But what if creating an OS for a specific embedded environment was easier and saved more time?

1.3 Objectives

Taking in consideration the motivation, it is the aim of this Dissertation to simplify the creation of Embedded OS images by abstracting the user from all the hard work, and making them even more specific to the project, fulfilling one of the main constraints: low size. The target platforms for this Dissertation are the boards used in the laboratory by other students of the development team.

The objectives of this Dissertation are:

- Understand the Embedded Linux and Kernel;

- Study Buildroot and analyse its features;
- Create a minimal configuration OS image for Raspberry Pi family;
- Automate the OS image creation;
- Abstract the user from all the hard work;
- Grant portability across machines;
- Make it easy to scale for other architectures.

1.4 Dissertation Structure

This document is split in six chapters, and its structure follows a logical order according to the development process that occurred during this *Master's Thesis*.

The first chapter introduces the topic of this Dissertation, referring the context and motivation for its development, as well as what is aimed to achieve and how.

The chapter two starts by introducing the basic concepts needed for a deeper understanding of this Dissertation. It also explores the market offers to overcome the problem that this project tries to solve.

The third chapter gives an overview of the system, its requirements and constraints as well as a deeper specification of the target platforms.

In the chapter four a brief explanation on how the project was implemented is given, segregating each path taken in order to achieve the aim of the project.

Chapter five shows the tests that were made alongside some considerations about the obtained results.

In the sixth chapter, the main conclusion of this project is presented beside the possible future improvements that can be made.

1.5 Methodoly

Before starting the development of this Disseration's final application, the market needs had to be analyzed deeply in order to make a project that doesn't lack what others try to achieve.

Firstly, it was necessary to analyze the tools that already create custom OS and two different types were found:

- Applications that create custom General Purpose Operating Systems;
- Applications that create custom Embedded Operating Systems;

These two groups will be analyzed in the next chapter, pointing to the main differences and finding the best combination to work with in order to develop an application that can be used for everyone.

The user needs and the most common features of an Embedded Project will also be taken into account in order to create an user friendly environment.

Chapter 2

State of the Art

In order to have a greater vision on what can be done it is crucial to scan the market to understand what has been done so far. Having access to the tools that are already popular and used for similar works can be enlightening for the project development and have direct impact on it.

Considering the goals and objectives of this dissertation, it is important to make the first step and understand the range of the project and its audience necessities. Understanding that the desired application of this dissertation is to obtain tailored final images for an OS to run on an Embedded System, makes it urgent to find out why do the existing distributions do not fulfill this request and what is missing to achieve this goal. Also it is indispensable to learn about tools that customize the final image and what are their suitable environments for deployment and future work.

But first it is necessary to understand the theoretical basis on which this dissertation stands. This chapter also gives the theoretical background, explaining some crucial concepts and how they work.

2.1 Basic concepts

In this section, some crucial concepts to this dissertation will be briefly explained. This way, all the subsequent explanation on how the thesis was implemented will be clearer.

2.1.1 Operating System

2.1.1.1 Kernel

Kernel is a program that constitutes the central core of a computer OS, it is the lowest level of easily replaceable software that interfaces with the hardware, having complete control over everything that occurs in the system. Kernel itself does not interact directly with the user but rather with the shell and other programs as well as with hardware devices on the system as it can be observed in the figure 2.1.

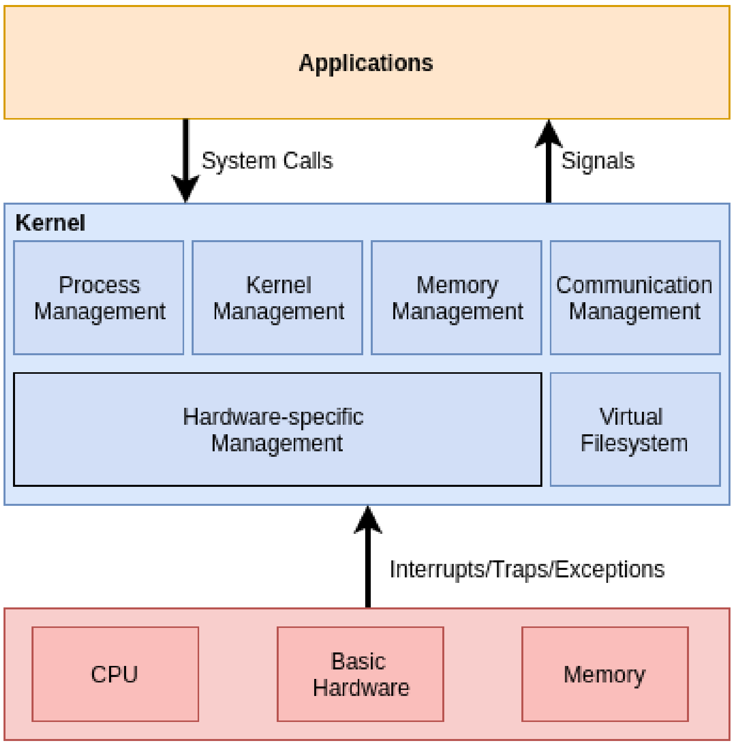


Figure 2.1: Kernel layout (adapted from [3]).

Kernel is the first part of the OS to load into memory during the startup, and it remains there for the entire duration of the session. Because of its critical nature, kernel code is loaded into a protected area of memory preventing it to be overwritten. Hence, the executed tasks are similarly divided into kernel space - which carries the processes executions and handles interruptions, and user space - running user applications that can't have access to kernel directly. This separation prevents user data and kernel data from interfering with each other and thereby reducing the performance level or causing the system to become unstable.

Kernel provides basic services for all the other parts of the Operating System, including the management of memory, processes, files and I/O devices.

Kernel's trivial components are:

- Scheduler - determines how the various processes share kernel's processing time;
- Supervisor - grants use of the computer to each process when it is scheduled;
- Interrupt Handler - handles all requests from the various hardware devices that compete for the kernel service;
- Memory manager - allocates the system address spaces.

Kernels can be classified in various different categories. The most used are monolithic and microkernel.

Monolithic kernel was built having all the basic system services like process and memory management and interrupt handling packaged into a single module in kernel space. This architecture is characterized by the huge size of kernel and poor maintainability. The modern approach to monolithic architecture allows different modules to be dynamically loaded and un-loaded, permitting to manage every module individually, increasing the maintainability of the project.

Microkernel resolves the problem of ever growing size of kernel code. This architecture provides only minimal services such as defining memory address spaces, inter process communication and process management. All other functions like driver management, protocol stack and filesystem run in user space. Using this method, the kernel code size is reduced and the security and stability are increased as only a minimum code runs in kernel space. The differences between these 2 types of kernel can be visualized in the figure 2.2.

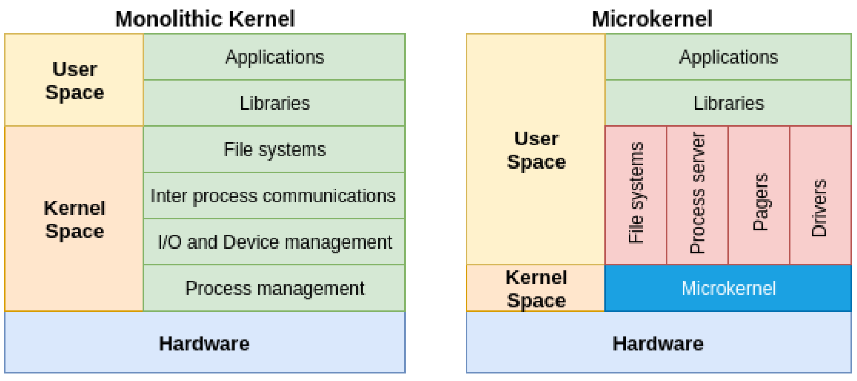


Figure 2.2: Overview of Monolithic Kernel (on the left) and Microkernel (on the right) (adapted from [4]).

2.1.1.2 Linux Kernel

Linux Operating system's kernel is monolithic, which means large size and great complexity. As it was referred before, in this type of kernel not only the CPU (Central Processing Unit), memory and IPC (Inter Process Communication) but also services like device drivers, filesystem management and system server calls are included in kernel space. This approach, although facilitating the communication between processes and allowing a more direct access to hardware, is unable do handle the general underlying desire to keep the kernel configuration as simple as possible. Monolithic's characteristic of being easy to design was the main reason to opt by this type and using this advantage some of the flaws were successfully overhauled by Linux kernel developers by creating kernel modules that can be loaded and unloaded in run time, meaning the possibility to add and remove features on the fly.[5] These kernel modules are also known as Loadable Kernel Module (LKM) which function is to add functionalities to the base kernel for things like devices, filesystems, and system calls. Additionally, unlike standard monolithic kernel, the device drivers can be preempted under certain conditions in order to handle hardware interrupts correctly.

Linux kernel is the most flexible OS in the market having over 7 million lines of code. Due to its malleability it can be tuned for a wide range of different systems. Using LKM it is possible to customize kernel for a specific environment, creating Operating Systems with different focus. For example, it is possible to remove support for the many different networked filesystems from an embedded device that has no networking support.[3]

2.1.1.3 Embedded Operating System

No specific kernel is available for embedded systems. Nevertheless, there are kernels specially configured/tailored for specific embedded hardware configurations.

Using these commonly configured kernels it is possible to create OS for embedded environments. These OS are specified to perform a particular task for a device that is not a computer and run the code for the device to do its job. The idea behind embedded systems is to be compact. Being limited in terms of functions, these systems may only run a single application, making every process crucial for the device operations. Hence, an embedded OS must be reliable and run the applications regarding constraints on memory, size and processing power.

What makes embedded systems so different is their diversity.[6] Embedded devices' hardware vary in both design and capability and these platforms contain a fair amount of custom, closed-source software which is not packaged by any distributor. In order to make the core functionality of the OS to work on a specific hardware, a BSP engineer must modify the low-level code. Also, getting all the necessary software components together to generate a Linux distribution for a particular embedded product can be a nightmare. Combining the effects of this hardware and software diversity, embedded systems present themselves as unique and extremely demanding systems. It is highly difficult to design a distribution which is simultaneously general enough to be useful and targeted to be efficient across a wide range of embedded platforms.

In order to simplify kernel's customization, various approaches are being developed by a number of embedded Linux projects generating a set of tools which may be brought to bear on a particular project according to its needs. These configurable build architectures have the capacity and flexibility to let the developer select, configure and build the sources by hand using a framework that hides details until the developer needs them. Buildroot is perhaps the simplest build framework project that automates the process of building a cross-compiling toolchain and a root fileystem for an embedded system.

2.1.2 Buidroot

Buildroot is a build system that allows embedded Linux developers to generate a working embedded Linux system almost from scratch. This tool is a set of makefiles that automate the process of downloading, configuring, compiling and installing all the available software packages. The dependencies are managed and cross-compiling issues already solved for most of the platforms.

Buildroot has the capability to automatically build the required cross-compilation toolchain, create a root filesystem, compile Linux kernel image and generate a bootloader for the target embedded system, as it can be observed in 2.3.

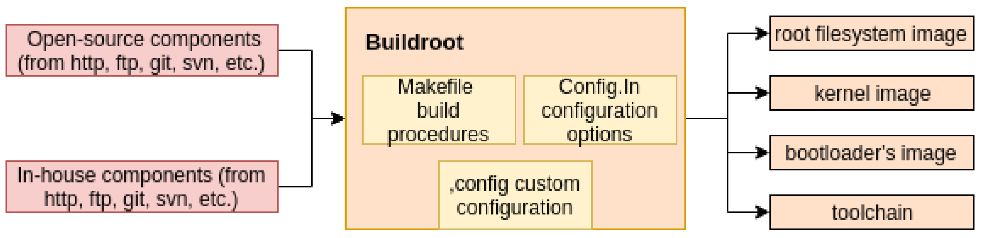

Open-source components (from http, ftp, git, svn, etc.)
In-house components (from http, ftp, git, svn, etc.)
Buildroot
Makefile build procedures
Config.In configuration options
,config custom configuration
root filesystem image
kernel image
bootloader's image
toolchain

Figure 2.3: Buildroot overview. (Adapted from [7]).

This build system was designed having simplicity in mind and it is best suited for small to medium-sized embedded systems. In the general root filesystem, Buildroot doesn't track which source packaged installed what. A full clean up is necessary in order to rebuild the file system.

This section will approach Buildroot's components and give a brief explanation about general files that make the customization of embedded OS possible.

2.1.2.1 Bootloader

The bootloader is a piece of code that runs before the Operating System. This code, usually small in size, has the specifications of target board and processor, preparing the necessary initialization for Operating System to come in. It usually starts from the flash memory (ROM).

Generally the bootloader is written to empower a controller with self-burning capabilities. The bootloader program has access to any of inbuilt peripherals like USB, USART, SPI, etc. Its first task is to map RAM to predefined addresses and after this function is done, the Stack Pointer is set up. "On a PC, it is used to boot the OS. In case of a microcontroller's bootloader, it enriches its capabilities and makes it a self-programmable device." [8]

Using Buildroot, the bootloader is controlled by config.txt and cmdline.txt which are files that teach the bootloader where to find kernel and which parameters to pass.

2.1.2.2 Makefile

Make file, typically designated as Makefile contains a set of directives used by a make build automation to generate a target/goal. It is a guide on how to compile and link a particular program. Buildroot's toplevel makefile handles the configuration and general composition of the build. Parallel execution of this Makefile is disabled because it changes the packages building order, that can be a problem due to package dependencies and in case two packages manipulate the same file in the target directory.

What does Buildroot's makefile do:

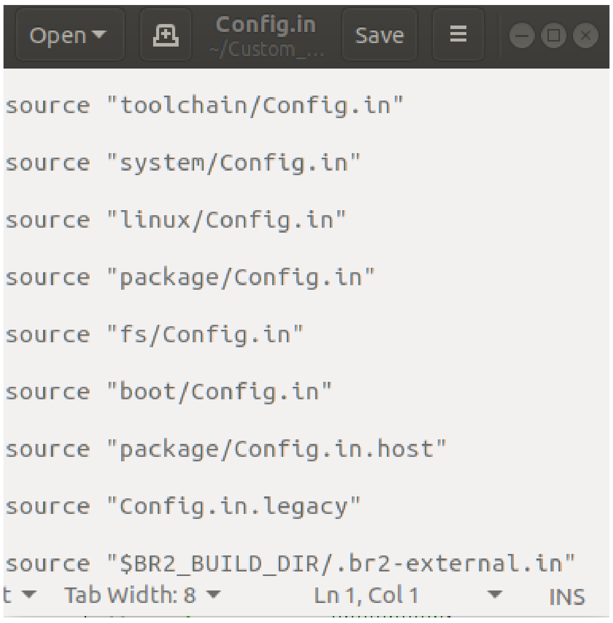
- Defines bash as shell;
- Sets output path (O=/path);

- Checks if current BR executions meets all the prerequisites. If not, buildroot will take care of it;
- Includes some helper macros and variables;
- Makes sure .config is overridden;
- Determines the userland (32bit or 64bit) the system is running on;
- Manages external trees.

The kernel is built with a system of individual makefiles that are all linked together when the kernel is built, forming a large makefile. The individual makefiles do not look like any standard makefile, but instead follow a special format that is unique to the kernel build process. The makefile needs to build only the necessary files, depending on the configuration options enabled, in the proper format.[9]

2.1.2.3 Configuration files

The configuration files are what defines the generated embedded operating system. The inmost configuration is kernel's and it is managed with KBuild. Kernel configurations divide themselves through modules and packages, meaning that each package has its own configuration foled, usually stored as *Config.in*. There is one main configuration file that sources all the necessary chain dependencies. In case of an external tree, its configuration file is also included. The *Config.in* file can be visualized in 2.4.



```
source "toolchain/Config.in"
source "system/Config.in"
source "linux/Config.in"
source "package/Config.in"
source "fs/Config.in"
source "boot/Config.in"
source "package/Config.in.host"
source "Config.in.legacy"
source "$BR2_BUILD_DIR/.br2-external.in"
```

Figure 2.4: Main Config.in file showing how all dependencies are included.

After kernel configurations, general configure file step in. The *.config* file holds all the options for the final image of the OS. The *.config* is big and hard to read, meaning it is necessary to recur to alternatives in order to modify it.

The first option is to create it from scratch using the *menuconfig* graphical window available in Buildroot in order to choose the desired options and tailor the final image. The process is not as simple as it sounds considering the vast range of options handed by Buildroot. It is necessary to specify the system's architecture, processor, CPU features and kernel options before building de image. A full and correct process requires knowledge and consumes time.

Another alternative is to build the image from default configurations available in Buildroot for a great range of architectures. These files, denominated as *defconfig* have the general architecture specifications for each board and also include some minimal features that do not have a default value. Using make technology, by just running *make BOARD_defconfig*, the pre selected options instantly overwrite *.config*, defining the features and packages included in the final image. This process is very simple as the hardest part is finding the architecture that suits the project, but the problem is the user do not have control over the *defconfig* file. The Operating System is committed to have the options specified in the file. It also possible to write your own *defconfig* file and include it to the list, yet this brings down the initial problem of advanced knowledge and time consuming.

2.1.2.4 BusyBox

BusyBox is a project initiated in 1996 to help the build of install disks for the Debian distributor. It was revived in 1999 by uClibc maintainer and since then it can be found in most of embedded Linux systems and all embedded Linux distributors.

Busybox is a collection of standard Linux utilities (e.g Shell, Init, etc.) optimized for low memory footprint systems. It contains everything that is needed in order to boot and verify if the system is working properly. The only missing link to being considered an OS is Kernel.

Busybox implements most Unix commands through a single execution file that has usually less than 1 MB (MegaByte). This executable acts as desired command by checking the name by which it was called. When a command is typed during the system's normal operation, BusyBox is invoked via the symbolic link. In turn, it determines the actually invoked command using the name being used to run it. E.g if the BusyBox is invoked through a symlink called *ls* then will act like *ls*.

Using BusyBox, users save enormous amount of space and, since it is not necessary to configure and build the sources of each tool, less time is consumed and not so great amount of knowledge is required.

Configurating the BusyBox:

- ***make menuconfig:*** text based interface for choosing the desired options;
- ***make defconfig:*** generic configuration enabling the most common options;
- ***make allnoconfig:*** configures only a strict minimum of options.

2.1.2.5 Shell Scripts

A script is a sequence of instructions that is interpreted or carried out by another program rather than by the processor, without being compiled. Shell scripts are POSIX (Portable Operating System Interface) compliant, meaning they follow a number of standards about how commands are executed in the shell. This ensures that the results of scripts used in OS can be predicted reliably and their behavior is kept within POSIX parameters.

Buildroot besides using *make *config* mechanism, also makes use of shell scripts to configure and customize the generated target filesystem, essentially using two scripts: *post-build.sh* and *post-image.sh*.

The *post-buid.sh*, as the name indicates, is a shell script that is run after Buildroot builds all the necessary packages but before rootfs files are assembled. Using this script it is possible to remove or modify any file in the target filesystem, however it should not be used to fix packages which either have unneeded files or were wrongly generated. Post-build script is run with Buildroot main tree as current working directory and can be enabled by setting *BR2_ROOTFS_POST_BUILD_SCRIPT* in the configuration file, specifying the path (may it be absolute or relative to the root of the Buildroot tree).

The *post-image.sh* are used to perform actions after the image has been created. It can be used to automatically extract the root filesystem into a chosen directory or to create specific firmware images that bundles the root file system or any other action specific to project. This script also run in the main Buildroot tree as current working directory and is set by the configuration option *BR2_ROOTFS_POST_IMAGE_SCRIPT*

, specifying the path. The post-image script will be executed, similarly to Buildroot, as not root. Therefore, actions requiring root permissions need special handling which is left to the developer.

2.1.3 Output

Buildroot is based on *KBuild* and *Make* technologies. As explained before, KBuild is responsible for kernel build and configurations. Make, in its turn, is responsible for the direct interaction with the user. The menu configuration windows and configuration files are run with *make* command. Make beholds the ability to build the entire system with all the previously chosen options.

After running *make* command, following steps will be performed:

- Download source files;
- Configure, build and install the cross-compilation toolchain, or simply import and external one;
- Configure, build and install selected target packages;
- Build a kernel image;
- Build a bootloader image;
- Create a root file system in selected formats.

All the outputs originated from *make* command are stored in an /output directory generated after this process, containing several subdirectories.

- **images/** where all the files that are necessary to place on the target system (kernel, bootloader and filesystem images) are stored;
- **build/** where all the components are built, containing one subdirectory for each of them;
- **staging/** that contains headers and libraries of the cross-compilation toolchain and all the userspace packages selected for the target;
- **target/** which contains practically the complete root filesystem for the target. However, it should not be used on the target system.
- **host/** contains the installation of tools compiled for the host that are crucial for the proper execution of Buildroot, including the cross compilation toolchain.

2.2 Linux From Scratch (LFS)

Linux From Scratch is a project that provides step-by-step instructions for building a custom Linux-based Operating System, entirely from source code created by Gerrard Beekmans and managed by Bruce Dubbs. This process is naturally a longer process than installing a pre-compiled distro but the advantages to this method are a compact, flexible and secure system. Also a greater understanding of how the Linux system works internally is granted. As it is said on the official website "Building LFS could be compared to a finished house. LFS will give you the skeleton of a house, but it's up to you to install plumbing, electrical outlets, kitchen, bath, wallpaper, etc. You have the ability to turn it into whatever type of system you need it to be, customized completely for you." [11]

LFS is not intended to be used by any kind of user, it takes some existing knowledge of Unix system administration in order to resolve problems and correctly execute the listed commands. Unlike installing a regular distribution which often comes with a lot of programs that won't ever be used but still take up precious disk space, the LFS system can be fully customized and produced as a very compact Linux System. The OS often have less than 100 MB and some deep tests show that with further stripping it is possible to bring the OS down to 5 MB.

The LFS system is built using an already installed Linux distribution. The existing Linux system will act as host and will be used as a starting point to provide necessary programs, including a compiler, linker and a shell to build the new system. It is assumed by selecting "development" option during the installation in order to have access to these tools. The host system is meant to have a completely blank hard disk with no existing partitions to start.

The primary target architectures of LFS are the AMD/Intel x86 (32-bit) and x86_64(64-bit) CPUs [12], although it is also known to work, with some modifications, with Power PC and ARM CPUs. It is also important to note that a 32-bit distribution can be installed and used as a host system on 64-bit AMD/Intel computer.

The LFS can have different focuses and create OS with different features based on the end-user's needs. With this purpose, many different projects were created under LFS.

- Beyond LFS - created in order to keep LFS small and compact. The instructions book presents the best way on how further develop the basic Linux system created with LFS;

- Cross LFS - focuses on cross-compile for headless or embedded systems that can run on Linux, but lack the resources needed to compile it;
- Hardened LFS - focuses on security enhancements such as hardened kernel patches, mandatory access control policies, stack-smashing protection and address space layout randomization.
- Automated LFS - designed to automate the process of creating a LFS system, aimed to reduce the amount of work included in the process.

Although being a toolkit that allows the user to fully customize the desired Operating System by editing its source code and also having focuses on different parts of an OS and making the user go through a learning process while developing the final output, this tool is not for everyone. The precise fact of having to edit the source code (what makes the final OS fully customized) is an obstacle as it reduces the scope this project hits. Not knowing how Linux works and not having the basic programming skills excludes the user right away.

2.3 OpenWRT

OpenWRT is an open source project for embedded systems based on Linux. Although the project started to develop custom firmware for consumer routers and was primarly used to route network traffic on embedded devices, the code was used as base for the creation of a Linux distributor that offers many features not previously found in consumer-level routers. Instead of single and static firmware, it provides a fully writable filesystem with package management. It was built from the ground up to be a full-featured, easily modifiable OS for embedded devices.

OpenWRT can be configured through either command line, which is simplified by the set of scripts provided by the project itself, or web-based interface (LuCI & Gargoyle) and has the ability to run on various types of devices.

As it was previously mentioned, OpenWRT features a fully writable root filesystem which enables end-users to modify any file and easily install additional software, unlike many other read-only firmware. This malleability and customization is obtained by overlaying the read-only files compressed by SquashFS with a writable JFFS2 using overlayfs, a union mount filesystem.

The OpenWRT development environment and build system is based on Buildroot (but heavily modified). It is a set of Makefiles and patches that automates the process of building a complete Linux-based OpenWRT system for an embedded device, using a cross-compilation toolchain. This toolchain is trivial since embedded devices usually use a different processor from the one in the host computer. OpenWRT build system makes it easy to port software and similarly to Buildroot uses kconfig (Linux Kernel menuconfig) for the configuration of desired features and although being designed for developers, the amount of abstraction given by the Makefiles and patches makes it easy to use for inexperienced users to build their own custom firmware.

OpenWRT owns a tool which is a pre-compiled environment suitable for creating custom images without the need to go through the entire compilation process. The tool is named **Image Generator** and is very useful for users that wish to fit more packages in a small flash size which is possible because the packages are embedded directly into the SquashFS. Building minimal images without the web interface saving manpower when flashing many devices by embedding packages and configuration files directly into SquashFS are other reasons for using this tool. While creating firmware images with OpenWRT, Image Generator is compulsorily created because it is needed to eventually create the image file. Fortunately, the project has the archive which contains Image Generator available and it is possible to download it alone.

The workflow of OpenWRT can be visualized in the figure 2.5.

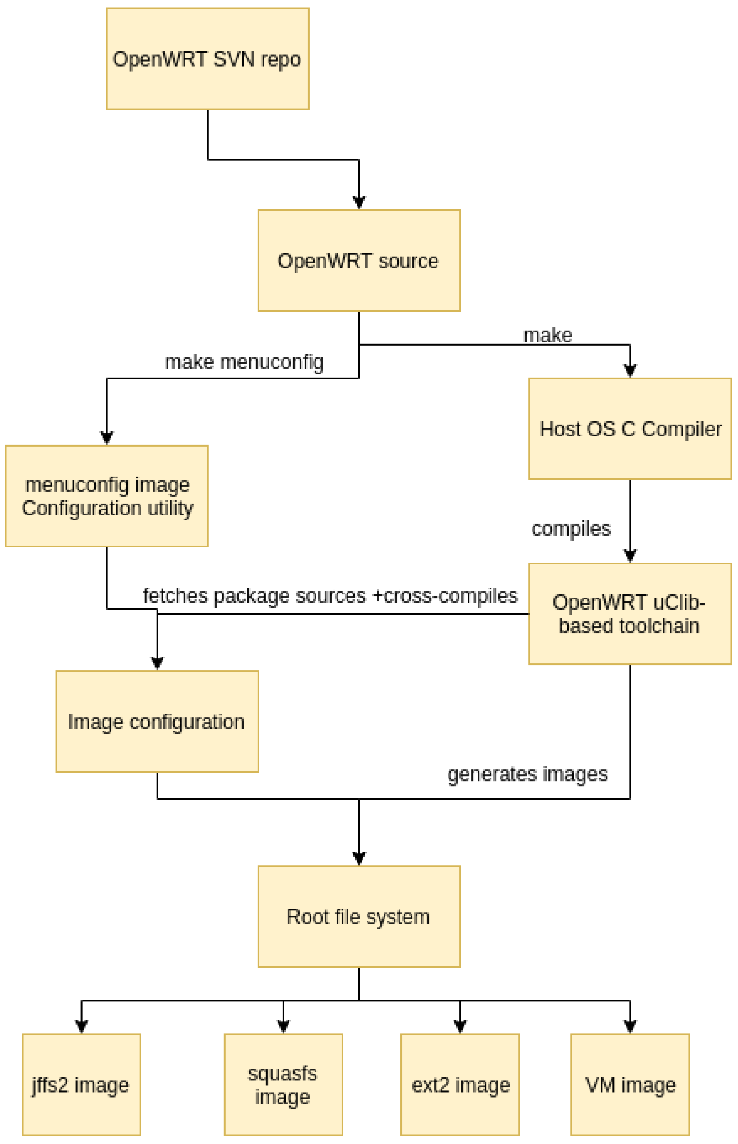


Figure 2.5: OpenWRT image generation workflow. (Adapted from [13]).

In conclusion, OpenWRT presents itself as a strong Linux-based distribution for embedded systems. It has plenty of user-friendly interfaces and also the possibility for users to follow a learning path by following the command line customization. The main target are embedded systems, the generated images are small in size but the focus of this project are mainly network. As it was created for router optimization even after rewriting of the source code and forks into other projects that have the main goal to build customized OS, the customization lie on networking features. So unless the user wish to develop a project that requires deep network usage, it is possible to find more useful general purpose distros.

2.4 Yocto Project

Yocto Project is an Open Source project kept by Linux Foundation. It is essentially a collection of recipes, configuration values and dependencies used to create a custom Linux runtime image tailored do user's specific needs. The project provides interoperable tools, metadata and processes that enable the rapid and repeatable development of Linux-based embedded systems, allowing to customize every aspect of the process. Yocto Project combines the convenience of a ready-to-run Linux distribution with the flexibility of a custom Linux OS stack and, although being roughly analogous to a Desktop Linux distribution it isn't one, it creates a custom one for embedded environment instead. In traditional desktop Linux distribution models, the installation is generally done from a CD or USB key and then the additional package installations and configurations are performed in the running target systems. The Yocto workflow differentiates itself by executing a full build on the Host, the output of which is an image containing the entire target system. With a bit of care, it is possible to eliminate most, if not all, of the configuration steps required in the running target allowing for a more predictable software load and reducing the number of dimensions in test matrix which to base their activities as a function of their needs.

Yocto has its uniqueness because of the Yocto Layer Infrastructure which is an integral part of its model. All the functionalities are divided into different layers and added to build only when required. This process reduces the complexity of each layer and also grants the opportunity of developing each one individually at its own pace. The layers are added with priority ordering which allows the hierarchically higher layers to override and modify the base layers. Another advantage of using this model is the logical separation made in build stage as building every feature in the same layer can seem easier and more perceptive, the separation shortens future customization and reuse.

As mentioned before, Yocto Project is a colletion of tools to generate a tailored Linux-based OS. A key part of this is the OpenEmbedded build system, which enables developers to create their own Linux distribution specific to their environment. Yocto Project and OpenEmbedded share maintainership of the main parts of the OpenEmbedded build system: the build engine - **BitBake**- and the core metadata - **OpenEmbedded-Core**. It is also provided a reference implementation called **Poky**, containing the OpenEmbedded build system plus a large set of recipes, arranged in a hierarchical system of layers that can be used as fully functional template for a customized embedded OS.

The Yocto Project's structure can be visualized in the figure 2.6.

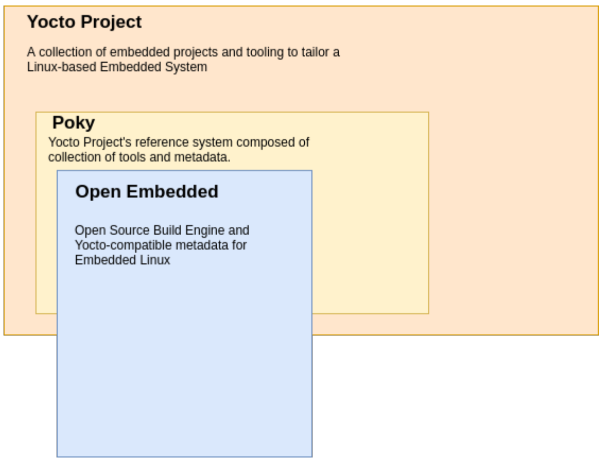

Yocto Project
A collection of embedded projects and tooling to tailor a Linux-based Embedded System
Poky
Yocto Project's reference system composed of collection of tools and metadata.
Open Embedded
Open Source Build Engine and Yocto-compatible metadata for Embedded Linux

Figure 2.6: Yocto Project content. (Adapted from [14]).

2.4.1 OpenEmbedded

OpenEmbedded is a build automation framework and cross-compile environment used to create Linux distributions for embedded systems. It is the build system of the Yocto Project, consisted of some crucial components associated to the workflow of the system.

BitBake is a core component of the Yocto Project used by OpenEmbedded system to build images with special focus on embedded Linux cross-compilation. It is a generic task execution engine that allows shell and Python task to be run efficiently and simultaneously while working within complex inter-task dependency constraints. BitBake is composed of recipes which store metadata and specify how each package is built. It also has descriptive information about the package, existing dependencies and is where the source code resides.

OpenEmbedded-Core is a common layer of metadata, containing base layer of recipes, classes and associated files, used by OpenEmbedded-derrived systems, which includes Yocto Project.

Yocto's workflow can be observed in the following figure 2.7.

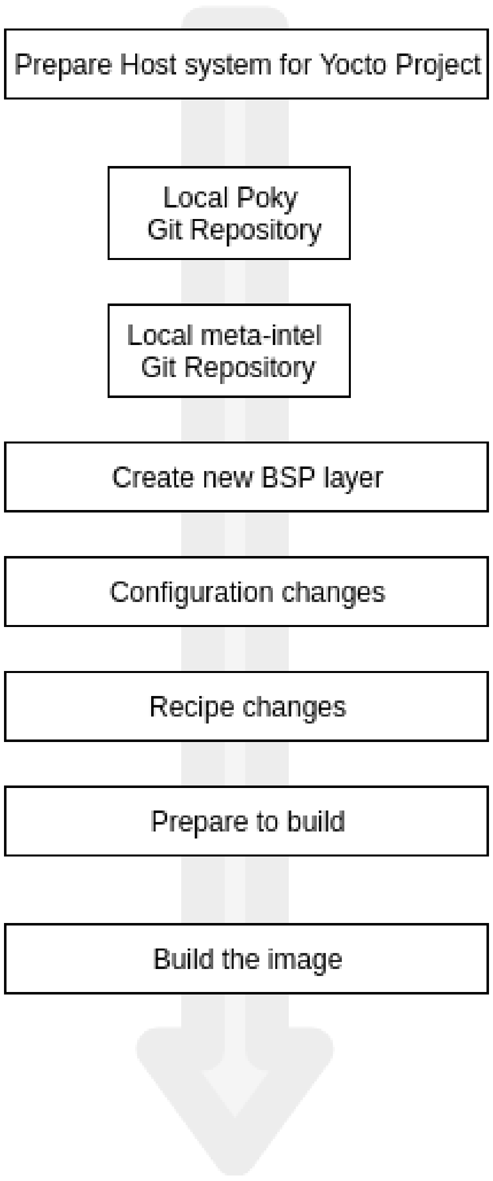

Prepare Host system for Yocto Project
Local Poky Git Repository
Local meta-intel Git Repository
Create new BSP layer
Configuration changes
Recipe changes
Prepare to build
Build the image

Figure 2.7: Yocto Project workflow. (Adapted from [14])

2.4.2 Poky

Poky is a reference distribution of the Yocto Project and is composed of collection of tools and metadata. It is platform-independent and performs cross-compiling using BitBake and OpenEmbedded-Core, mentioned before, and a default set of metadata. In other words, Poky is a base specification of the functionality needed for a typical embedded system as well as the components from the Yocto Project that allow the user to build a distribution into a usable binary image. The main objective of Poky is to provide all the features an embedded developer needs.

Poky is sometimes mentioned as being a ”default configuration”. It is possible to use Poky to create an image ranging from a shell-accessible minimal image all the way up to a Linux Standard Base-compliant image. One of the most powerful properties of Poky is that every aspect of a build is controlled by the metadata. It is possible to use metadata to augment these base image types by adding metadata layers that extend functionality. These layers can provide, for example, an additional software stack for an image type, add a BSP for additional hardware or even create a new image type.

There is fairly steep learning curve with Yocto. The terminology can be daunting as the difference between components can be unclear. The number of options for configuring the target can make it difficult to assess the best choices. Creating a basic default system can be achieved quickly by following the tutorials. However, understanding what changes must be done for a particular design may require a nontrivial amount of research and investigation.

2.4.3 Toaster

In order to offload the end user and make the system more perceptive, a web interface to OpenEmbedded and BitBake was created. This build system, used by Yocto Project, allows the user to configure and run builds, and provides information and statistics about the build process.

Toaster can be used in either Analysis or Build Mode.

- **Analysis Mode:** It is possible to record builds and statistics. In this mode, the *bitbake* command used to build images, is directly accessed by initiating the build using *bitbake* command from the shell. Analysis Mode is capable of keeping track of recipes and packages installed to the final image; browsing the directory structure of the image; accessing the values of all variables in the build configuration and performance information such as build time, task time, CPU usage, and disk I/O. The figure 2.8 shows this mode's workflow.

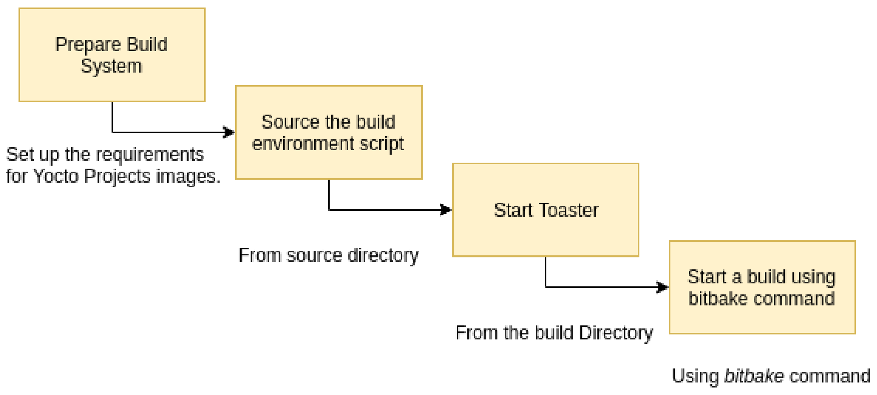

Prepare Build System
Set up the requirements for Yocto Projects images.
Source the build environment script
From source directory
Start Toaster
From the build Directory
Start a build using bitbake command
Using *bitbake* command

Figure 2.8: Analysis Mode workflow

- **Build Mode:** Toaster handles the build configuration, scheduling and execution. In this mode, all interactions with the build system happen through the web interface and *bitbake* command isn't directly accessed. Using this mode the build is configured and started within Toaster's GUI. In Build Mode, Toaster is capable of browsing layers listed in the various layer sources that are available; import and manage custom layers; set configuration variables; select one or multiple targets to start the build. This mode's workflow can be observed on the following figure 2.9.

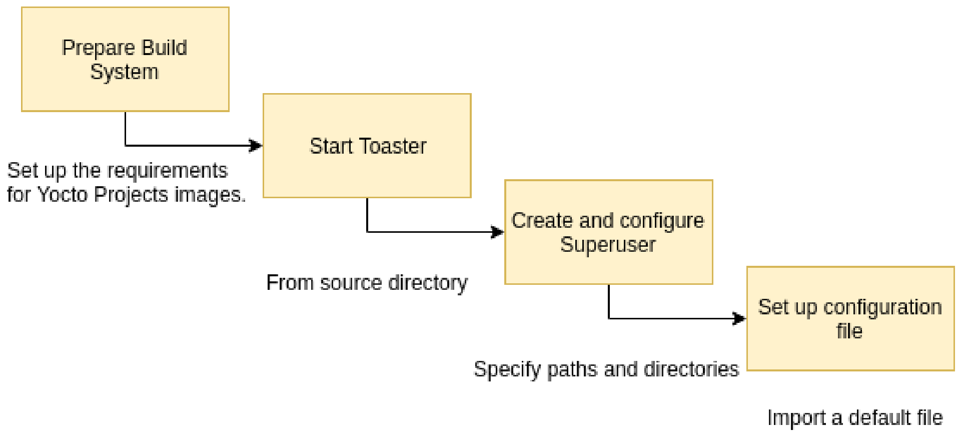

Prepare Build System
Set up the requirements for Yocto Projects images.
Start Toaster
From source directory
Create and configure Superuser
Specify paths and directories
Set up configuration file
Import a default file

Figure 2.9: Build Mode workflow

2.5 COSMOS

C# Open Source Managed Operating System (COSMOS) is a toolkit for building Operating Systems in an user-friendly way using any .NET language. This toolkit uses Microsoft Visual studio as development environment in which the Operating System source code is customized and the OS is created following the normal application process.

Cosmos is available in two different kits: development kit (dev-kit) and user-kit. The dev-kit might be thought as COSMOS SDK (Software Development Kit), designed for users to work with COSMO. It is a consistently updated open source code. The user-kit is a stable but not often updated version designed to build an Operating System as an application using Visual Studio, it is a part of COSMOS designed to make it easier for developers to use. It adds a new project type called COSMOS project to Visual Studio which is a modified version of a console application with the compiler and bootup already added. Its workflow can be analyzed in the following figure 2.10.

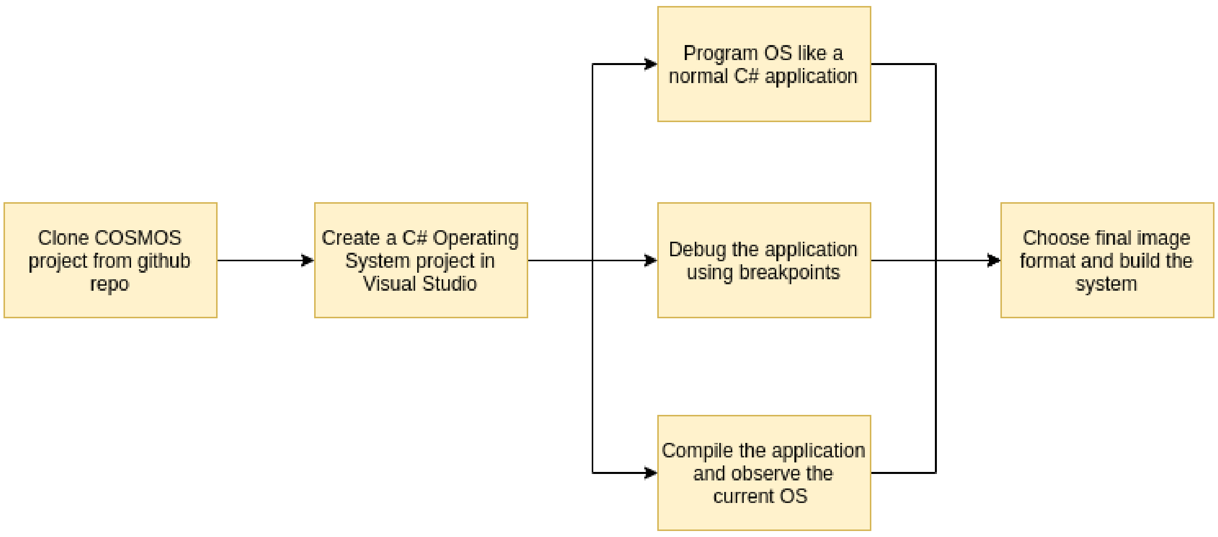

Clone COSMOS project from github repo
Create a C# Operating System project in Visual Studio
Program OS like a normal C# application
Debug the application using breakpoints
Compile the application and observe the current OS
Choose final image format and build the system

Figure 2.10: COSMOS workflow.

Debugging is a major issue to consider while developing an Operating System [10]. By integrating COSMOS and the Visual studio it is possible to debug the OS using breakpoints. Since the debugger uses the serial port to communicate, debugging only works with virtualization environments that support serial ports such as VMWare and QEMU (Quick Emulator).

COSMOS uses its own developed compiler - IL2CPU - to translate Common Intermediate Language (CIL) into machine code (x86 opcode mostly). This compiler, when invoked, systematically scans through all the application's CIL code , converting it into assembly language for the selected processor architecture. COSMOS invokes the selected assembler to convert the assembly language into native CPU opcode. Finally the desired output option is activated producing an ISO disk image file.

2.6 Remastersys/Respin

Remastersys is a free and open-source program for Debian, Ubuntu-based, Linux Mint or derivative software systems. With its usage it is possible to create a customized LIVE CD (Compact Disk)/LIVE USB (Universial Serial Bus) of Debian and its derivatives by cloning your current system.

Remastersys is intended to be an easy way to create a customized Ubuntu-based OS on a CD/DVD. After installing all the necessary packages (and getting rid of all the unnecessary ones by running *aptitude clean* or *remastersys clean*) the *remastersys dist* is ready to run. Using this simple distributor it is possible to customize only a small portion of the OS within an interface provided by the program. Before running the command to create the OS it is advised to update the system and remove any unnecessary applications to save room. The generated OS will not have all the personal data included on the disk as anything in /home will be excluded. Any desired files should be moved to the respective folder in /usr/share. All the further customization can be done by editing the .config file.

After choosing and customizing the features, a CDFS (Compact Disk Filesystem) is created which eventually is converted into a dist ISO that can be tested using a virtual machine before burning it on any CD. The generated OS are not meant for embedded use and their size can go up 4GB.

In April 2013 the originator's direct development has ceased but a group of developers forked this project into **Respin** in order to continue the development. This toolkit is only available for Debian users though. The process of building the OS is very similar to Remastersys as the final output will be an ISO file of the Debian system.

The procedure is to download all the required packages and install Respin and its dependencies. After the installation and customization of the OS, several options for backup and iso creation are availabe:

- *repsin backup* to make a live CD/DVD iso called "custom.iso";
- *respin backup custom.iso* to create a live CD/DV of the system;
- *respin dist* to create a live CD/DVD of the system - filesystem only;
- *respin dist cdfs* to make a distributable iso, but only if cdfs exists; content...

2.7 Linux Live Kit

Linux Live Kit present itself as the most innovative toolkit available to generate customized Operating Systems. It is a set of shell scripts that turn the existing pre installed OS distribution into a Live Kit. It is possible to make a Live Kit out of the main installation on the PC but it is recommended to install a new OS to start. Although Debian is the recommended OS, any distribution can be used for this process.

After choosing the desired distribution, its installation must be done to disk partition. Some packages and kernel modules are required such as aufs, that combines multiple directories into one that appears to contain their combined content, and squashfs for file compressing. Since the generated live distro have the same content as the one installed it is necessary to proceed to the removal of all the unnecessary files so the final image is kept as small as possible.

Following the customization of the OS, it is required to download the Linux Live Kit fro GitHub. All the downloaded files must be stored in /tmp. The next step is simply running the shell script as root user, generating two types of images:

- .iso file - which can be burnt to a CD or tested from a Virtual Machine;
- bootable usb flash - that must be unzipped to an USB flash drive which is made bootable by running the bootinst.sh shell script.

2.8 Ubuntu Imager

Distroshare Ubuntu Imager is another project that creates an installable Ubuntu Live ISO from a pre installed Ubuntu or derivative distribution. It is developed by Distroshare which is a website for sharing customized open source operating system distributions. The developers main idea is to assist users that intend to run a customized open source OS to work correctly on their machines.

Similarly to Remastersys and its forks, Ubuntu Imager decided to automate the process by putting it all into a bash script that does the bulk of the work. With minimal configuration it is possible to create a live environment with as little as editing a text file and running a shell script.

The Distroshare Ubuntu Image script is hosted on GitHub and can be cloned from its official repository. After downloading and unpacking the zip, the process is ready to use.

From all the downloaded files it is important to focus *./distroshare-ubuntu-imager.config* that contains all the configurations of the system. By editing this .config file it is possible to customize the OS by specifying the kernel version, directory of the final output image and other parameters. The lines of the .config file are self explanatory and it takes no effort for the end user to understand and customize it in the way he wants.

Next it is needed to run the script as root user. The script will read the configuration details that were specified in the .config file and all necessary directories will be created as well as the required software will be installed. Then, the temporary files are cleared and the ubiquity is removed from the system compressing the new filesystem and putting it into an ISO file.

The final ISO image that has been created can be burned onto a CD, mounted to a virtual machine or run directly from grub. It is also possible to copy the image to a USB drive using dd.

\{}section{COSMOS} \{}par C\{}# Open Source Managed Operating System (COSMOS) is a toolkit for building Operating Systems in an user-friendly way using any .NET language. This toolkit uses Microsoft Visual studio as development environment in which the Operating System source code is customized and the OS is created following the normal application process.

\{}par Cosmos is available in two different kits: development kit (dev-kit) and user-kit. The dev-kit might be thought as COSMOS SDK (Software Development Kit), designed for users to work with COSMO. It is a consistently updated open source code. The user-kit is a stable but not often updated version designed to build an Operating System as an application using Visual Studio, it is a part of COSMOS designed to make it easier for developers to use. It adds a new project type called COSMOS project to Visual Studio which is a modified version of a console application with the compiler and bootup already added. Its workflow can be analyzed in the following figure \{}ref{fig:cosmosforkflow}. \{}begin{figure}[h!] \{}centering \{}include-graphics[width=1\{}linewidth]{/Images/StateOfTheArt/cosmosworkflow.png} \{}caption{COSMOS workflow.} \{}label{fig:cosmosforkflow} \{}end{figure} \{}par Debugging is a major issue to consider while developing an Operating System \{}cite{cosmos}. By integrating COSMOS and the Visual studio it is possible to debug the OS using breakpoints. Since the debugger uses the serial port to communicate, debugging only works with virtualization environments that support serial ports such as VMWare and QEMU (Quick Emulator).

\{}par COSMOS uses its own developed compiler - IL2CPU - to translate Common Intermediate Language (CIL) into machine code (x86 opcode mostly). This compiler, when invoked, systematically scans through all the application's CIL code , converting it into assembly language for the selected processor architecture. COSMOS invokes the selected assembler to convert the assembly language into native CPU opcode. Finally the desired output option is activated producing an ISO disk image file.

\{}section{Remastersys/Respin}

Remastersys is a free and open-source program for Debian, Ubuntu-based, Linux Mint or derivative software systems. With its usage it is possible to create a customized LIVE CD (Compact Disk)/LIVE USB (Universial Serial Bus) of Debian and its derivatives by cloning your current system. \{}par Remastersys is intended to be an easy way to create a customized Ubuntu-based OS on a CD/DVD. After installing all the necessary packages (and getting rid of all the unnecessary ones by running \{}textit{aptitude clean} or \{}textit{remastersys clean}) the \{}textit{remastersys dist} is ready to run. Using this simple distributor it is possible to customize only a small portion of the OS within an interface provided by the program. Before running the command to create the OS it is advised to update the system and remove any unnecessary applications to save room. The generated OS will not have all the personal data included on the disk as anything in /home will be excluded. Any desired files should be moved to the respective folder in /usr/share. All the further customization can be done by editing the .config file.

After choosing and customizing the features, a CDFS (Compact Disk Filesystem) is created which eventually is converted into a dist ISO that can be tested using a virtual machine before burning it on any CD. The generated OS are not meant for embedded use and their size can go up 4GB.

\{}par In April 2013 the originator's direct development has ceased but a group of developers forked this project into \{}textbf{Respin} in order to continue the development. This toolkit is only available for Debian users though. The process of building the OS is very similar to Remastersys as the final output will be an ISO file of the Debian system. \{}par The procedure is to download all the required packages and install Respin and its dependencies. After the installation and customization of the OS, several options for backup and iso creation are availabe: \{}begin{itemize} \{}item \{}textit{repsin backup} to make a live CD/DVD iso called ”custom.iso”; \{}item \{}textit{respin backup custom.iso} to create a live CD/DV of the system; \{}item \{}textit{respin dist} to create a live CD/DVD of the system - filesystem only; \{}item \{}textit{respin dist cdfs} to make a distributable iso, but only if cdfs exists; content... \{}end{itemize}

\{}section{Linux Live Kit}

Linux Live Kit present itself as the most innovative toolkit available to generate customized Operating Systems. It is a set of shell scripts that turn the existing pre installed OS distribution into a Live Kit. It is possible to make a Live Kit out of the main installation on the PC but it is recommended to install a new OS to start. Although Debian is the recommended OS, any distribution can be used for this process. \{}par After choosing the desired distribution, its installation must be done to disk partition. Some packages and kernel modules are required such as aufs, that combines multiple directories into one that appears to contain their combined content, and squashfs for file compressing. Since the generated live distro have the same content as the one installed it is necessary to proceed to the removal of all the unnecessary files so the final image is kept as small as possible. \{}par Following the customization of the OS, it is required to download the Linux Live Kit fro GitHub. All the downloaded files must be stored in /tmp. The next step is simply running the shell script as root user, generating two types of images: \{}begin{itemize}

\{}item .iso file - which can be burnt to a CD or tested from a Virtual Machine; \{}item bootable usb flash - that must be unzipped to an USB flash drive which is made bootable by running the bootinst.sh shell script. \{}end{itemize}

\{}section{Ubuntu Imager} Distroshare Ubuntu Imager is another project that creates an installable Ubuntu Live ISO from a pre installed Ubuntu or derivative distribution. It is developed by Distroshare which is a website for sharing customized open source operating system distributions. The developers main idea is to assist users that intend to run a customized open source OS to work correctly on their machines. \{}par Similarly to Remastersys and its forks, Ubuntu Imager decided to automate the process by putting it all into a bash script that does the bulk of the work. With minimal configuration it is possible to create a live environment with as little as editing a text file and running a shell script. \{}par The Distroshare Ubuntu Image script is hosted on GitHub and can be cloned from its official repository. After downloading and unpacking the zip, the process is ready to use. \{}par From all the downloaded files it is important to focus \{}textit{./distroshare-ubuntu-imager.config } that contains all the configurations of the system. By editing this .config file it is possible to customize the OS by specifying the kernel version, directory of the final output image and other parameters. The lines of the .config file are self explanatory and it takes no effort for the end user to understand and customize it in the way he wants. \{}par Next it is needed to run the script as root user. The script will read the configuration details that were specified in the .config file and all necessary directories will be created as well as the required software will be installed. Then, the temporary files are cleared and the ubiquity is removed from the system compressing the new filesystem and putting it into an ISO file. \{}par The final ISO image that has been created can be burned onto a CD, mounted to a virtual machine or run directly from grub. It is also possible to copy the image to a USB drive using dd.

2.9 Debian Live

Debian Live is a project by Debian team that produces tools to create official Debian Live images and also maintains other packages that are used to build live images.

Although the existence of plenty Debian based live systems available in the market, none of them is an official version. By that it is meant the lack of support and the vast majority is just a mix of different distributions that only support i386 (32-bit AMD) and also have additional (and unofficial) patches on kernel. Debian live on the other side uses only official and verified packages from Debian repository.

The requirements for generating a live OS with Debian is

- Root access;
- Up-to-date live-build;
- POSIX-compliant shell (bash or dash).

The first step is to install live-build which is a set of scripts to build live system images. The idea behind live-build is a tool suite that uses a configuration directory to completely automate and customize all aspects of building a Live image. Subsequently comes the customization that is done by editing the auto/config script that allows the user to choose the desired configurations by writing the key words and selecting the pretended options. After this step the config file is generated and the local package list is populated automatically. The build comes subsequently, controlling the generatiion of live images for further usage.

It is possible to optimize the image making use of tradeoffs between size and functionality. The default generated image has approximately 192 MB. The minimal obtained size is 91 MB that presumes the removal of documentation and other file packages. However it violates the integrity of those packages what has unforeseen consequences. In order to obtain a smaller image without forcing the system it is needed to use an additional tool: **minimal debootstrep**.

2.9.1 Debootstrap

Debootstrap is a tool which installs Debian based systems into a sub directory of another already installed system. It gives the user a simple and consistent way to cleanly setup machines by installing packages straight into the filesystem directly from the repository. It installs Debian in a system without using a installation disk but can also be used to run a different Debian flavor in a chroot environment.

This way you can create a full (minimal) Debian installation which can be used for testing purposes.

Once again, the main focus on this live image distributions isn't the embedded systems. The control over the features does not allow the image to be small and malleable in the pretended way.

2.10 KIWI

KIWI is an open source project that represents a back-end command line tool, written in Perl, with the goal of building Linux-based images. It allows the user to configure, build, and deploy your own OS images in a vast variety of formats. KIWI's workflow is divided into 2 stages:

- **Preparation:** this is the stage where the root directory holding the contents of the new filesystem is created, all the necessary packages are installed and the image description file is created and optionally customized.
- **Creation:** in this stage the image itself is created using the unpacked root tree that was previously generated. The image creation process does not require any user interaction but it can be tuned by modifying the image.sh which is the script that is called during the creation option.
- **Deployment:** The final image can be deployed using various different methods, be it live CD or USB flash drive.

KIWI holds different configuration files and shell scripts, being config.xml the main configuration file containing the most important aspects of the image such as image type, base name and other options.

The images.sh is a clean up script that runs before the image creation process is started, it removes files that are only needed while the physical extent exists. Config.sh is the configuration script that is used to tailor the final image. This script is executed at the end of image installation and is responsible for the activation/deactivation of features.

As mentioned before, KIWI is a command line tool and all the configuration and customization is done through shell scripts. Although being perceptive, this tool requires some additional knowledge and can be an obstacle to certain users. In order to assist this back-end engine and increase the scope of this project, **SUSE** developed a front-end image creator.

With the GUI (Graphical User Interface) designed by SUSE, user's mission is facilitated as the image customization, build and deployment are just one click away. When creating an image, the system allows user to build it based on a template. Many templates with different focuses are available depending on the OS final goal. For more advanced systems it is possible to tailor the OS and add/remove all the necessary features by just selecting a checkbox with the respective name. After choosing the output path and the image type, it is only necessary to wait for the image to be generated.

Using KIWI, the end-user has full control of the generated image by choosing the necessary features to run in OS. In combination with openSUSE, the system can be easily tailored with no additional effort from user. However, the generated OS is large in size as it is not suitable for embedded systems but only for Desktop.

2.11 Conclusion

Having taken into consideration all the available solutions for the problem carried by this dissertation, it is possible to evaluate its importance and novelty.

After examining plenty of options, the conclusions are that the current mechanisms available on the market divide themselves into two different groups:

The first group consists of projects that are capable of generating and distributing images for an embedded environment, but on the other side lack malleability and have a steep learning curve . It is possible to build and deploy Operating Systems for embedded devices from different templates and choose only the features that are necessary for the system to run. However, this action isn't done in a user-friendly way and assumes that the user has some additional knowledge.

The tools from the second group offload any addition work from the end-user and permit a tailored image creation within few clicks. Although a user-friendly environment to customize the Operating System is granted, the final images are abundant in size and aren't suitable for embedded environments. The generated images fulfill the requirements for Desktop Operating Systems and can be used for this purpose but aren't capable of being compatible with hardware-dependent components on a embedded device.

This chapter is essential to understand the importance of this dissertation as it leans to combine the advantages of both groups that have been identified. The final project should be able to generate tailored images for embedded devices with a low learning curve, increasing the target audience of the project.

Chapter 3

System Specification

After having a theoretical insight on the concepts and aspects focused in this dissertation and analyzing different tools that are used in the market to reach similar goals, it is possible to define the components that will be part of the system.

First and foremost, it is important to outline that this dissertation's goal is to create tailored images, using Buildroot, for Raspberry Pi and its Compute Module family. However, the portability to other boards can be easily done by creating default configuration files for requested architectures without making any changes in code or structure of the application.

This chapter presents the design and specification of the system and target's architecture as well as constraints and requirements for the dissertation.

3.1 System Requirements

In order to properly design and conceive the system, it is necessary to define all the requirements and constraints beforehand by visualizing the project from several points of view. It is crucial to define both functional and non-functional requirements to fulfill them in the decision making process.

- Create embedded images for kernel, bootloader and root filesystem.

  The system must be able to generate all the necessary images for the embedded OS to run properly. The kernel image is usually compressed into a zImage binary file and the root filesystem format can be chosen by the user.

- Tailor the embedded OS image

  The user must be able to customize the desired OS in order to fulfill the embedded system's requirements.

- Have a minimal base image for the OS

  All the user's customization must be written upon a default OS image that is minimally configured by choosing only the crucial components common to most of embedded systems.

- Suit the intended architecture.

  The system's final image must be suitable for the architecture chosen by the user.

- Be user friendly

  The system must be user friendly by abstracting the final user from all the back-end processes such as installing the necessary packages, configuring the system and scripting. The user's only job is to choose the desired features to include in the Operating System.

Having specified how the system should behave, it is important to understand the characteristics that the final project should carry. Since the dissertation's main goal is to abstract the user from all the back-end procedures and to only choose the desired features through a graphical interface, the system must contain **Usability** as non-functional requirement. The application developed by this dissertation must run on every computer, independently of the architecture and containing OS, granting **Portability**. Although the main focus is held on Raspberry Pi family, the transition for other architectures must be easy by only editing the default configuration file and specifying all the necessary information - **Scalability**. Since this Dissertation uses Buildroot as back end including all the features it has, **Security** is granted by Buildroot. Other security issues are out of scope of this Dissertation.

3.2 System Architecture

As mentioned in previous chapters, the main goal of this dissertation is to create customized Operating Systems for embedded environments that will later be used by developers for their specific projects. Taking into account all the specifications and requirements of this dissertation, it is possible to define the field where it will act.

A generic embedded system presents the lower layer, where all the hardware is located; a mid layer that contains the operating system that will process all the incoming information from the hardware; and the top layer that represents the system's direct interaction with the user. This system's focus is in the area between the lower and mid layer, right upon HAL (Hardware Abstraction Layer), in the lowest level of easily replaceable software that interfaces with the hardware. Usually, the mid layer is the Operating System that has all the necessary features to process the information coming from lower layer and send it, in the pretended way, to the upper level. This layer obtained its capabilities from an OS creation process previously done where all the necessary features where chosen and built alongside kernel and bootloader, like it was exposed in the first chapter's basic concepts (Figure 3.1).

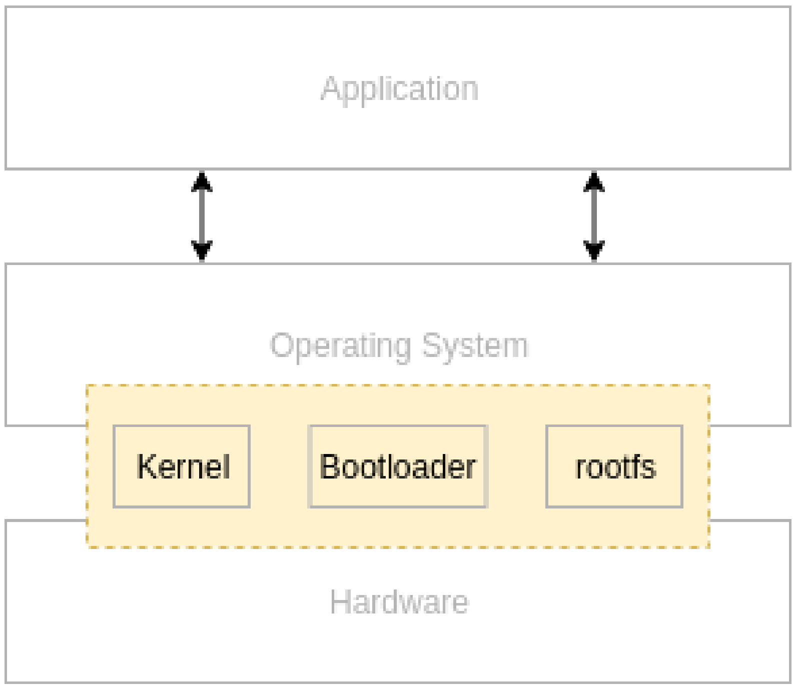


Figure 3.1: Generic embedded system architecture, showing where this Dissertation's final application will act

A better understanding of how the system works can be visualized on the figure 3.2:

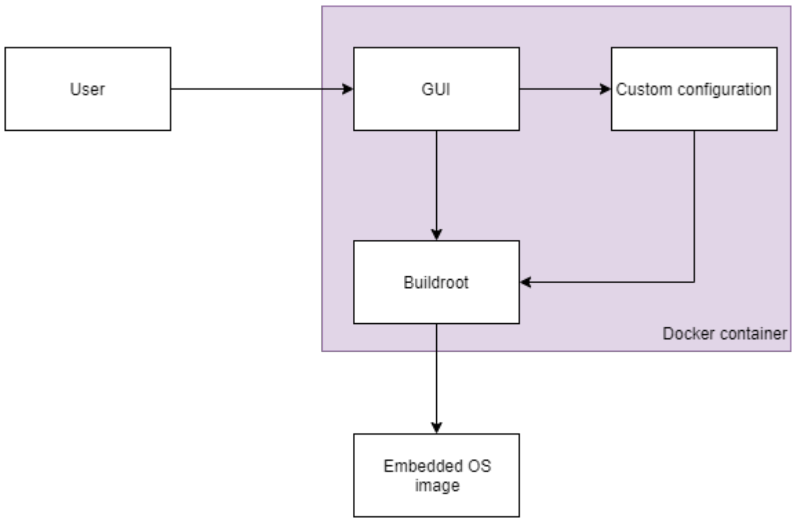


Figure 3.2: System's architecture

3.3 Hardware Specification

After specifying the system requirements and and placing this project in a typical embedded system architecture, it is possible to specify the hardware that was used in this Dissertation. Considering that to achieve the main goal, no specific hardware components were used as all the objectives are fulfilled through software, the only hardware that was used and can be specified are the target platforms.

3.3.1 Raspberry Pi 3 model B

The Raspberry Pi 3 model B is the third generation Raspberry Pi. This open-source, Linux based, credit card sized computer board can be used for many applications and whilst maintaining the popular board format of previous versions, it brings a more powerful processor which is 10x faster than the one from the first generation. Additionally it adds wireless LAN and Bluetooth connectivity, making it the ideal solution for powerful connected designs. This platform was chosen for the reason that it was frequently used and it is a very known board. Previous projects developed on this board through the master course demanded the creation of an OS with Buildroot, making it a recommendable choice for the initial tests and the configuration of a minimal OS image.

Some of Raspberry Pi 3 model B characteristics are:

- Broadcom BCM2387 chipset;

- 1.2GHz Quad-core ARM Cortex-A53;
- 1GB Low-Power DDR SDRAM;
- 64-bit CPU;
- -20°C to +60°C temperature range.

3.3.2 Compute Module 3

The Compute Module is a Raspberry Pi in a more flexible form factor, intended for industrial application. It contains the same processor and RAM as Raspberry Pi and additionally it has a 4GB eMMC flash device that is the equivalent for the SD (Secure Digital) card. All of this is integrated on to a small 67.6mm x 31mm board.

Compute Module is designed for use in custom devices where the capabilities of RPi (Raspberry Pi) are desired, but its size and layout are unwanted. With CM (Compute Module) it becomes possible to design custom boards where the Raspberry Pi is just another component. That delivers an enormous amount of flexibility as it gives access to a greater number of GPIO (General Purpose Input/Output). The on-board eMMC eliminates the need for an external micro SD card, making the Compute Module perfect for designing new products.[15]

This module is a sublime combination with the project that this Dissertation tries to the develop for the reason that the chosen features for the embedded OS are supported by the customized hardware that is added to the Compute Module. These reasons make this platform elected and predominant to test the final project.

Specifications of Compute Module 3:

- Broadcom BCM2837 quad core Cortex A53 processor @ 1.2 GHz with Videocore IV GPU;
- 1GB LPDDR2;
- 4GB eMMC flash;
- 48x GPIO;
- 2 I2C, SPI and UART;

- 2.5V to 5V power supply;
- -25°C to +80°C temperature range.

3.4 Software Specification

After presenting all the hardware that will be used in the development of the project, this sections tends to specify the software. Some other software tools crucial to the understanding of this Dissertation were briefly explained in the State of the Art chapter. This sections aspire to analyze the software platforms that will be used for the further implementations.

3.4.1 Qt

Qt is a cross-platform application development framework developed by the Qt Company for desktop, embedded and mobile devices. It is also a free and open-source widget toolkit for creating graphical user interfaces. It is important to note that Qt is not a programming language but rather it is a framework that extends the C++ language with the usage of a preprocessor - MOC (Meta-Object Compiler). Before stepping into compilation, the source files written in the C++ extended are parsed with MOC to generate C++ compliant sources. Following this method, Qt is able to add important features to the programming languages that will be used to develop this project. Despite the fact that any build system can be used, Qt brings its own qmake - a cross-platform front-end platform build system like GNU (GNU's Not Unix) Make, Visual Studio and Xcode.

Qt Creator is the cross-platform integrated development environment which is part of the SDK for QT and the version that was used is 5.12.3. It includes:

- an intelligent code completion;
- syntax highlighting;
- an integrated help system;
- a debugger;
- an integrated GUI layout and forms designer.

The reasons for opting by this software platform to develop the project for this Dissertation are: it is a very well known platform that was used throughout the course; its extremely intuitive module for programming GUI; and the cross-platform development that allows to run the code on most of embedded platforms. Also, the C++ expanded framework allows the usage of some outstanding features like signals and slots. This mechanism is used for communication between objects that is crucial for GUI development but that was also used to send notifications between different processes started by another Qt feature - QProcess.

3.4.2 Docker

Docker is an open-source project that automates the deployment of software applications inside containers by providing an additional layer of abstraction and automation of OS-level virtualization on Linux. It provides tools for simplifying the development process by creating lightweight virtual machines called containers. The containers provide an image-based deployment model that makes it very simple to share an application or a set of services with all of their dependencies across multiple environments. This is possible for the reason that Docker technology uses the Linux kernel and its features (E.g Cgroups, namespaces, etc.) to segregate processes making it possible to run them independently [16].

Although sometimes the differences between Containers and Virtual Machines are not completely clear, these two processes differ in many ways as it can be observed on the figure 3.3.

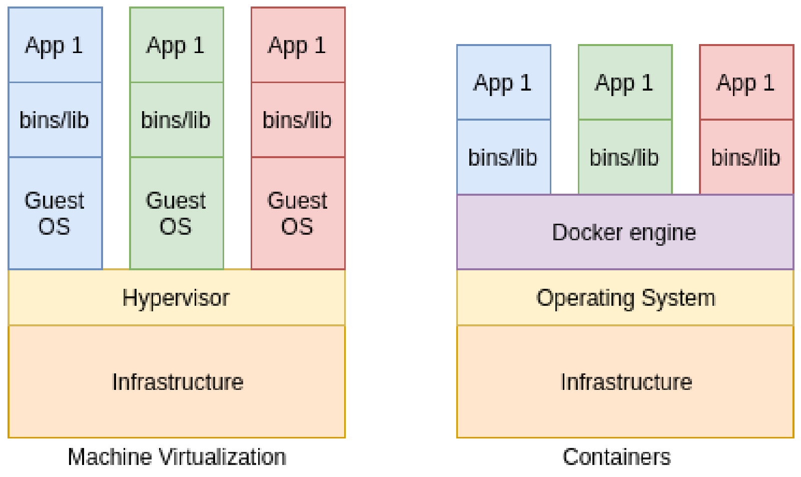


Figure 3.3: Visualization of differences between VM (Virtual Machine) and Docker. Adapted from [16]

As it can be observed on the previous figure, Docker differ from a Virtual machine in several ways, but the primary difference is that with the usage of containers, an OS is virtualized in a way that multiple workloads are able to run upon a single OS instance. VMs, on the other hand, virtualize the hardware to run multiple OS instances.

The Docker architecture is composed mainly of these elements:

- **Image:** a file used to execute code in a Docker container and each image contains the software that is desired to run;
- **Container:** a running instance of an image using the Docker run command, including everything needed to run an application: code, runtime, system tools, system libraries and settings;
- **Dockerfile:** a text file of Docker instructions used to assemble Docker's image.

To conclude, with the usage of Docker the constraint of Portability is granted, allowing the final project to be run on any environment. Since the Docker was never a topic in any subject during the course, the learning curve is also important due to its increasing importance in the real world.

Chapter 4

Implementation

After defining the system requirements and constraints and representing the hardware and software components, it was possible to proceed to the implementation of a system that fulfills the objectives of this Dissertation.

For matters of contextualization, the project was developed using Qt 5.12.2 (x86_64-little_endian-lp64) that run upon an Ubuntu 18.04.2 LTS. The Buildroot version's that was used as a back end is Buildroot-2019.02.1. For purposes of simplification, the implementation was divided into tree different sections.

The first section goal was to create a minimal initial configuration for the OS image by writing configuration files for kernel, busybox and OS. These files included the minimal configurations for the system to run, that were previously studied by exploring Buildroot and Kernel environment.

The second section made sure all the requirements to install Buildroot and all the necessary packages were met and all the needed changes in the system were made.

In the third section, the OS was customized by the user in an perceptive way, creating the final image with the selected features upon a minimal default configuration.

All of the steps of the installation and customization processes are saved into a log file in a *status* variable. This way, when the system powers in, it knows where it exactly stopped.

4.1 Minimal configuration

The minimal OS image was a final result of the make build of the three configuration files. The *minimal_defconfig* that had the basic features and the enabling of the external toolchain as well as the path for kernel and busybox configuration files that were previously written for purposes of a minimalist configuration.

The *minimal_defconfig* file specifies the kind of processor of the architecture that will run the generated OS; gives the path for the previously created folders; enables and downloads an external toolchain; gives the path for post-build and post-image scripts; specifies the kernel custom config file as well busybox's; and the final maximum size of the root filesystem and its formats are defined. All the following customization will be written upon this file.

4.2 Initial configuration of the System

When the system first starts, it needs to make sure that all the necessary software packages and dependencies are installed in order to proceed with the OS image customization. In case the system was interrupted or canceled for some reason, the status is saved into a log file. Next time the system starts, it knows were it previously stopped so the installation may continue instead of restarting. When the system boots for the first time, the status variable is always 0.

The program that is started by *QProcess* is a shell script that runs all the necessary commands. The script has different purposes depending on the arguments it receives.

The very first action is to ask the user for root permission with the usage of *QInputDialog* that asks for the password and sends it as a parameter to the initialization script. The password is asked through a GUI QDialog like it is observed in the figure 4.1.

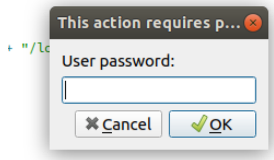


Figure 4.1: GUI QDialog window created with Qt Design

```
sh_params<<param1<<sh_param2;
check_pw->setWorkingDirectory(QDir::currentPath());
check_pw->start(QCoreApplication::applicationDirPath()+"/initialization.sh",
    sh_params);

if(!check_pw->waitForStarted()){
QMessageBox::information(this, "error","Error openning script");
qDebug()<<"error openning script";
}
check_pw->waitForFinished(-1);
```

Listing 4.1: QProcess for root recognition.

The initialization script is run by *QProcess* sending a *QStringlist sh_params* with two parameters:

- *param1* that indicates the required action (checking the password)
- *sh_param2* containing the root password that is going to be used.

If the terminal output shows an incorrect password, the process is restarted from the *QInputDialog* and the parameters are cleared. The code for this action can be analyzed in 4.1.

The script then runs the command for the installation of Buildroot and its dependencies with sudo having the password as an argument like it is observed in the following code listing 4.2

```
echo $2 | sudo -S -k apt-get install
```

Listing 4.2: Passing the password as an argument to a shell script.

When the process stops, the system invokes the *ReadStdError()*. If the output has an error, the password is wrong and the user is once again asked to insert it. In contrast, if no error has occurred, the password is wiped and the following steps start, informing the user of its state through a GUI.

In case of a correct password, the script will run a set of commands that represent the steps of this process. When all the proceedings are done, the user should have on his working environment: all the required tools for running buildroot and cross compile; a recent version of Buildroot; a configured external toolchain with a customized output path and minimal configuration files. After each of the commands is executed, the script updates the status in the log file.

- **Step 1:** Installing required packages

In order to run Buildroot, the installation of the tools listen in the 4.3 is mandatory.

```
gzip tar wget bash binutils file python3 bc libcurl4-gnutls-dev
   libexpat1-dev gettext libz-dev libssl-dev git g++ minicom
```

Listing 4.3: Installing pre requisites for Buildroot.

The status is then updated to 1 in the log file, permitting the user to grant this as the new start position in case any error occurs.

- **Step 2:** Downloading Buildroot

  Once all the requirements are installed, the Buildroot can be downloaded from its official website using *wget* command. The current up to date version was installed for this project - buildroot-2019.05.1.tar.gz and the internet address for the download can be seen in the listing 4.4.

```
wget "https://buildroot.org/downloads/buildroot-2019.05.1.tar.gz"
```

Listing 4.4: Downloading the most recent version of Buildroot.

- **Step 3:** Unpacking Buildroot The download zipped file must then be unpacked by previously installed either *gzip* or *tar* tool. After executing this command, all Buildroot's folders will be created with the necessary files for using it to create standard embedded Operating Systems. The extraction process is listed in 4.5.

```
tar -xzvf buildroot-2019.05.1.tar.gz
```

Listing 4.5: Unpacking the Buildroot.

- **Step 4:** Creating necessary folders: In order to have a better control and to enable the custom configuration of the OS images, it is necessary to create additional folders for external build and downloads and set the path for the next step to be executed properly. The commands for this operation are listed in 4.6.

```
mkdir build
mkdir buildroot_dl
cd build
```

Listing 4.6: Creating additional folders for external processes.

The *status* variable is then incremented.

- **Step 5:** Configurations of external files The minimalist configuration files previously defined after a deep study and analysis of Buildroot and Kernel's features must be included in the process of creating an OS image with Buildroot. This is done by running *BR2_EXTERNAL* command in the terminal, setting the directory of the external build files and specifying the output directory. The minimal configuration file, that is responsible for defining all the custom files and packages is added to the standard */configs* list for default configurations for different platforms. All is executed in one single command and the *status* variable is actualized. The shell commands for defining an external tree in Buildroot can be observed in the listing 4.7.

```
 BR2_EXTERNAL=~/Custom_OS_BR/Custom_OS_BR/minimal_raspberrypi_
droot-master/
WD -C ~/Custom_OS_W_BR/Custom_OS_BR/mnimal_raspberrypi_
droot-master/buildroot-2019.05.1/
mal_deconfig
```

Listing 4.7: Setting an external build path to Buildroot.

To let the user aware of the steps that are being executed, a QT Gui was created, having a progress bar that is incremented with the mechanism that checks for the status that is altered after every step of the initialization of the system. When the last command of the script is executed, the *status* is set to 100, killing the process that runs the script and closing this graphic window and starting a new one, that will step into the customization of the final embedded Operating System image. All the necessary installation are showed to user via GUI like it can be observed in the figure 4.2.

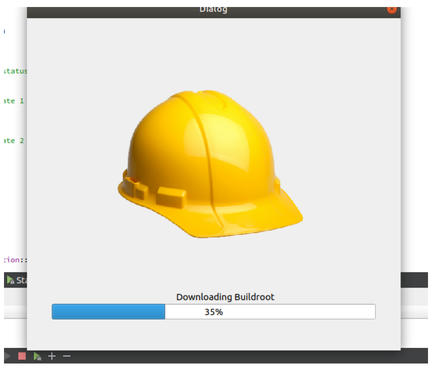


Figure 4.2: GUI for the Installation / Configuration of Buildroot and its dependencies

This section overall workflow can be observed in the figure 4.3.

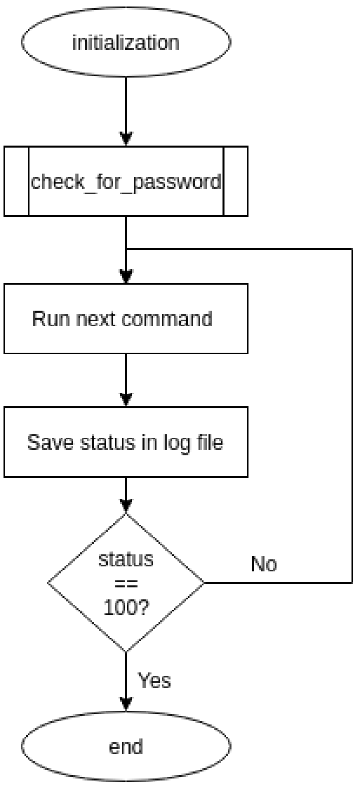


Figure 4.3: System initialization flowchart

After obtaining root permission, the shell script is run through a *QProcess* that is invoked through the *signals and slots* feature available in kernel.The signal is sent with the usage of *connect* command and the receiving slot is meant to read the terminal output every time it is updated.

```
connect(shell_script,
SIGNAL(readyReadStandardOutput()),
this,
SLOT(sh_output()));


connect(shell_script,
SIGNAL(readyReadStandardError()),
this,
SLOT(sh_output()))
```

Listing 4.8: Sending script's terminal output through SIGNALS

The *check_for_password's* workflow also must be specified as it is a crucial part of the system. The majority of the commands written in the shell script must be run as root in order for further customization and OS image creation. The workflow of this process can be visualized in the figure 4.4.

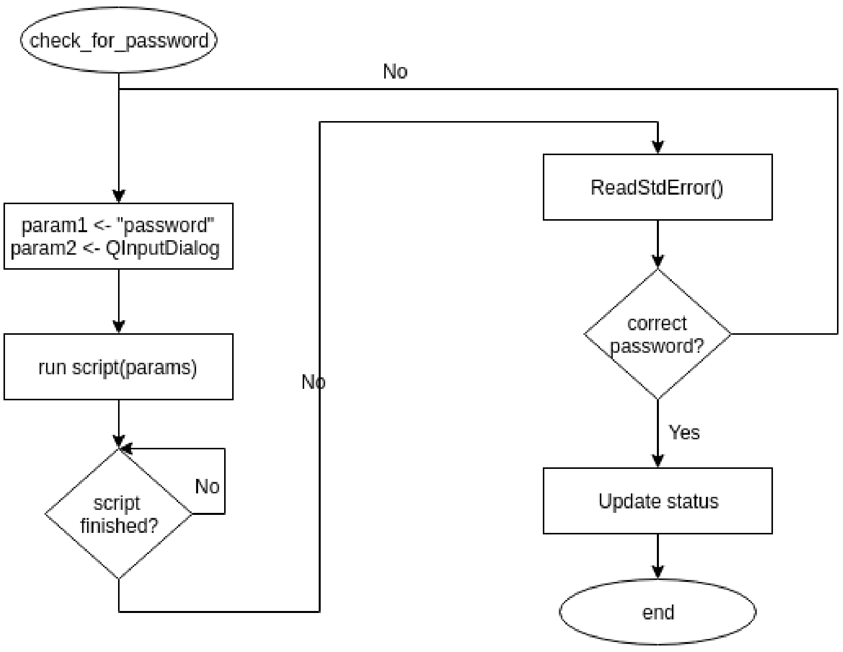


Figure 4.4: Check root permission flowchart

When all the preceding steps of this section are concluded, the working directory where the application resides should be ready for the user to take over and start the customization. All the packages should be installed and all the dependencies managed. The main folder will be the installed Buildroot which will act as a back-end, having all the necessary files for the image build and deployment. The external configuration files and scripts that were written must also be specified and linked to the main system in order to provide the user the necessary environment to tailor the final OS image specific to his needs. The working directory of the application will have an structure like the one represented in the figure 4.5.

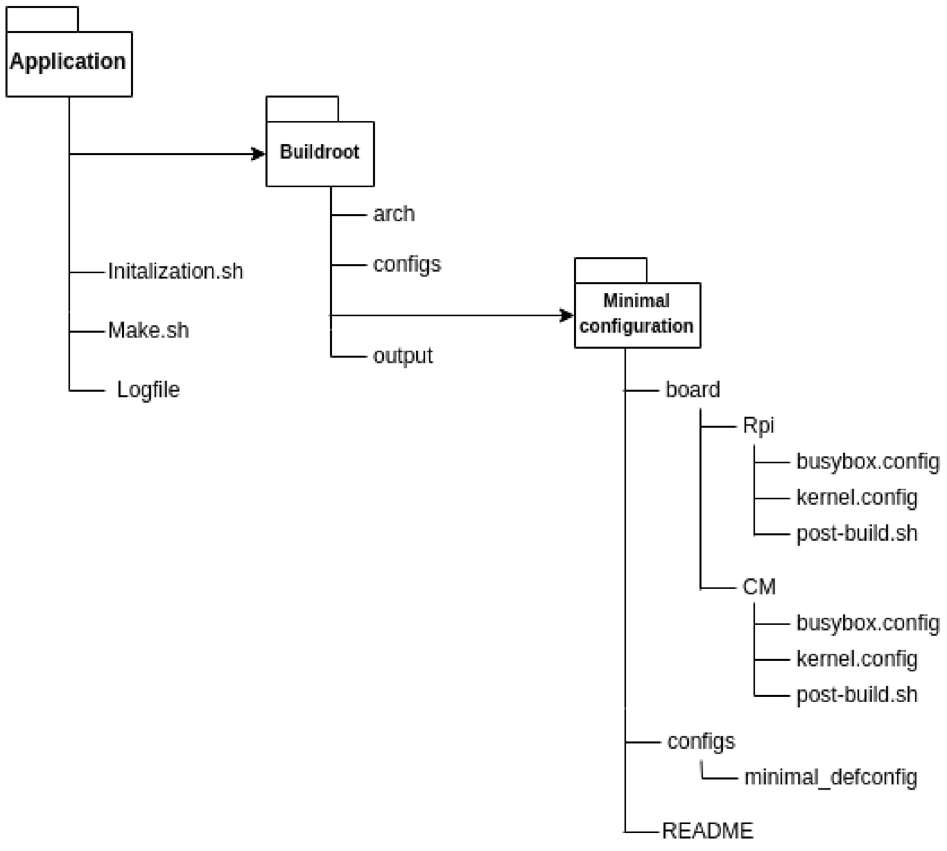


Figure 4.5: Application's working directory structure

4.3 Customization

After the last command of the shell script is run successfully, the status is updated to 100 meaning that the initialization and preparation of the system is ready. This closes the first GUI developed by Qt Design. A second graphic window then opens, allowing to activate some additional features in the final Operating System using *QCheckBoxes*.

The GUI presented to the user divides itself it different elements:

In the upper part of the window it is possible to choose the target architecture to run the generated OS. The scope of this Dissertation is to create OS images for Raspberry Pi and the Compute Module (that may run with a custom board that was created in our university's laboratory). Currently, only these platforms will be presented.

In the center part of the window, the additional features may be chosen. Each selected checkbox will add the respective package to the OS that will be generated in the end. The mechanism for adding the features to the final image is similar for every checkbox, having some exceptions.

The lower part permits to choose the desired path for the output that will contain the generated image iso file, compressed root filesystem, the bootloader and other components crucial to the OS functioning. In this part, the user can also start the *make* process that will build all the files in the chosen directory. This mechanisms are implemented using *QtPushButtons*.

All the customization and additional information is presented to the user through a text browser that is also embedded in the window. The interaction window where all the customization is chosen can be visualized in the figure 4.6.

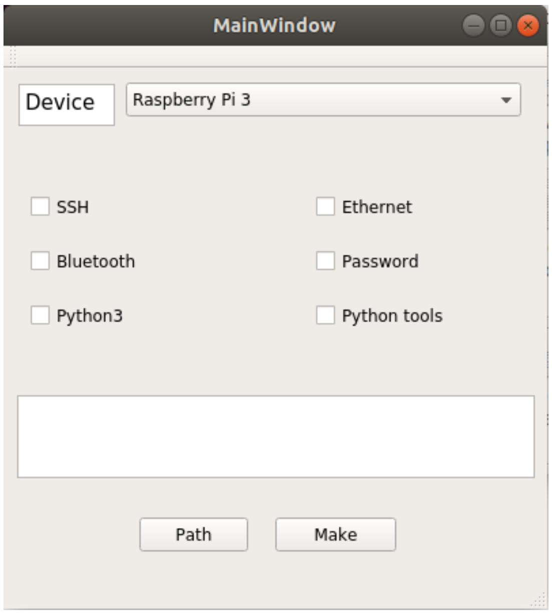


Figure 4.6: Customization GUI

Each of the features that is presented as checkbox can be added to the OS by enabling the respective package in the defconfig file that was previously created. This way, every check box has a configuration line assigned that is written to the file upon selection.

When selecting **SSH** the bit for the configuration line must activated. The bit is enabling the *BR2_PACKAGE_DROPBEAR=y* in the defconfig file when the *make* button is clicked. The same process is applied for the others checkboxes, except the **Password**. If the user opts for having a root password in his system, when selecting the box the password must be immediately provided through the *BR2_TARGET_GENERIC_ROOT_PASSWD="rootpassword"* configuration line. The password choice appears as a pop up window as it can be observed in the figure 4.7.

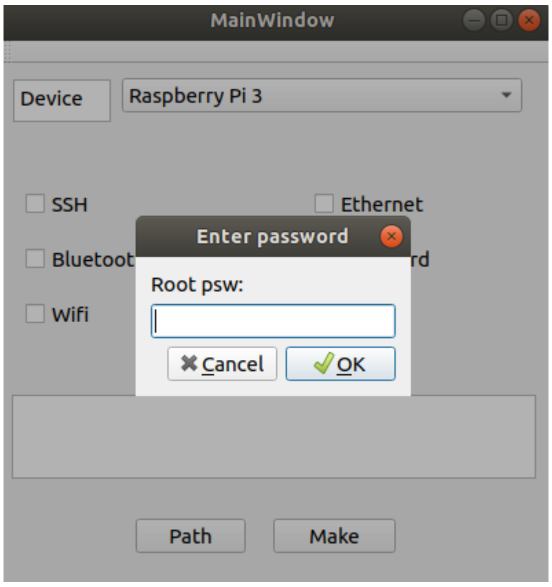


Figure 4.7: Customization GUI asking to set a root password for the Embedded Operating System

As it was explained, every checkbox feature has a bit associated that as activated upon selection. The bits are assigned to the *QStrings* that contain the configuration lines that will be added.

When *make* is clicked, it triggers the *config()* function that receives the configuration line as an argument, in case that bit was set (meaning the checkbox was selected). This function opens the defconfig file and adds all the configuration lines that were previously selected. The minimal default configuration file is now altered tailoring the OS image according to user's needs.

The next step of the customization section is to start a QProcess that will run the script with a different argument notifying that the system is ready to be built. The first command to be run is *make minimal_defconfig* that builds the options that were just altered in the default configuration file. After setting the options, the command *make* is run that creates the entire system in the *output/* directory, installing all the selected target packages, building a kernel and a bootloader as well as the root filesystem.

The whole implementation of this section can be observed in the figure 4.8 that follows.

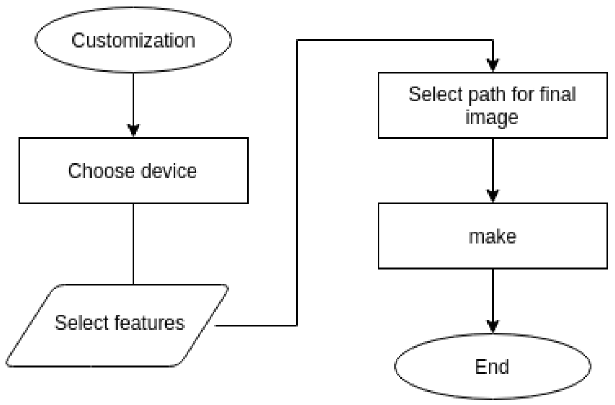


Figure 4.8: Customization flowchart

4.4 Integration with Docker

Before proceeding to the first step of integration with Docker - creation of a Dockerfile - it is necessary to set the right environment for the Qt project and upload it to a github repository in order to make the integration easier. Since all the Docker procedures are run from the Terminal, it is necessary to compile the Qt project from the terminal also, using *qmake*. After deleting all the *.pro* and *Makefile* from the folder, it is necessary to run the command listed in 4.9.

```
$ qmake -project
```

Listing 4.9: Creating a Qt Project File.

This command creates a Qt project file. After this step is completed, a little configuration to the *.pro* must be made, adding all the necessary components. The next step is to make the project platform specific and create a *Makefile* by running the following command listed in 4.10.

```
$ qmake ProjectName.pro
```

Listing 4.10: Making a Qt Project platform-specific.

In this way, the project environment is set for Docker as only the last step that creates the executable for the application is missing, which will be completed later in the Dockerfile. Now it is possible to create a github repository with the project.

In order to start the integration of the final project with Docker, it is trivial to proceed to the creation of a Dockerfile - a script that contains the collection of the commands necessary to make the project work.

4.4.1 Dockerfile

The first step in order to create a Dockerfile is to find the right base image that must be used for building a new image of our project. This image is introduced by the *FROM* command, at the top of the Dockerfile. This instruction initializes a new build stage and sets the base image for subsequent instructions, specifying the underlying OS architecture that is going to be used afterwards. Since this dissertation's project was formed using Ubuntu, that would be the base image to use. Many Ubuntu images can be found in the Docker official repository. The Dockerfile starts with the following instruction listed in 4.11.

```
    FROM ubuntu:latest
```

Listing 4.11: Dockerfile's base image.

Once the base image is chosen, it is necessary to create the ideal environment for the program to run. This is done by ensuring that all the required components are installed. The installation of the requirements can now be made using commands from Ubuntu, since it was the chosen base image. Terminal commands must be preceded by the Docker instruction *RUN* and followed by *-y* to automate the answer "yes" to any emerged questions. Buildroot requirements, Qt components and other software need to be installed using the command listed in 4.12.

```
RUN apt-get install qt5-qmake -y; apt-get build-essentials -y;
```

Listing 4.12: Dockerfile: installation of the requirements example.

After setting up the environment for the Qt project, it can be cloned from a github repository recurring once again to the *RUN* instruction. The last step of building the Qt project can now be made inside the Docker in order to create the application's executable to be later run. The next step is to give root permissions to the project files and scripts using Ubuntu's instruction *chmod*.

The complete Dockerfile is listed in 4.13.

```
FROM ubuntu:latest
RUN apt-get update -y;

RUN apt-get install locales -y; apt-get install sudo -y; apt-get install
    cmake -y; apt-get install devscripts -y; apt-get install qt5-default -y;
    apt-get install wget -y; apt-get install git-core -y; apt-get install
    unzip -y; apt-get install texinfo -y; apt-get install gcc-multilib -y;
    apt-get install build-essential -y; apt-get install gzip -y; apt-get
    install tar -y; apt-get install bash -y; apt-get install binutilus -y;
    apt-get install file -y; apt-get install python3 -y; apt-get install bc
    -y; apt-get install libcurl4-gnutls-dev -y; apt-get install
    libexpat1-dev -y; apt-get install gettext -y; apt-get install libz-dev
    -y; apt-get install libssl-dev -y; apt-get install git -y; apt-get
    install g++ -y; apt-get install minicom -y; apt-get install cpio -y;
    apt-get install python -y; apt-get install rsync -y;

EXPOSE 22
VOLUME ["/data"]
COPY . /data
WORKDIR /root
RUN git clone https://github.com/vladreznikov/tailorOS.git
RUN make -C tailorOS
RUN chmod 777 tailorOS/initialization.sh
RUN chmod 777 tailorOS/make.sh
RUN locale-gen en_US.UTF-8
RUN export LC_ALL=en_US.UTF-8
RUN export LANG=en_US.UTF-8
```

Listing 4.13: Final Dockerfile of the project

4.5 Conclusion

The system implementation was divided into several different sections in order to facilitate the development. The approach used in the development of this Dissertation's project was iterative, finish each section of the implementation completely in order to advance for the next one.

Firstly, it was important to create a minimal configuration file that allowed the system to boot in the specified architecture. Subsequently it was necessary to make sure that the user will have the dependent software tools and packages for this application to run correctly. To fulfill that purpose, a script that automates the installations and manipulates the directories for the creation of the appropriate environment was created. Once all the necessary modifications are done, it is possible to run the GUI that lets the user tailor the OS image and build it to the selected directory. As last step, the system was integrated with Docker in order to grant the application portability for every working environment.

Chapter 5

Tests and Results

When no additional packages are selected, it is possible to build the minimal image that is the template and all the subsequent OS will be built upon it. The final result is a 2.4 MB Kernel and a 3.7 MB Root filesystem compared to a standard 60 MB image generated by Buildroot. When adding the root password tool and activating Ethernet driver in kernel in order to be able to use the *SSH* feature, the root filesytem size increases to 8.6 MB and the boot partition has 6 MB. The image has a working HDMI and serial console as well as Ethernet and Internal Wi-Fi with DHCP (Dynamic Host Configuration Protocol).

5.1 Minimal image tests

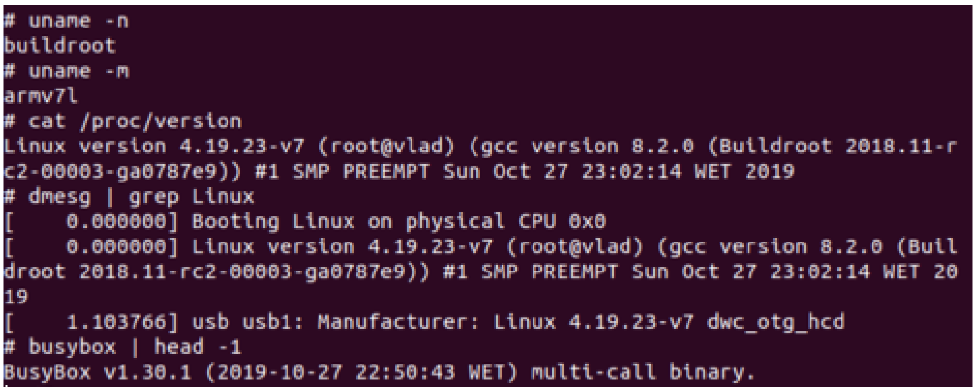
Before proceeding with development of the Dissertation, the minimal image of the OS had to be tested carefully as it would serve as base image for the further builds.

The necessary features to be added and specified in this image are:

- **Specify the processor**: since the tests will be run upon boards of Raspberry Pi family, Cortex A53 is specified.
- **Specify the custom toolchain to be downloaded**: an external toolchain is downloaded in order to step up the whole process.
- **Download the Kernel**: the latest version of kernel is downloaded.
- **Generate the image**: making sure the iso of the image is generated.

The final OS must be able to boot and connect to a screen through HDMI. Since busybox regular configuration is installed, Ubuntu basic commands can be run and the system must behave like a normal Ubuntu (without most of the features).

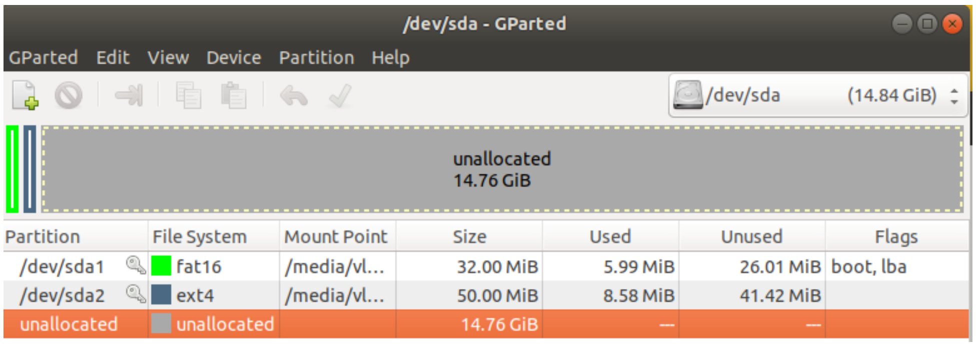
The image shows the OS source, which is Buildroot that was used as backend in order to build the image, and also the hardware architecture upon which the system is running. In addition it is possible to observe the versions of Linux, Kernel and BusyBox respectively confirming the proper functioning of the system. The root filesystem size is also displayed. This information can be observed in the figure 5.1



```
# uname -n
buildroot
# uname -m
armv7l
# cat /proc/version
Linux version 4.19.23-v7 (root@vlad) (gcc version 8.2.0 (Buildroot 2018.11-r
c2-00003-ga0787e9)) #1 SMP PREEMPT Sun Oct 27 23:02:14 WET 2019
# dmesg | grep Linux
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.19.23-v7 (root@vlad) (gcc version 8.2.0 (Buil
droot 2018.11-rc2-00003-ga0787e9)) #1 SMP PREEMPT Sun Oct 27 23:02:14 WET 20
19
[    1.103766] usb usb1: Manufacturer: Linux 4.19.23-v7 dwc_otg_hcd
# busybox | head -1
BusyBox v1.30.1 (2019-10-27 22:50:43 WET) multi-call binary.
```

Figure 5.1: Minimal image running on Raspberry Pi

The final OS size can be confirmed using *Gparted* tool, where the information about the size of the SD Card that has the OS image is shown. The final minimal sizes can be observed in 5.2.



/dev/sda - GParted

GParted Edit View Device Partition Help

/dev/sda (14.84 GiB)

unallocated
14.76 GiB

Partition	File System	Mount Point	Size	Used	Unused	Flags
/dev/sda1	🟩 fat16	/media/vl...	32.00 MiB	5.99 MiB	26.01 MiB	boot, lba
/dev/sda2	🟦 ext4	/media/vl...	50.00 MiB	8.58 MiB	41.42 MiB	
unallocated	▨ unallocated		14.76 GiB	---	---	

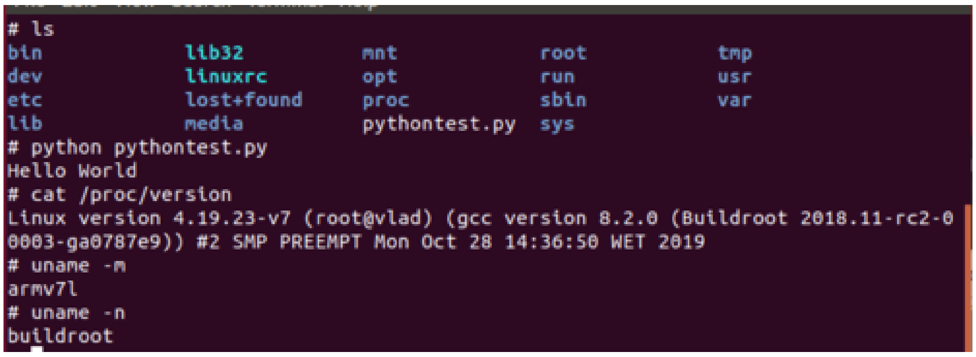
Figure 5.2: Minimal image size on GParted

5.2 Python tests

Python is a language that is rising in the world of embedded applications. Although not being the best candidate because of real time requirements that a typical embedded system demands, the day-by-day improvement of hardware allows SoCs to run embedded Linux which can easily be adapted for python scripts.

Python also has good data processing support which can be utilized when sensor data must be processed. In addition, network modules, socket programming and web server hosting capabilities of Python come handy for some applications. The online support and developers base have also arisen for boards like Raspberry Pi.

In order to create an image with working Python, it is necessary to enable *Python3*, *Setuptools* and its dependencies. The result is a bootable 20.1 MB root filesystem. In order to test the OS, a simple Python script was run using the proper command like it can be visualized in 5.3.



```
# 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
# cat /proc/version
Linux version 4.19.23-v7 (root@vlad) (gcc version 8.2.0 (Buildroot 2018.11-rc2-0
0003-ga0787e9)) #2 SMP PREEMPT Mon Oct 28 14:36:50 WET 2019
# uname -m
armv7l
# uname -n
buildroot
```

Figure 5.3: Testing a python script with generated OS

As it can be observed in the Figure 5.3, the OS running on a Raspberry Pi successfully runs a python script that has the following test command.

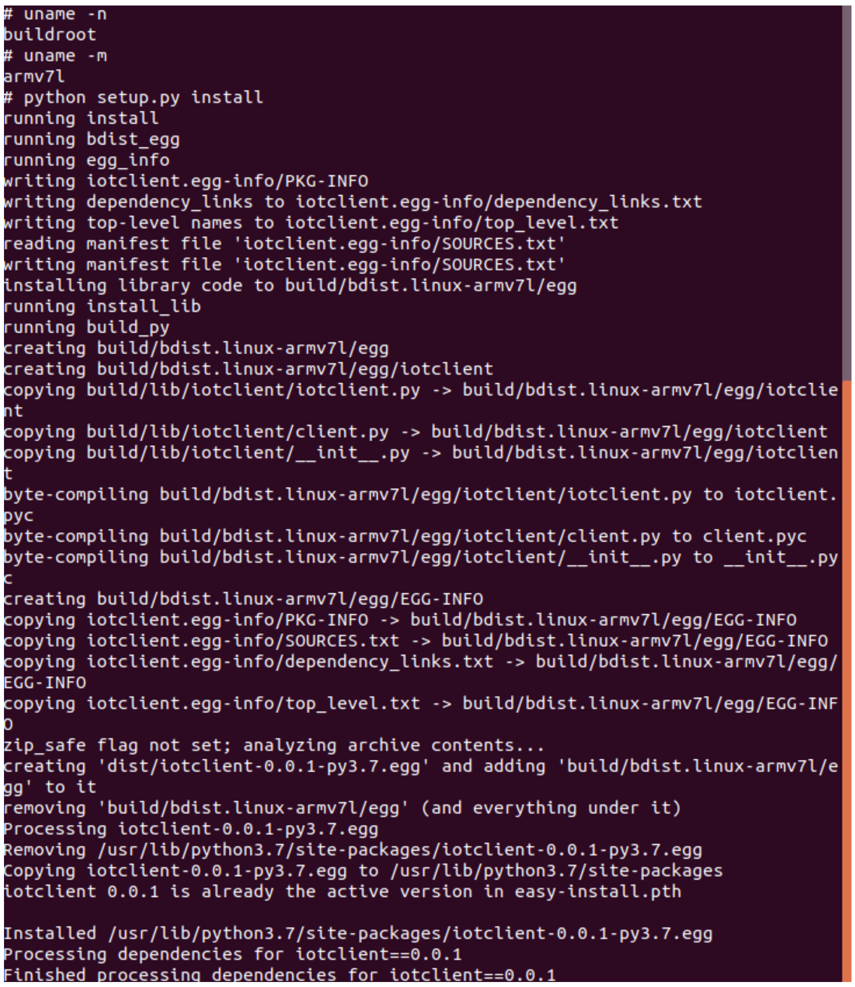
```
print("Hello World")
```

After making sure the python tools are operating in a correct way, it is possible to proceed to testing more advanced programs.

5.3 IoT tests

In order to prove the correct functioning of this Dissertation, a more sophisticated utility had to be tested using an OS generated by the developed application. Recurring to IoT projects previously elaborated in the laboratory, it was possible to make a *Python* script that creates a *Client* inside an *IoT Server* and receives packages through network.

The first step of this process is to run a *setup.py* script that installs the necessary packages in order to start the IoT server/client data exchange. This process can be observed in the following figure 5.4.

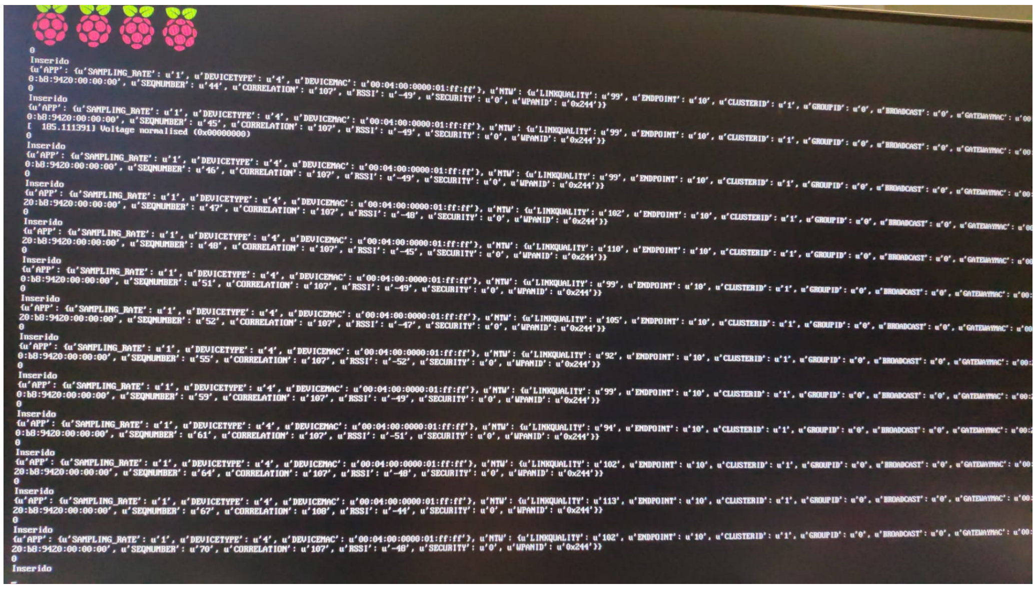


```
# uname -n
buildroot
# uname -m
armv7l
# python setup.py install
running install
running bdist_egg
running egg_info
writing iotclient.egg-info/PKG-INFO
writing dependency_links to iotclient.egg-info/dependency_links.txt
writing top-level names to iotclient.egg-info/top_level.txt
reading manifest file 'iotclient.egg-info/SOURCES.txt'
writing manifest file 'iotclient.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-armv7l/egg
running install_lib
running build_py
creating build/bdist.linux-armv7l/egg
creating build/bdist.linux-armv7l/egg/iotclient
copying build/lib/iotclient/iotclient.py -> build/bdist.linux-armv7l/egg/iotclie
nt
copying build/lib/iotclient/client.py -> build/bdist.linux-armv7l/egg/iotclient
copying build/lib/iotclient/__init__.py -> build/bdist.linux-armv7l/egg/iotclien
t
byte-compiling build/bdist.linux-armv7l/egg/iotclient/iotclient.py to iotclient.
pyc
byte-compiling build/bdist.linux-armv7l/egg/iotclient/client.py to client.pyc
byte-compiling build/bdist.linux-armv7l/egg/iotclient/__init__.py to __init__.py
c
creating build/bdist.linux-armv7l/egg/EGG-INFO
copying iotclient.egg-info/PKG-INFO -> build/bdist.linux-armv7l/egg/EGG-INFO
copying iotclient.egg-info/SOURCES.txt -> build/bdist.linux-armv7l/egg/EGG-INFO
copying iotclient.egg-info/dependency_links.txt -> build/bdist.linux-armv7l/egg/
EGG-INFO
copying iotclient.egg-info/top_level.txt -> build/bdist.linux-armv7l/egg/EGG-INF
O
zip_safe flag not set; analyzing archive contents...
creating 'dist/iotclient-0.0.1-py3.7.egg' and adding 'build/bdist.linux-armv7l/e
gg' to it
removing 'build/bdist.linux-armv7l/egg' (and everything under it)
Processing iotclient-0.0.1-py3.7.egg
Removing /usr/lib/python3.7/site-packages/iotclient-0.0.1-py3.7.egg
Copying iotclient-0.0.1-py3.7.egg to /usr/lib/python3.7/site-packages
iotclient 0.0.1 is already the active version in easy-install.pth

Installed /usr/lib/python3.7/site-packages/iotclient-0.0.1-py3.7.egg
Processing dependencies for iotclient==0.0.1
Finished processing dependencies for iotclient==0.0.1
```

Figure 5.4: Running setup.py script that creates an IoT client

After setting up the IoT server using previously commands, it is is possible to run another python script that creates a client to exchange data with the the server. The Raspberry Pi running this script becomes now part of a cluster where other similar boards are connected using the same IoT server and the exchange is done between them. The proper functioning and the data transmission can be observed in the figure 5.5.



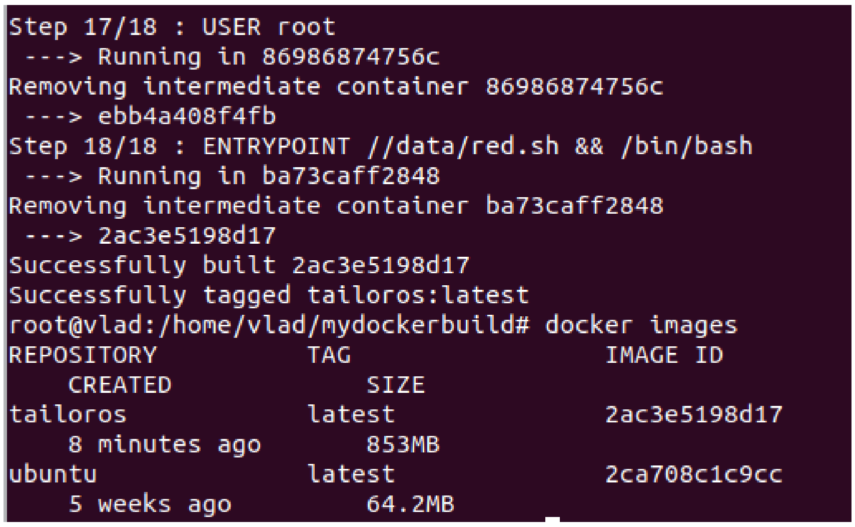
```
0
Inserido
{u'APP': {u'SAMPLING_RATE': u'1', u'DEVICETYPE': u'4', u'DEVICEMAC': u'00:04:00:0000:01:ff:ff'}, u'NTW': {u'LINKQUALITY': u'99', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00
0:b8:9420:00:00:00', u'SEQNUMBER': u'44', u'CORRELATION': u'107', u'RSSI': u'-49', 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'99', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00
0:b8:9420:00:00:00', u'SEQNUMBER': u'45', u'CORRELATION': u'107', u'RSSI': u'-49', u'SECURITY': u'0', u'WPANID': u'0x244'}}
[  185.111391] Voltage normalised (0x00000000)
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'99', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00
0:b8:9420:00:00:00', u'SEQNUMBER': u'46', u'CORRELATION': u'107', u'RSSI': u'-49', 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'102', 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'47', u'CORRELATION': u'107', u'RSSI': u'-48', 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'110', 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'48', u'CORRELATION': u'107', u'RSSI': u'-45', 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'99', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
0:b8:9420:00:00:00', u'SEQNUMBER': u'51', u'CORRELATION': u'107', u'RSSI': u'-49', 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'105', 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'52', u'CORRELATION': u'107', u'RSSI': u'-47', 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'92', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
0:b8:9420:00:00:00', u'SEQNUMBER': u'55', u'CORRELATION': u'107', u'RSSI': u'-52', 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'99', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
0:b8:9420:00:00:00', u'SEQNUMBER': u'59', u'CORRELATION': u'107', u'RSSI': u'-49', 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'94', u'ENDPOINT': u'10', u'CLUSTERID': u'1', u'GROUPID': u'0', u'BROADCAST': u'0', u'GATEWAYMAC': u'00:
0:b8:9420:00:00:00', u'SEQNUMBER': u'61', u'CORRELATION': u'107', u'RSSI': u'-51', 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'102', 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'64', u'CORRELATION': u'107', u'RSSI': u'-48', 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'67', u'CORRELATION': u'108', 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'102', 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'70', u'CORRELATION': u'107', u'RSSI': u'-48', u'SECURITY': u'0', u'WPANID': u'0x244'}}
0
Inserido
```

Figure 5.5: IoT Server/Client data exchange

5.4 Docker tests

In order to grant the portability of the final project of this Dissertation, it was necessary to create a docker image from which the container would run, executing all the necessary actions.

The first step is to build the image using the previously mentioned Dockerfile. In the following figure it is possible to observe the successful build of the last steps as well as the final image in the Repository named *tailoros*. The output from the Dockerfile can be observed on the figure 5.6.



```
Step 17/18 : USER root
 ---> Running in 86986874756c
Removing intermediate container 86986874756c
 ---> ebb4a408f4fb
Step 18/18 : ENTRYPOINT //data/red.sh && /bin/bash
 ---> Running in ba73caff2848
Removing intermediate container ba73caff2848
 ---> 2ac3e5198d17
Successfully built 2ac3e5198d17
Successfully tagged tailoros:latest
root@vlad:/home/vlad/mydockerbuild# docker images
REPOSITORY          TAG                 IMAGE ID
    CREATED             SIZE
tailoros            latest              2ac3e5198d17
    8 minutes ago       853MB
ubuntu              latest              2ca708c1c9cc
    5 weeks ago         64.2MB
```

Figure 5.6: Output from the build of the Dockerfile.

The next is to run the container from the created docker image and operate inside it. In order to recognize the host display to run the GUI, the command listed in 5.1 has to be run.

```
xhost +local:docker
```

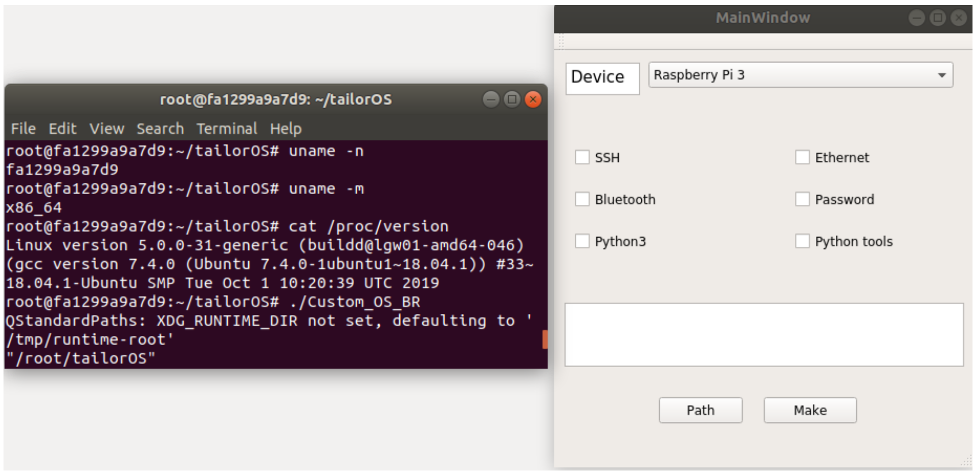
Listing 5.1: Command to run GUI on host display from a docker container.

After that, it is possible to run the container, recurring to the *run* command of docker with some extras that make the process operational. The command listed in 5.2 must be used:

```
docker run -ti --rm -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix -v
    /home:/new tailoros
```

Listing 5.2: Running a docker container.

After the previous command is executed, the system steps in into the docker container where all the changes and runs can be made. When entering the container, it posses the Qt files that are necessary to run the system that will download all the requirements as well as Buildroot and the external minimal configuration that can be found on Github repository. The only step is to run the executable inside the folder and proceed with the OS customization. This can be observed in the following figure 5.7. It can be



```
root@fa1299a9a7d9:~/tailorOS# uname -n
fa1299a9a7d9
root@fa1299a9a7d9:~/tailorOS# uname -m
x86_64
root@fa1299a9a7d9:~/tailorOS# cat /proc/version
Linux version 5.0.0-31-generic (buildd@lgw01-amd64-046)
(gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1~18.04.1)) #33~
18.04.1-Ubuntu SMP Tue Oct 1 10:20:39 UTC 2019
root@fa1299a9a7d9:~/tailorOS# ./Custom_OS_BR
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '
/tmp/runtime-root'
"/root/tailorOS"
```

Device: Raspberry Pi 3

Figure 5.7: Running the project inside of a Docker container

noticed that the command line is operating inside the tag of the previously built docker image. The GUI is started from this directory and all the process is executed correctly inside the docker container.

All the code necessary to run the application can be found on Github:

https://github.com/vladreznikov/tailorOS

As well as the minimal configuration necessary to run the application:

https://github.com/vladreznikov/minimal_raspberrypi_config

Chapter 6

Conclusions and Future Work

After the whole implementation and tests, conclusions can be made from the developed work. This chapter aims to make an overview of the results obtained taking into account what is being developed in this area nowadays and the problems the embedded developers face. It is also a goal of this chapter to suggest further improvements that can be made in future implementations.

6.1 Conclusions

The conclusions are made based on all the tests that were made and all the objectives that were imposed in the Introduction chapter, creating an application that allows developers to generate low-sized embedded OS tailored to their needs without having a deep knowledge of the system and saving precious time.

As it was previously explained in this Dissertation, an embedded system is always integrated in a bigger system, usually having only one function that has strict constraints and must be executed flawlessly. Hence, the OS that is run on an embedded device normally doesn't have the need to hold most of the features as they aren't used during the functioning of the system. As follows, the created OS can be even smaller in size containing only necessary tools for the system with a specific function to run. Many systems with different purposes were tested to confirm that it is possible to run an application with the same aim saving a great a mount of precious space. An IoT client/server application using python scripts was successfully run on a system generated by this Dissertation's application proving its functionality. The IoT application, previously executed on a Raspbian XXXXX MB is now auspiciously running in a XXX MB system using only the needed features and packages

Moreover, this Dissertation overcomes another problem in generating an OS tailored to developer's needs - it doesn't have a steep learning curve as everything is automated and simple for the user to hold. In contrast with simply using Buildroot, the developer doesn't have to worry about knowing the right name of the package to enable in order to make a certain feature work or being concerned about the dependencies that each package has. Everything is managed by the final application of this Dissertation and the only job the user has in this process is selecting the desired platform and the checkboxes with the needed features for the system to operate in a proper way.

6.2 Future Work

This Dissertation created an application for developers to generate OS tailored to their needs. The architectures used for implementation and tests were the most common Raspberry Pi's family boards that are well known across the Embedded Systems course and used by most of the students.

Although the steps of tailoring and generating an OS are automated for these boards, some major modifications in the project structure have to be made in order to do the same process for other architectures. One of the future work improvements would be to extend the amount of compatible architectures.

By now, the GUI of the application allows the user to choose from a limited list of features that were considered the ones to have the most common uses across the embedded systems world. Most of the embedded applications rely on these characteristics, but other more specific projects may as well require other functions. In order to improve this topic, a future work improvement could list all the packages from the Buildroot list and let the user choose by searching a name instead of having a limited number of options as checkboxes. By the reason that the names of the packages aren't often instinctive, other improvement could be linking the desired features with the necessary packages and carrying the necessary dependencies.

Finally, the last improvement would be to suggest which features and packages are necessary for the user to choose based on a slight description of the project to be developed.

References

[1] Dimitrios Serpanos and Tilman Wolf. *Architecture of Network Systems*. 2011.

[2] *Global Embedded Systems Market Growth 2019-2024*. LP INFORMATION INC, Feb. 13, 2019.

[3] Karim Yaghmour and Jon Masters. *Building Embedded Linux Systems, Second Edition*. 2008.

[4] "Difference Between Monolithic Kernel and Microkernel". In: *Tech Differences* (June 24, 2019).

[5] Justin Garrison. *What is the Linux Kernel and What Does It Do?* July 12, 2017. URL: `https://www.howtogeek.com/howto/31632/what-is-the-linux-kernel-and-what-does-it-do/`.

[6] LWN. *Tools and distributions for embedded Linux development*. Apr. 27, 2010. URL: `https://lwn.net/Articles/384713/`.

[7] Thomas Petazzoni. *Getting Started with Buildroot*. 2018.

[8] Magnus Unemyr. *Why every Cortex-M developer should consider using a bootloader*. Oct. 28, 2016. URL: `http://blog.atollic.com/why-every-cortex-m-developer-should-consider-using-a-bootloader`.

[9] Greg Kroah-Hartman. "The Kernel Configuration and Build Process". In: *Linux Journal* ().

[10] Chad Z. Hower. "Introducing Cosmos". In: *CodeProject* (2010).

[11] Gerard Beekmans. *Linux From Scratch*. URL: `http://wiki.linuxfromscratch.org/lfs/`.

[12] Gerard Beekmans. *Linux From Scratch*. 2000.

[13] Johnatan McCrohan. "An Embedded Linux Based Remote Control System". B.A.I Engineering. Trinity College Dublin, 2011.

[14] Alexandru Vaduva. *Learning Embedded Linux Using the Yocto Project*. 2015.

[15] Manolis Agkopian. *Design Your Own Raspberry Pi Compute Module PCB*. 2019. URL: `https://www.instructables.com/id/Design-Your-Own-Raspberry-Pi-Compute-Module-PCB/`.

[16] Mike Raab. "Intro to Docker containers". Mar. 2018.

[17] Dedoimedo. *Remastersys - Create custom Ubuntu (live) CD - Tutorial*. 2008. URL: https://www.dedoimedo.com/computers/remastersys.html.

[18] Make Tech Easier. *How To Backup Your Ubuntu System With Remastersys*. 2008. URL: https://www.maketecheasier.com/backup-ubuntu-with-remastersys/.

[19] Michael Reed. "Customize a Distro with Remastersys". In: *Linux Journal* (2011).

[20] Bruce Byfield. "Custom Linux Installations". In: *Linux Magazine* (year).

[21] Aseem Kishore. *Make a Custom Live Linux Distro with Linux Live Kit*. 2019. URL: https://helpdeskgeek.com/linux-tips/make-a-custom-live-linux-distro-with-linux-live-kit/.

[22] Make Tech Easier. *Create Your Own Linux Distro with Ubuntu Imager*. 2015. URL: https://www.maketecheasier.com/create-linux-distro/.

[23] Ladislav Bodnar. *Learnig with Linux From Scratch*. 2004. URL: https://lwn.net/Articles/85865/.

[24] Debian. *Live Systems Manual*. URL: https://live-team.pages.debian.net/live-manual/html/live-manual/index.en.html.

[25] Matt Kraai. *Debootstrap*. 2000. URL: https://live-team.pages.debian.net/live-manual/html/live-manual/index.en.html.

[26] Khalid Baheyeldin. "OpenWRT". In: *2Bits* (2014). URL: https://openwrt.org/start.

[27] Mike Jaret-Schachter, ed. *OpenWRT for embedded development*. Ohio LinuxFest 2017. 2017.

[28] Marcus Schäfer, ed. *openSUSE-KIWI Imaga System*. 2016.

[29] Drew Mosoley. "Why the Yocto Project for my IoT Project?" In: *Embedded* (2017).

[30] George Kroah-Hartman. *Linux Kernel in a Nutshell*.

[31] *Demystifiying the Linux Kernel*. May 11, 2015. URL: https://blog.digilentinc.com/demystifiying-the-linux-kernel/.