POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)
SPECJALNOŚĆ: Embedded Robotics (AER)

PRACA DYPLOMOWA MAGISTERSKA

Embedded Linux
build systems

Systemy implementacji
wbudowanego Linuxa

AUTOR:
Cezary Dynak

PROWADZĄCY PRACĘ:
dr Witold Paluszyński

OCENA PRACY:

WROCŁAW 2018

To my wife and son

Contents

Chapter 1

Introduction

The main task of this work is to describe and compare tools, that help in the process of creating and deploying Linux-based operating system on embedded devices. There are many publications, that prioritize one build system or are focused on low-level problems common for all, but there was a lack of treating them as one type of software. I'm trying to fill this gap and encourage everybody who is working with embedded Linux, to give a try for all presented tools.

It's also worth to emphasize, that this topic is not limited to hobby projects and academic research, but it's essential for commercial and industrial applications. Since nowadays micro-controllers are much more powerful and enormously cheaper than yesterday's supercomputers, the main focus has moved from resources optimization to convenient maintaining. Growing costs of employment are also forcing companies to shorten time to market with the use of existing software components, especially free and open source ones. The overwhelming use of Linux in modern embedded devices, instead of traditional bare-metal programming, is shown on the Figure 1.1.

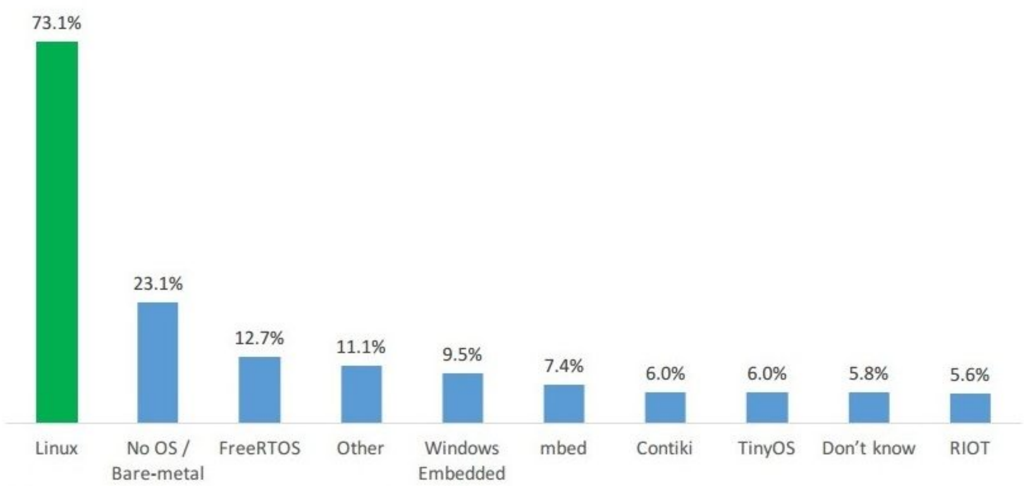

Figure 1.1: Survey results for Operating Systems used for IoT Devices[3]

1.1 Basic definitions

Despite that every aspect of computing could be described by numbers and mathematical operations, there could be plenty of definitions for one and the same thing. Unfortunately the naming conventions are sometimes misleading, as it is in the area which I am describing. People understand the word “Linux” differently in different contexts, as there is also the never ending debate about the term “GNU/Linux”.[28] Sometimes buzzwords like “cloud” and “IoT” are strongly promoted due to commercial issues, which makes common understanding even more unclear. Therefore, I would like to explain how to interpret the key wording.

Operating system is a software that manages hardware resources and provides interface to run other programs.[9]

Operating system kernel is a core part of operating system, that provides basic low level interfaces.[25]

Linux is an open source operating system kernel, that was inspired by the Unix tradition. The initial release took place in 1991 and since then it is being developed by thousands of voluntary collaborators as well as major IT companies. Formerly it was compatible only with the x86 standard processors for personal computers, but now it is available for other architectures, especially ARM that is used by most mobile and embedded devices. The work of its creator Linux Torvalds and lead maintainer Greg Kroah-Hartman is currently sponsored by the Linux Foundation. As for 2018, all of the world fastest supercomputers from TOP 500 ranking are using this kernel.

Linux distribution is the operating system, that is built upon the Linux kernel.

Embedded device is a micro-controller based device, that has a fixed purpose and strictly limited user interface.

Cross-compiler is a compiler that creates executable code for system architecture that is different from its own.

Embedded Linux build system is a set of software development tools, that create Linux distribution with the use of cross-compiler and produce complete operating system image, to be deployed on an embedded device.

The umbrella them “Embedded Linux build system” is not widely in use, but despite the similarities between described tools, no other was proposed. Different projects identify themselves as “distribution creators” or “executable documentation”, but they share the same goal and they need to be compared in one place.

1.2 Sources of knowledge

There are two important issues, that I faced while trying to make literature review: firstly what kind of materials should I analyze and secondly how to extend my searches, to get a full overview of current state of knowledge on this topic.

The first issue was very clear in the previous (20th) century - making literature review was mostly limited to analyzing two kind of printed materials: longer ones which were just scientific books and shorter ones which were peer reviewed articles in technical journals. Right now they are also available and easy accessible in electronic form and still they are the most reliable sources for literature review, but not every IT engineer is using it. Open Source Software and Open Source Hardware movement, beside creating lot of tools, also created a lot of written knowledge, but they are distributed in very different ways: project documentation, presentations, tutorials, source code comments, READMEs, How Tos, FAQs, wikis, finally blog and forum posts, but also in other forms. Because they are all available via Internet, there exists one unified way to reference them: URL (Unified Resource Locator) also known as web address or link. Because of that, this and most of other bibliographies are filled with URLs, not ISBNs.

The second thing is how to be sure that I covered every aspect, at least by mentioning. Search results are personalized, both by behavior of searcher, that is choosing keywords and also Search Engine Optimization. Nobody could tell, that this is complete, but sources are cross referenced, so after some research, the loop closes. Most important factor for choosing materials is their universality and chance that they will be not so fast outdated.

elinux.org

The Embedded Linux Developer wiki elinux.org is undeniably the most extensive source of knowledge about all aspects of Embedded Linux. In this form it was possible to consolidate a powerful community which extend its contents. From one side it is greatly filled with a lot of technical details and still extended, but from the other the materials are not always universal and some pages are not maintained.

The elinux.org domain was registered on 1999-11-04 [33] and used by the Linux specialist Tim Riker just as a placeholder.[32][30] About 2003 the Embedded Linux wiki was initiated here using the MoinMoin framework [31]. In 2007 it was moved to the Media Wiki engine, that is still in use. On the top level, its contents are divided into two main parts: “Development Portals” about various aspects of embedded Linux and “Hardware Pages” for different development boards. The most relevant page for this thesis is `http://elinux.org/Build_Systems`, but build systems are only listed there without any comparison.

Currently it is maintained by the Core Embedded Linux Project which belongs to Linux foundation [29]. CELP is also the coordinator of other important Embedded Linux activities.

https://en.wikipedia.org/wiki/Category:Embedded_Linux	
https://en.wikipedia.org/wiki/Category:Software_related_to_embedded_Linux	
https://en.wikipedia.org/wiki/Category:Embedded_Linux_distributions	
https://en.wikipedia.org/wiki/Linux_on_embedded_systems	
https://en.wikipedia.org/wiki/List_of_build_automation_software	
https://en.wikipedia.org/wiki/Cross_compiler	
https://en.wikipedia.org/wiki/Microprocessor_development_board	
https://en.wikipedia.org/wiki/Comparison_of_single-board_computers	
https://en.wikipedia.org/wiki/Embedded_Linux_build_systems	

Tabela. 1.1: The Wikipedia articles, that I both learned from and extended

wikipedia.org

Wikipedia allows anyone to edit articles, so professionalism of every page can not be assured, but it is the largest and most popular general reference work on the Internet. The most important thing from my point of view, is that it redirects and groups abstract entities on a high level. The Table 1.1 summarizes pages that are most relevant for this work.

Marcin Bis publications

In Poland, the most extensive source of written knowledge about this topic is provided by Mr Marcin Bis. He has published two books so far: “Linux w systemach embedded” (eng. “Linux in embedded systems”) in 2011 and “Linux w systemach i.MX6 series” (eng. “Linux in i.MX 6 series systems”) in 2015. The company BIS-LINUX.COM also offers various paid workshops and consultations.

Conferences and workshops

From one side there are public events, like Embedded Linux Conferences organized by CELP, from which slides and recordings are available on elinux.org:

- `http://elinux.org/Category:ELC` - Embedded Linux Conference (America)
- `http://elinux.org/Category:ELCE` - Embedded Linux Conference Europe

Embedded Linux issues connected to build systems are also present on Linux Sessions [1] coordinated by Academic IT Association from Wroclaw University of Science & Technology.

From another side, private companies make mostly interactive workshops. The paid ones are organized i.e. by Marcin Bis [19]. There are also free of charge workshops i.e. ones organized by EBV in Poland in 2016, in which I was participating.

scholar.google.com

The phrase “embedded linux build systems” gives only 2 results [13]. First of them is the presentation “Embedded Linux system development” by Thomas Petazzoni from year 2004. It describes Open Embedded which is currently included as part of the Yocto Project. The second one is the book “Mastering embedded Linux programming” by Chris Simmonds from year 2015. It differentiates only Buildroot and Yocto Project.

The phrase “embedded linux build system” gives 24 results [14]. Most of recent works enumerates Buildroot, Yocto Project, OpenWrt without comparison between them. Materials prior to 2010 are mostly using the therm “embedded linux distribution”, rather than “embedded linux build system”.

Standalone web publications

There are some very inspiring materials, that could be found with the help of search engines like Google or DuckDuckGo and they do not fit into any previous categories. Most of them, like blog and forum posts, are focused on one specific technical problem. In this section I would like to mention one article, that both covers the entire topic and also make use of the term “Embedded Linux build systems”: `https://www.embarcados.com.br/embedded-linux-build-systems/`

Official documentation

The most valuable source of technical information is and always should be official documentation. URLs are listed in the bibliography and direct to the websites where more information about each project can be found.

1.3 OS build systems structure

Source code

The key of the Linux great success is the easy accessible source code and the free software license. The possibility to engage unlimited number of people to add new functionality and fix bugs is something closer to science than to merchandising, which is the best way for engineers to get things done. Exactly the same rule apply to creating Linux based OS’es for embedded devices. Because there is a lot of subtle differences between microcontrollers, it is essential to be able to modify or just analyze every line of code.

However, the way of obtaining the whole build system is not that obvious and even differ a lot from one, to another. The most convenient way seems to be just downloading the archival version as one file through HTTP and then decompress it on our host machine. But in some cases it would be better to get it through a version control system as it is convenient for browsing history and sharing our changes. It’s also worth mentioning that all the tools I am describing are versioned also with the use of open source software tools: mostly git, which itself also came from Linus Torvalds, and sometimes subversion. Things are getting harder when the build system is split into independent parts or we need additional meta tools to get each of them. I am clarifying it step by step in Chapter 3.

Host OS requirements

Host Operating System is the operating system where the cross-compilation is made.

Software requirements

As it seems obvious, host OS need to be Unix-like, preferably Linux-based and with rpm or deb package manager. Whenever source code is under free licence, you could run it anywhere, but it may need some manual tweaking. When running build systems for the first time, Debian or Red Hat or their derivatives, like Ubuntu or Fedora are only reasonable choices. Whenever I am mentioning Host OS here, I mean Debian GNU/Linux, because that's the one, that I am most familiar with and to be precise it is stable version 9 (code name Stretch). As a result, please notice, that I refer to all packages with their deb name as default. In most cases there are equivalents in rpm packages, sometimes with different naming convention like suffix "-devel" instead of "-dev" for so called "development packages", that contain source code.

Hardware requirements

The software requirements are easy to satisfy, because we could get Linux distribution with no cost and deploy it within minutes on a virtual machine, but the hardware resources are more crucial. I prefer to use cheap laptops and the last time I tried, the compilation of the Linux kernel took a lot of lime, so It is worth to notice, because it was only kernel compilation, excluding any packages. For modern machines it is not such a big problem, but anyway most tutorials are suggesting to get away from computer and enjoy a "large hot drink". Whenever it is not explicitly noted otherwise, I am using externally hosted dedicated server with Intel(R) Xeon(R) CPU E3-1270 v6 @ 3.80GHz (8 cores) and 32 GiB RAM. The comparison of resource usage could be found in Chapter 4.

Cross-compilation toolchain

As mentioned at the beginning of this chapter, cross-compilation is essential for the whole process, because embedded devices have different processor architecture than the host system in all considered cases. Before cross-compilation could start, the cross-compiler must already exist on the system or it will be created at the beginning. It needs to be done only once, but I am treating it as part of the complete build process.

The default compiler is obviously GCC in all cases, because it fits perfectly in the open source / Unix-like ecosystem and it is its base building block. There is also a possibility to select another compiler, like i.e. clang. We could choose different linker as well. The default one is the GNU linker (or GNU ld), but because of the enormous number of linking operations, different one like “gold linker” could be considered, which seems to be faster. That one is also a part of GNU Project. At the beginning I will choose the most basic setup, to make things easily comparable.

Target OS configuration

Target Operating System is the operating system of embedded device.

Boot loader

It is important to notice, that Linux kernel is not invoked directly by micro-controller at power on. The program whose main purpose it to load another system is called “boot loader” and in embedded device or any other device is not limited to have only one boot loader. They could start each other sequentially, so the first one is hard-coded into the micro-controller and it evokes the next one, which reside on the same external memory as Linux distribution. Examples include: coreboot, Libreboot, Das U-Boot and barebox (initially called U-Boot-NG).

Device tree

One of the most problematic issues when dealing with embedded device software is handling its hardware diversity. Even if the micro-controller architecture is exactly the same, lets say ARM Cortex-A9 or even the same SoC (System on a Chip) like TI OMAP4430, but devices differ greatly with issues like IO pins configurations and interfaces. Fortunately, there is an unified hardware description data structure called the “device tree”. It was initially developed within the U-Boot project, but currently (as of 2018) it is a de facto standard for both boot loaders and the Linux kernel.

Kernel

Device tree is rescuing Linux kernel from using weird and complicated mechanism of applying patches, but there is much more things to configure there beside hardware. Core security options as well as device drivers could not be provided in upper layers of operating system, but need to be compiled into kernel. Linux has its well established configuration system based on Makefiles, which could be managed text-based or graphically with invoking commands like `make config`, `make menuconfig` or `make xconfig`. This is also available from each embedded Linux build system, where we also have ability to choose separate and independently compiled kernel.

Init system

The first process that is evoked by the kernel is called “init”; Most popular are:

- busybox-init
- sysvinit
- systemd

Software packages

The highest layer of target OS configuration is managing software packages and this is where we see nobleness of described tools. Most of systems borrow the Makefile style from Linux kernel, but Yocto Project is notable exception and is using its own system named “bitbake”. Every system gives a possibility to choose from variety of prepared packages, but makes it also quite easy to add own software, which is the essence of whole process.

Produced output

For real-life rapid-prototyping and also for purpose of this publication, the most desired output is just one file with image, that could be flashed to SD card and making our Linux distribution run on target embedded device. Nevertheless, it’s the final product and also other useful semi-finished products becomes available. It could be grouped in the following way:

- cross-compilation toolchain
- compiled kernel
- compiled boot loader
- compiled packages
- root file system image
- SD-card image

Chapter 2

Development boards

From the beginning of embedded systems history, microprocessor development boards were just a training resource for engineers. Usually they were designed and produced by companies that have created certain chip and costs a lot of money, that it was not affordable for hobbyists or students. In recent years this state has been changed by a few factors:

- the spread of ARM architecture micro-controllers with efficiency comparable to personal computers (1GHz),
- creating development boards by community as open hardware, that resulted in cost reduction,
- adding peripherals specific not for embedded systems but for personal computers.

This led to discovering totally new use cases for development boards which explode due to widespread success of Raspberry Pi platform. I have selected some of most popular devices with ARM to compare them, but also included my personal laptop computer as a reference for x86 architecture. Because of very low popularity, I have not investigated other chips with SPARC or Power Architecture.

At minimum, each device has:

- at least one UART, for serial console
- at least one Ethernet port, 100 MB/s or higher
- at least one USB port, 2.0 or higher
- at least one GPIO pin
- SD card interface
- low voltage (5V-24V) power supply

2.1 Raspberry Pi 1

It was designed to be pocket size and "very low cost"[26], which together with its educational purpose, resulted in organizing big open source community around it.

The revolutionary approach here is that there is no need to configure any development environment on own host operating system, to start working with this embedded device. Special Debian-based distribution with graphical interface called "Raspbian" was prepared and its SD card image could be downloaded from `raspberrypi.org` website. A lot of deb packages were compiled for ARMv6 architecture and are available in binary form, so it is very easy to get familiar with it.[15] When standard modern PC devices like USB keyboard and HDMI display are connected, after plugging power through Micro-USB socket, it could be used as ordinary workstation. It is definitely not the most efficient way of embedded development, but that is how this board have changed rules of the game.

The initial release of Raspberry Pi in 2012 was so successful, that because of enormous number of orders, people had to wait even a few months to get their one. Beside price and enthusiastic hobbyists, also "Raspberry Pi Foundation", which is responsible for designing those boards, makes a big effort with providing documentation, educational materials and online forum.[27] The project takes a lot of inspiration from British "BBC Micro" educational computers, including the split into reduced Model A and Model B as a full version, which also takes the most attention.

The Broadcom BCM2835 System on a Chip that is used on first version of Raspberry Pi is using Vector Floating Point, which makes a lot of computing operations slower than with full-blown Floating-Point unit, but it also make the production cheaper. This is the reason why the same SoC is used in other new Raspberry Pi devices, including Industrial Compute Module and Raspberry Pi Zero.

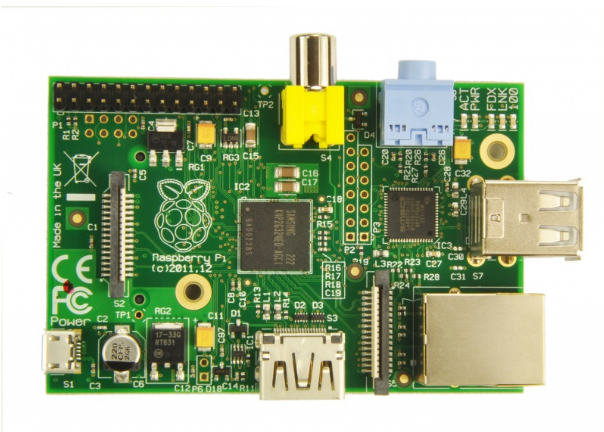

Figure 2.1: Raspberry Pi 1 Model B

2.2 Raspberry Pi 2

Despite mostly positive reviews for first version of Raspberry Pi, project was also criticized for some hardware design issues. Fragile SD card dock instead of microSD, electrolytic capacitor used for DC stabilization, wrong USB ports placement and unneeded composite video RCA jack was the most commonly complained.[11] Fortunately, all those remarks were taken into consideration and models A+ and B+ were released in 2014 without those drawbacks. The iconic hardware layout was established, although it was still “version 1” without any further innovations, because all of the core parts, including micro-controller, remains the same.

In the 2015, version 2 was released, with the same layout, but new Broadcom BCM2836 SoC. The most important is that it comes with newer ARMv7-A architecture. More and more project are being inspired by Raspberry Pi: from its expansions like pi-top modular laptop to imitations like Orange Pi or Banana Pi.[6]

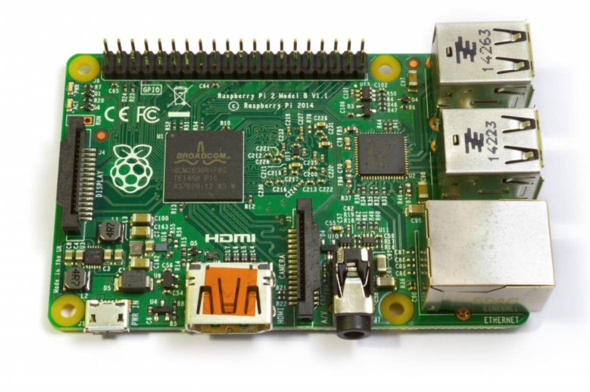

Figure 2.2: Raspberry Pi 2 Model B

2.3 BeagleBone Black

BeagleBone Black, together with preceding BeagleBone (White) and BeagleBoard are often named a competitors for the Raspbery Pi series. Most important difference is that they are build upon Texas Instrument SoCs.

The “Black”, that was released in 2013 gains a lot of attention from more advanced users, that were able to see some of its advantage points, especially in comparison to first version of Raspberry Pi. Most appraisal goes to well placed expansion connectors, standard 5V power jack, MII based Ethernet (not through USB) and 2 GB of eMMC flash memory with pre-installed Angstrom Linux distribution. From build systems perspective, it is a big plus that TI AM3358 SoC support, is included into Linux kernel mainline, which means that there is no need for special patches.

There are still more and more boards being developed from this community including BeagleBoard X15 (2016), BeagleBone Green, BeagleBoard Blue and competitor to Raspbery Pi Zero: PockerBeagle (2017).

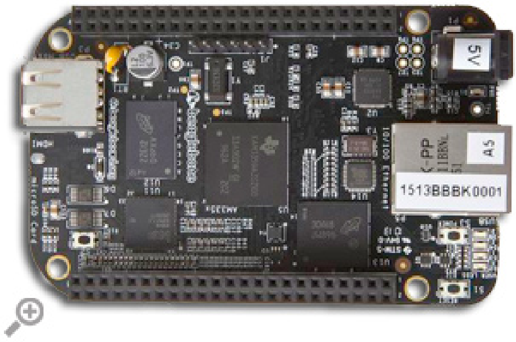

Figure 2.3: BeagleBone Black

2.4 PandaBoard

PandaBoard was released in 2010, so it could be called a predecessor of other boards in its class. Despite having very strong ARM Cortex-A9 based TI OMAP4430 chip, that was used also for top class smartphones, everything seems to be very unfortunate.

This board is very large compared to others, and high audio and Ethernet/USB connectors makes it especially unshapely, together with protruding full-size SD card. Ethernet is unfortunately provided just through USB hub chip. RS232 serial DB-9 connector is much less useful, that just making UART pins available, but using totally not popular USB type AB is even more ridiculous. After some initial popularity, most of community has scatted, that is especially painful when dealing with no support for its graphic hardware. The sad last cord happens when the domain `pandaboard.org` was not prolonged in 2017 and acquired by bots, which make all links from official documentation unavailable.

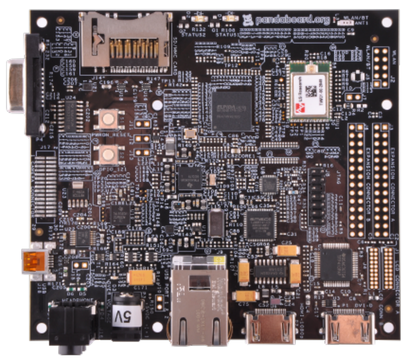

Figure 2.4: Pandaboard

Figure 2.5: Wandboard

2.5 Wandboard Quad

This board represents the most powerfull i.MX 6 series SoC that was produced by Freescale (currently NXP, but also acquired by Qualcomm). One of the best advantages from this series, is that there are many types (i.e. Solo, Dual, Quad), that are pin to pin compatible. That makes it very friendly to mass-scale, but diverse hardware development, so there exists three typed of WandBoard depending on number of cores. Many aspects and examples compatible with this board were described in “Linux w systemach i.MX 6 series” by Marcin Bis.[20]

2.6 x86_64 (Asus Eee PC 1215n)

There is nothing special about this laptop, I choose it mostly because that is the x86_64 device that I own. Because it is quite old and not aiming to be very powerful, its performance is comparable to development boards, like i.e. Wadnboard Quad. Fortunately it also have a SD card slot, so could be booted in the same way as others.

Figure 2.6: Asus Eee PC 1215n

2.7 Devices comparison

name	Raspberry Pi 1	Raspberry Pi 2	BeagleBone Black	PandaBoard	Wandboard Quad	Asus Eee PC 1215n
release date	April 2012	February 2015	April 2013	October 2010	February 2013	August 2010
target price	$35	$35	$45	$174	$129	$499
word size	32-bit	32-bit	32-bit	32-bit	32-bit	32-bit/64-bit
SoC	Broadcom BCM2835	Broadcom BCM2836	Texas Instruments AM3358/9	Texas Instruments OMAP4430	Freescale i.MX6 Quad	Intel Atom
architecture	ARM Cortex-A7	ARM Cortex-A8	ARM Cortex-A8	ARM Cortex-A9	ARM Cortex-A9	x86
CPU frequency	700 MHz	1000 MHz	1000 MHz	1000 MHz	1000 MHz	1800 MHz
RAM size	512 GB DDR3	1 GB	512 MiB DDR3	1 GB	2GB DDR3	2GB DDR3
Power source	5 V (Micro USB/GPIO)	5 V (Micro USB/GPIO)	Mini USB / 5 V jack	5V	5V	19V
USB	2 (via the onboard 5-port USB hub)	4 (via the onboard 5-port USB hub)	USB 2.0	two USB host ports and one USB On-The-Go	USB 3.0	USB 2.0 + USB 3.0
Network	10/100 Mbit/s Ethernet on the USB hub	10/100 Mbit/s Ethernet on the USB hub	Ethernet Fast Ethernet (MII based)	10/100 Ethernet on USB hub	GbE	10/100 Ethernet (MII based)
storage	microSDHC slot	microSDHC slot	4GB eMMC / microSDHC slot	SDHC slot	microSDHC	SATA (default 320 GB HDD)

Tabela. 2.1: Development boards comparison

Chapter 3

Linux build systems for embedded devices

In this chapter I will describe all embedded Linux build systems in order of rising complexity. Since there is nothing more practical than a good example, the list of all shell commands that are essential to run complete build process, is presented on the beginning of each section. It will be then discussed line by line, in a reference to OS build systems structure presented in Section 1.3. Suggested way is not to copy and paste entire script, but rather execute commands line by line. In some cases input from user is needed, what will be marked with commented lines (using #).

To run freshly created distribution on Raspberry Pi from SD card in smallest possible number of steps is for distribution builders something like like printing “Hello world!” for programming or “blinking a led” for electronics project. The most generic way seems to be just using Host OS as Target OS, but it excludes cross-compilation and testing system on real embedded device.

Each of the examples assumes, that it is executed on freshly installed Debian instance. To get prerequisites for all builders installed at once, following command could be executed:

```
sudo apt install make gcc g++ libncurses-dev unzip git \
  patch python python-dev rsync bc bzip2 gawk zlib1g-dev \
  bison flex tcl gettext lzop gawk diffstat texinfo \
  build-essential chrpath pkg-config pv
```

At the end of each script, the final image is being copied to home directory, to highlight its location. It could be especially useful when running build on special server and then downloading it to local machine in order to deploy it on real SD card. It could be done in many ways, even on Windows systems, but below I present the one that is most convenient for me:

```
pv sdcard.img | sudo dd of=/dev/sdb bs=4M oflag=dsync
```

With the use of `pv` tool, the progress of mission-critical `dd` could be monitored. Here, `/dev/sdb` is just most common example of SD card device, but each time it should be checked.

3.1 Buildroot

Figure 3.1: Buildroot logo

Buildroot is named a simple, efficient and easy-to-use tool to generate embedded Linux systems through cross-compilation. The name came from the process of building the root file system.

It's initial public release takes place at January 2005. Latest stable release is 2017.11.1. The project is backed by the group of open source developers, not directly by any organization, but gets financial sponsorship from various companies. Especially the involvement of Thomas Petazzoni, the Chief Technical Officer of Free Electrons is worth to notice.[12]

```
1  sudo apt install make gcc g++ libncurses-dev unzip git
2
3  wget https://buildroot.org/downloads/buildroot-2017.11.1.tar.bz2
4  tar -xjf buildroot-2017.11.1.tar.bz2
5  cd buildroot-2017.11.1/
6
7  export MACHINE=raspberrypi
8
9  make raspberrypi_defconfig
10 time make
11 cp output/images/sdcard.img ~
```

Source code

Buildroot is the simplest among build systems from every point of view. Its source code releases could be downloaded from main website in two popular archive formats: tar.bz2 and tar.gz. The example of downloading is shown in line number 3. Source code that do not belong into buildroot core is downloaded during build process.

Host OS requirements

This initial package requirements for host OS, visible in line 1, are the smallest among all build systems. They seems to be obvious, but because they are only a few, each could be

```
Welcome to Buildroot
buildroot login: root
# uname -a
Linux buildroot 4.9.52 #1 Sat Wed 24 12:00:00 UTC 2018 armv6l GNU/Linux
```

Figure 3.2: buildroot - raspberrypi

described:

- `make` - the main build utility, that run commands specified in `Makefiles`
- `gcc`, `g++` - C and C++ compilers, that will create a specyfic cross-compilation toolchain
- `libncurses-dev` - library that makes able to create `menuconfig` GUI in terminal
- `unzip`, `git` - most basic tools to getting source code and unpack archives

Cross-compilation toolchain

The buildroot toolchain is based on GCC and could be compiled with use of various C libraries, like uClibc-ng, glibc and musl. It is possible to use buildroot only for its creation, so that it could compile stand alone project or included as part of another builder. To achieve that, just command `make toolchain` need to be invoked.

Target OS configuration

As mentioned before, configuration management is borrowed from Linux kernel. `make menuconfig` will open terminal GUI and results could be saved in `.config` file. It is very convenient to just put that file under version control of target embedded project. There is also a lot of base configurations, that could be listed with `make list-defconfig` and applied as shown in line 9. The configuration of Linux kernel could be accessed via `make linux-menuconfig`

Produced output

The login prompt on embedded device is shown on picture below. It is a tradition that default and only available admin user is `root` with empty password. Of course it could stay that way only for development purposes.

3.2 OpenWRT

OpenWrt is named a Linux distribution for embedded devices. The name came from open source firmware for WRT54G series of wireless routers.

Figure 3.3: OpenWrt logo

It's initial public release takes place at January 2004. In May 2016 a group of its core developers decided to fork the project and create LEDE - Linux Embedded Development Environment. Fortunately in January 2018, it was officially announced that both projects will merge under the "OpenWrt" branding.

```
1  sudo apt install make gcc g++ libncurses-dev unzip git gawk file zlib1g-dev
2  git clone -b v17.01.4 https://github.com/openwrt/openwrt
3  cd openwrt/
4  ./scripts/feeds update -a
5  ./scripts/feeds install -a
6  make defconfig
7  make menuconfig
8  # Target System (Broadcom BCM27xx)
9  # Target Profile (Raspberry Pi B/B+/CM/Zero/ZeroW)
10 time make
11 cp build_dir/target-arm_arm1176jzf-s+vfp_musl-1.1.16_eabi/linux-brcm2708_bcm2708/tmp/lede-brcm2708-bcm2708-rpi-ext4-sdcard.img ~
```

Source code

To get specific version of OpenWrt code one should get it thought git SCM, eventually pointing to a specific version tag. Significantly, that from the beginning of 2018, merged source code is available from OpenWrt repository, but in some places still LEDE brand could be seen. It is specific for this system, that there is also a need to update and install package definitions, as shown in lines 4 and 5

Host OS requirements

Package preliminary requirements are just a little bigger than in Buildroot.

Cross-compilation toolchain

Since OpenWrt is a Buildroot-based fork, there is also a possibility to create only toolchain in a very similar way: `make toolchain/install`.

```
BusyBox v1.25.1 () built-in shell (ash)

     _________
    /        /\      _    ___ ___  ___
   /  LE    /  \    | |  | __|   \| __|
  /    DE  /    \   | |__| _|| |) | _|
 /________/  LE  \  |____|___|___/|___|                      lede-project.org
 \        \   DE /
  \    LE  \    /  -----------------------------------------------------------
   \  DE    \  /    Reboot (17.01.4, r3560-79f57e422d)
    \________\/    -----------------------------------------------------------

=== WARNING! =====================================
There is no root password defined on this device!
Use the "passwd" command to set up a new password
in order to prevent unauthorized SSH logins.
--------------------------------------------------
root@LEDE:/# uname -a
Linux LEDE 4.4.92 #0 Wed Oct 24 12:00:00 2017 armv6l GNU/Linux
```

Figure 3.4: openwrt - raspberrypi

Target OS configuration

Unfortunately, in this build system it is not possible to choose one of available base configurations from command line, so it need to be selected from menuconfig ncurses-based GUI. As OprnWrt puts primary focus on routers and other networking devices, it is strongly visible in great availability of related packages.

Produced output

Login prompt could be seen on included figure. LEDE logo is still shown, as a result of ongoing merge process.

3.3 LTIB

Figure 3.5: LTIB logo

LTIB is officially called a tool that can be used to develop and deploy BSPs (Board Support Packages) for a number of embedded target platforms. The name came from Linux Target Image Builder.

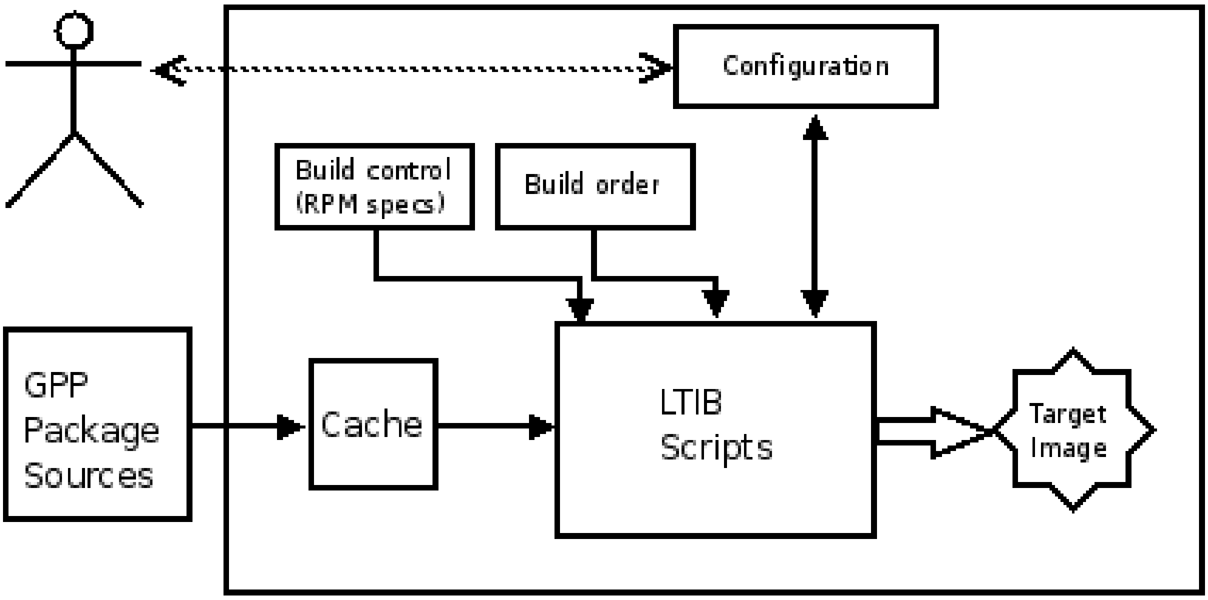

Figure 3.6: LTIB structure diagram

It was launched and financed by Freescale Seminconductor around 2004-2005. In 2009 project was transferred to Savannah, but its development stopped at 2013 with version 13.2.1.

```
sudo apt install make gcc g++ libncurses-dev unzip rpm bison patch tcl zlib1g-dev

wget https://github.com/downloads/midnightyell/RPi-LTIB/raspberrypi-tools-9c3d7b6-1.i386.rpm
sudo mkdir -p /opt/ltib/pkgs/
sudo cp raspberrypi-tools-9c3d7b6-1.i386.rpm /opt/ltib/pkgs/

sudo dpkg --add-architecture i386
sudo apt update
sudo apt install zlib1g:i386 libstdc++:i386

wget http://download.savannah.nongnu.org/releases/ltib/ltib-13-2-1-sv.tar.gz
tar -xzf ltib-13-2-1-sv.tar.gz
cd ltib-13-2-1-sv/
time ./ltib

# Platform choice (Raspberry Pi with BCM2835 SoC)

cp output/images/sdcard.img ~
```

Source code

Since LTIB development was moved to Savannah, it is possible to obtain tar.gz compiled code through HTTP from there, as shown in line 13. Source code was maintained with the use of archaic cvs SCM, which shows how long time ago it was initiated. In relation to source code, it is also worth to mention, that `./ltib` which is the initiating bash script, have 3000+ lines of code which for sure made it hard to maintain.

Host OS requirements

Host requirements are more and more intricate, due to halted LTIB development and rapid development of host systems, like Debian. There is a need to add i386 architecture support on host OS (line 7) and then install necessary packages (line 9)

Cross-compilation toolchain

What differs it from previously described systems, is that cross-compilation toolchain is installed globally. Because of that, there is a need to have administrative privileges, most preferably just `sudo` permissions. Some of platform (Raspberry Pi) specific packages are missing and they need to be prepared separately, as described in lines 3-5.

Target OS configuration

Here configuration is also based on `menconfig`. The step with choosing platform need to be performed manually, in the way that is commented on line 16.

Produced output

Unfortunately this build system is so old, that it could support only first version of Raspberry Pi and no other from presented development boards.

3.4 PTXdist

PTXdist is called a build system that creates embedded Linux distributions directly from the source code. The name came from combining shortcut of company Pengutronix and and word “distribution”.

Its initial public release dates back to 2010 and the latest stable version is 2017.12.0. Its developed and maintained by the German company Pengutronix.

Figure 3.7: PTXdist logo

```
sudo apt install make gcc g++ libncurses-dev unzip git gawk flex bison gettext python-dev lzop pkg-config libxml-parser-perl

wget http://public.pengutronix.de/software/ptxdist/ptxdist-2018.01.0.tar.bz2
tar -xjf ptxdist-2018.01.0.tar.bz2
cd ptxdist-2018.01.0
./configure
make
sudo make install
cd ..

wget http://public.pengutronix.de/software/ptxdist/ptxdist-2016.06.0.tar.bz2
tar -xjf ptxdist-2016.06.0.tar.bz2
cd ptxdist-2016.06.0
./configure
make
sudo make install
cd ..

wget https://public.pengutronix.de/oselas/toolchain/OSELAS.Toolchain-2016.06.1.tar.bz2
tar -xjf OSELAS.Toolchain-2016.06.1.tar.bz2
cd OSELAS.Toolchain-2016.06.1/
ptxdist-2016.06.0 select ptxconfigs/arm-1136jfs-linux-gnueabihf_gcc-5.4.0_glibc-2.23_binutils-2.26_kernel-4.6-sanitized.ptxconfig
ptxdist-2016.06.0 migrate
time ptxdist-2016.06.0 go

git clone https://git.pengutronix.de/cgit/DistroKit/
cd DistroKit/
ptxdist-2018.01.0 platform configs/platform-rpi/platformconfig
ptxdist-2018.01.0 migrate
time ptxdist-2018.01.0 images
```

```
31 cp platform-rpi/images/hd.img ~
```

Source code

For this build system it is not possible to download all its sources, because it is split into 3 parts:

- PTXdist - specific higher level (than i.e. make) build tool, aimed to "execute documentation"
- OSELAS.Toolchain - toolchain suggested by PTXdist
- DistroKit - named a Build Support Package example, it is where default configurations for platforms like i.e. Raspbery Pi came from

Host OS requirements

There is nothing specific about hot OS requirements, beside that it is written explicitly in documentation, that there is a need to install globally two versions of PTXdist (lines 3-17). Latest one will be used for all operations, with the exception for building cross-compilation toolchain, for which the older one is needed.

Cross-compilation toolchain

The OSELAS.Toolchain has not been updated since 2016 and that is why compatible ptxdist-2016.06.0 is needed. There is also a need to make an arbitrary decision about target cross compilation architecture, because it will be installed globally before configuring target. Example for Raspberry Pi is shown in lines 19-25. To create one for each architecture at once the script `./build_all_v2.mk` could be run, but since there are about 16 different, it will take enormous amount of time. It is also possible to use binary packages of those toolchain, what is available to download from Pengutronix website.

Target OS configuration

PTXdist command line tool could be used to create so called BSP for any platform, but configurations for popular development boards are available in DistroKit. For every other build system "u-boot" is chosen as default boot loader, but here it is the "barebox", which is being developed by Pengutronix company.

Produced output

For ARMv7 devices, target configuration it grouped under `platform-v7a`, so final distinction for building SD card image is made upon device tree file.

3.5 Yocto Project

Figure 3.8: Yocto Project logo

The Yocto Project is named an open source collaboration project that provides templates, tools and methods to help you create custom Linux-based systems for embedded products. Its name came from the smallest metric system unit prefix "yocto" that emphasize possibility to create tiny distributions.

The project under it's current name was formed in 2010-2011 and the latest stable version is 2.4 "Rocko" released at October 2017. The Yocto Project is a lab workgroup of the Linux Foundation and its developed and maintained by the group of major companies, including ...

```
wget http://commondatastorage.googleapis.com/git-repo-downloads/repo
chmod a+x repo
sudo mv repo /usr/local/bin/

sudo apt install make gcc g++ unzip git
sudo apt install gawk diffstat texinfo build-essential chrpath

export MACHINE=raspberrypi

mkdir yoctoproject
cd yoctoproject
repo init -u https://github.com/cdynak/yocto-manifest -m $MACHINE.xml
repo sync

source poky/oe-init-build-env
MACHINE=wandboard DISTRO=poky source setup-environment build
% vi conf/bblayers.conf (...?)
time bitbake core-image-minimal
cp tmp/deploy/images/$MACHINE/*img ~
```

Source code

The source code structure of Yoctro Project could be named the most complex, because it introduced concept of layers. They are divided into several groups:

- Base - there are only two core layers: `openembedded-core` and `meta-oe`, which need to be included for all projects
- Machine (BSP) - responsibility certain platforms support is split in this group, i.e. “meta-ti” layer provide configuration for both BeagleBone and PandaBoard and it is maintained with the help of Texas Instruments company
- Software - specific pieces of software are divided into layers like “meta-nodejs” for nodejs and “meta-ros” for Robotic Operating System packages
- Distribution -
- Miscealous - the rest, that do not fit into any other group, i.e. when it could be classified for more than one group

Host OS requirements

There is a need to have only basic packages on host operating systems, because everything else will be created in local directory as needed. Specific exception could be a “repo” tool, which installation is made in lines 1-3. It was borrowed from Android project, because it helps in to download multiple git repositories, based on provided XML configuration file. Layers could be downloaded and configured manually, but most of Yocto Project developer guides suggests to use this tool.

Cross-compilation toolchain

It is also possible to only create a toolchain with command `bitbake meta-toolchain`.

Target OS configuration

Target configuration is something what makes this tool very different from the others. It abandons the “Makefiles” executed by “make”, in favor to “recepies” executed by python tool “bitbake”. This concept comes from time before Yocto Project was formed and there exists only OpenEmbedded project, which was not aimed to provide easy support for all groups of layers. Because of its well known complexity, but also flexibility, it reshapes common understanding of “distribution” therm, i.e. popular Angstrom Linux distribution is represented by “meta-angstrom” and this is how it is being created for x86_64 targets.

Produced output

The login prompt on embedded device is shown on picture below.

```
Poky (Yocto Project Reference Distro) 2.2.3 raspberrypi /dev/ttyAMA0
raspberrypi login: root
root@raspberrypi:~# uname -a
Linux raspberrypi 4.4.50 #1 Wed Jan 24 12:00:00 UTC 2018 armv6l GNU/Linux
```

Figure 3.9: yoctoproject - raspberrypi

Figure 3.10: Linux From Scratch logo

3.6 CLFS

In the beginning, there was just LFS project that provides step-by-step instructions for building your own custom Linux system, entirely from source code. Its name came from Linux From Scratch. Its initial release was made in December 1999 and its latest stable version is 8.1 from September 2017. The project was initiated by Gerard Beekmans and gains support from many open source enthusiasts.

Then this project grows, the few sub-projects appeared, like Beyond LFS, to instruct future maintaining of custom distribution, as well as Cross LFS. LFS is well maintained and often have an official release, but CLFS have no regularity in that regard and no useful automation process, so it could be treated rather as a source of tips and tricks, than a build system.

It is also possible to i.e. build new system from within Raspberry Pi using LFS instructions, but it then would not include the essential cross-compilation process.

name	Buildroot	OpenWrt	LTIB	PTXdist	Yocto Project	CLFS
official website	`https://buildroot.org`	`https://openwrt.org`	`http://ltib.org`	`https://ptxdist.org`	`https://yoctoproject.org`	`http://clfs.org`
source code repository	`https://git.busybox.net/buildroot/`	`https://git.openwrt.org`	`http://cvs.savannah.gnu.org/viewvc/ltib/`	`https://git.pengutronix.de/cgit/ptxdist`	`https://git.yoctoproject.org`	`http://git.clfs.org`
license	GPLv2[4]	GPLv2[22]	GPLv2[17]	GPLv2[23]	GPLv2, MIT and others[34]	OPLv1[7]
last stable release date	2017.11	2017.01	2013.02	2018.01	2017.10	2014.10
release cycle	three month[5]	irregular (approx 1 year)	irregular (approx 2 years)[18]	one month[24]	six months[35]	irregular[8]
number of supported packages						
number of supported boards						
total number of required packages						
total size of required packages						
name						

Tabela. 3.1: Embedded Linux build systems overview

Chapter 4

Build process comparison

4.1 Build servers

In Chapter 3, only the software requirements for each build system were specified, but because of enormous amount of compilation happening, hardware resources are significant. Even if local PC or laptop have have very good parameters, it is very hard to provide at home Internet connection with constant bandwidth, which is important to compare time of each build process, because all package sources are being downloaded meanwhile. To have isolated environment with guaranteed parameters it is reasonable to use external hosting service.

I choose OVH, because it is the one, that I am used to, especially I enjoy user interface, which is even open sources at `https://github.com/ovh-ux/ovh-manager-web`. More reasonable arguments in favor to OVH is that it have data center in Poland (WAW) and enables direct access to OpenStack interface. For measure basic compilation time for each board and system respectively I use dedicated server, which is the most efficient option. Nevertheless, I also made tests with virtual machines through OpenStack, because it is very easy to change their parameters and compare growing build time. Initially, I used “serwery.pl” (subsidiary of “nazwa.pl” / NetArt) which also provides OpenStack platform, but it was removed in the middle of 2017.

4.2 OpenStack configuration

Before instructions for building target OS, presented in Chapter 3, could be executed, there is a need to create and provision virtual machine.

VM creation

1. Login to Horizon panel (https://horizon.cloud.ovh.net)

2. Select “instances” and “launch instance”

3. Choose instance name (build-server), type (c2-30-flex), image (Debian 9)
4. Choose or create key pair
5. Launch instance
6. Select “volumes” and “create volume” or... just select “Boot from image (creates a new volume)”
7. Choose volume name (build-volume), type (SAS), size (50 GB)
8. Create volume
9. Attach volume to instance

4.2.1 VM provisioning

1. identify (`$HOST_KEYS $HOST_USER $HOST_IP $VOLUME_ID`)
2. Login with (`ssh -i $HOST_KEYS $HOST_USER@$HOST_IP`)
3. Update system (`sudo apt update && sudo apt upgrade`)
4. Create partition on volume (`sudo mkfs.ext4 /dev/sdb`)
5. Mount volume (`sudo mount /dev/sdb /mnt/`)
6. (`sudo chown debian:debian /mnt/`)
7. Navigate to mounted directory (`cd /mnt/`)

4.3 Tests in respect to host OS

name	Raspberry Pi 1	Raspberry Pi 2	BeagleBone Black	PandaBoard	Wandboard Quad	Asus Eee PC 1215n
real time	17m33.619s	20m15.836s	26m12.604s	11m41.240s	11m39.519s	
user time	73m47.968s	74m30.368s	55m52.408s	49m39.200s	48m33.172s	
sys time	3m54.620s	3m58.084s	3m13.860s	2m31.172s	2m27.688s	
buildroot-2017.11.1/	5.4G	5.4G	6.0G	4.8G	4.7G	
sdcard.img	93M	93M	77M	69M	61M	
boot time	4.926830	5.575466		...	3.640106	

Tabela. 4.1: Buildroot build comparison

name	Raspberry Pi 1 (make)	Raspberry Pi 2	BeagleBone Black	PandaBoard	Wandboard Quad (make -j)	Asus Eee PC 1215n
real time	47m49.305s				31m32.177s	
user time	36m41.768s				43m14.808s	
sys time	2m14.996s				2m33.428s	
openwrt/	8.4G				8.1G	
sdcard.img	285M					
boot time	8.492709					

Tabela. 4.2: OpenWrt build comparison

name	Raspberry Pi 1	Raspberry Pi 2*	BeagleBone Black*	PandaBoard	Wandboard Quad	Asus Eee PC 1215n
real time	27m17.629s	20m8.447s	20m8.447s			
user time	87m36.840s	74m21.468s	74m21.468s			
sys time	9m22.424s	4m31.152s	4m31.152s			
Toolchain	15G	15G	15G			
real time	28m7.177s	25m44.733s	25m44.733s			
user time	60m39.504s	58m52.088s	58m52.088s			
sys time	4m13.768s	3m59.556s	3m59.556s			
DistroKit/	5.5G	4.9G / 7.3G	4.9G / 7.3G			
sdcard.img	84M					
boot time						

Tabela. 4.3: PTXdist build comparison

name	Raspberry Pi 1	Raspberry Pi 2	BeagleBone Black	PandaBoard	Wandboard Quad	Asus Eee PC 1215n
real time	34m17.598s	34m24.622s	31m25.839s	19m0.417s	35m19.749s	
user time	201m14.080s	201m43.528s	178m14.064s	97m14.776s	0m9.384s	
sys time	14m52.688s	14m40.556s	13m8.788s	9m18.528s	0m0.868s	
yoctoproject/	24G	24G	26G		24G	
sdcard.img	53M	53M				
boot time	4.983091	3.951380				

Tabela. 4.4: Yocto Project build comparison

name	C2-30	B2-30	C2-15	B2-15	C2-7	B2-7
vCPUs	8 x 3.1 GHz	8 x 2.3 GHz	4 x 3.1 GHz	4 x 2.3 GHz	2 x 3.1 GHz	2 x 2.3 GHz
RAM	30 GB	30 GB	15 GB	15 GB	7 GB	7 GB
price	1.542	1.049	0.765	0.519	0.395	0.272
real	24m13.863s	24m52.772s	34m21.638s	37m10.730s	50m4.480s	57m17.304s
user	73m29.016s	75m56.208s	71m51.860s	81m13.392s	72m39.468s	82m59.928s
sys	8m11.556s	9m15.448s	7m35.004s	9m39.060s	7m41.560s	9m27.172s
total cost	0.62 PLN	0.42 PLN	0.43 PLN	0.32 PLN	0.33 PLN	0.26 PLN

Tabela. 4.5: Comparison of VM size and build time (CPU instances)

name	R2-30	R2-15	S1-8	S1-4
vCPUs	2 x 2.4 GHz	2 x 2.4 GHz	2 x 2.4 GHz	1 x 2.4 GHz
RAM	30 GB	15 GB	8 GB	4 GB
price PLN/h	0.457	0.395	0.148	0.081
real	52m33.380s	53m30.474s	65m42.849s	127m16.353s
user	77m29.732s	80m41.752s	91m34.000s	101m2.204s
sys	7m55.024s	7m36.960s	12m8.204s	15m2.652s
total cost	0.4 PLN	0.35 PLN	0.16 PLN	0.17 PLN

Tabela. 4.6: Comparison of VM size and build time (RAM instances)

4.4 Tests in respect to host OS

Chapter 5

Example use cases

5.1 Node.js IoT application

Application specifications

It could be very fashionable nowadays to run essential parts of IoT applications “in the cloud”. The most important reason for that is having a possibility to make live updates at any moments, but it results in big maintaining effort after release. It could be more convenient to run everything what is possible locally, because it enforces better software quality and protect from Internet connection problems.

As an example, application from my 2015 engineering project “System wizualizacji i sterowania obiektami automatyki” (eng. “Visualization and control system of automation objects”) was ported from ANSI C/PHP into easy deployable Node.js framework.

Building Node.js into target OS

```
cd buildroot-2017.02.4/
make clean # because of wchar
make menuconfig

# Toolchain -> Enable WCHAR support
# Toolchain -> Enable C++ support
# Target packages
# Interpreter languages and scripting -> nodejs -> NPM for the target
# Networking applications -> openssh
# Networking applications -> ntp -> ntpd
# Libraries -> Database
# mysql support -> mysql variant (mariadb) -> mariadb server
```

5.2 Building containers

Containerization basics

In therms of software development, container is an instance of isolated user-space, that shares the kernel with its host operating system.[10] It could be described as lightweight virtual machine, because while VMs are realizing concepts of hardware abstraction, containers are abstraction for application layer. There is no need to emulate whole operating system layer, so they could use even 100-1000 less disk space. There are various implementations of container engines, like LXC or CoreOS rkt, but as for 2018 the most resonable choice is Docker, which is most popular, mature and stable.[16] While comparing the Linux build systems, we will not touch the container orchestration topic, that is essential in real deployments[2], but the produced containers will be fully compatibile with orchestation tools.

Docker installation on host OS

```
$ sudo apt install apt-transport-https ca-certificates curl
    ↪ gnupg2 software-properties-common
$ curl -fsSL https://download.docker.com/linux/$(. /etc/os-
    ↪ release; echo "$ID")/gpg | sudo apt-key add -
$ sudo apt-key fingerprint 0EBFCD88
$ sudo add-apt-repository "deb [arch=amd64] https://download.
    ↪ docker.com/linux/$(. /etc/os-release; echo "$ID") $(
    ↪ lsb_release -cs) stable"
$ sudo apt update
$ sudo apt install docker-ce
```

```
$ docker pull apline
$ docker images
$ docker run alpine ls -l
$ docker run -it alpine sh
```

Minimal container

```
$ cd output/images
$ mkdir extra extra/etc extra/sbin extra/lib extra/lib64
$ touch extra/etc/resolv.conf
$ touch extra/sbin/init
$ cp /lib/x86_64-linux-gnu/libpthread.so.0 /lib/x86_64-linux-gnu
    ↪ /libc.so.6 extra/lib
$ cp /lib64/ld-linux-x86-64.so.2 extra/lib64
$ cp rootfs.tar fixup.tar
$ tar rvf --overwrite fixup.tar -C extra .
$ docker import - test/basic-system < fixup.tar
```

```
$ docker run -t -i test/basic-system sh
```

Chapter 6

Conclusions

6.1 Possible enhancements

Including rpm setup

The description was simplified to focus only on Debian GNU/Linux, just because of my experience. It should be enhanced also for distributions based on Red Hat packages.

Regular yearly updates

Source code development and evolution of standards is so rapid nowadays, that only plain ANSI C, without any libraries, could be considered truly as portable and stable in longer period (`cc -ansi -pedantic`). The same issue is happening here, in the area of embedded Linux distributions development tools. The only way to make this publication usable for others is to update it constantly in yearly or even bi-yearly circles.

Continuous integration

According to previous statement, it will be reasonable to include modern development practices, like continuous integration, to make things easier to maintain. Each of build scripts, that are attached here, should be executed on CI server whenever new stable version of particular tool is released, including new versions of host OS.

6.2 Summary

There is a need for diversity. It is dangerous when one kernel becomes hegemonic. In healthy ecosystem, this project should be just "embedded unix build systems" - to also include MINIX, BSD, Darvin and HURD kernels. I wish I could name this work in this way one day. I could compare it to something like Ubuntu Bug # 1 - “Microsoft has a majority market share”.[21]

Bibliography

[1] Akademickie Stowarzyszenie Informatyczne. sesja.linuksowa.pl. [Online; access 2018-01-24].

[2] Amber Ankerholz. https://www.linux.com/news/8-open-source-CONTAINER-ORCHESTRATION-TOOLS-KNOW. [Online; access 2018-01-24].

[3] Arrow Electronics. https://www.arrow.com/en/research-and-events/articles/iot-operating-systems. [Online; access 2018-01-24].

[4] Buildroot. https://git.busybox.net/buildroot/tree/COPYING. [Online; access 2018-01-24].

[5] Buildroot. https://buildroot.org/downloads/. [Online; access 2018-01-24].

[6] Ceed Ltd. https://pi-top.com/products/pi-top/. [Online; access 2018-01-24].

[7] CLFS. http://trac.clfs.org/browser/clfs-embedded/LICENSE. [Online; access 2018-01-24].

[8] CLFS. http://clfs.org/files/BOOK/. [Online; access 2018-01-24].

[9] Dictionary.com. http://www.dictionary.com/browse/operating-system. [Online; access 2018-01-24].

[10] Docker Inc. https://www.docker.com/what-container. [Online; access 2018-01-24].

[11] elinux.org. https://elinux.org/RPi_Hardware. [Online; access 2018-01-24].

[12] Free Electrons. https://free-electrons.com/company/staff/thomas-petazzoni/. [Online; access 2018-01-24].

[13] Google Scholar. https://scholar.google.pl/scholar?q="embedded+linux+build+systems. [Online; access 2018-01-24].

[14] Google Scholar. https://scholar.google.pl/scholar?q="embedded+linux+build+system. [Online; access 2018-01-24].

[15] Jon Brodkin. https://arstechnica.com/information-technology/2013/03/how-two-volunteers-built-the-raspberry-pis-operating-system/. [Online; access 2018-01-24].

[16] Jérôme Petazzoni. http://osseu17.container.training/kube.yml.html#82. [Online; access 2018-01-24].

[17] LTIB. http://cvs.savannah.gnu.org/viewvc/ltib/ltib/COPYING?view=markup. [Online; access 2018-01-24].

[18] LTIB. http://download.savannah.nongnu.org/releases/ltib/. [Online; access 2018-01-24].

[19] Marcin Bis. http://www.bis-linux.com/szkolenia. [Online; access 2018-01-24].

[20] Marcin Bis. Linux w systemach i.mx 6 series. BTC 2015.

[21] Mark Shuttleworth. https://bugs.launchpad.net/ubuntu/+bug/1. [Online; access 2018-01-24].

[22] OpenWrt. https://git.openwrt.org/?p=openwrt/openwrt.git;a=blob;f=LICENSE. [Online; access 2018-01-24].

[23] Pengutronix. https://git.pengutronix.de/cgit/ptxdist/tree/COPYING. [Online; access 2018-01-24].

[24] Pengutronix. https://public.pengutronix.de/software/ptxdist/. [Online; access 2018-01-24].

[25] Per Christensson. https://techterms.com/definition/kernel. [Online; access 2018-01-24].

[26] Raspberry Pi Foundation. https://www.raspberrypi.org/help/faqs/#performanceHardware. [Online; access 2018-01-24].

[27] Raspberry Pi Foundation. https://www.raspberrypi.org/forums/. [Online; access 2018-01-24].

[28] Richard Stallman. https://www.gnu.org/gnu/gnu-linux-faq.html. [Online; access 2018-01-24].

[29] The Linux Foundation. https://wiki.linuxfoundation.org/celp/start. [Online; access 2018-01-24].

[30] The Wayback Machine. https://web.archive.org/web/20010721180347/elinux.org. [Online; access 2018-01-24].

[31] The Wayback Machine. https://web.archive.org/web/20030220110526/http://www.elinux.org:80/wiki/. [Online; access 2018-01-24].

[32] Tim Riker. http://rikers.org. [Online; access 2018-01-24].

[33] Who.is. https://who.is/whois/elinux.org. [Online; access 2018-01-24].

[34] Yocto Project. https://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/LICENSE. [Online; access 2018-01-24].

[35] Yocto Project. https://wiki.yoctoproject.org/wiki/Releases. [Online; access 2018-01-24].

List of Figures

List of Tables

Streszczenie

Celem tej pracy jest opisanie oraz porównanie narzędzi, które są wykorzystywane w procesie tworzenia systemu operacyjnego opartego o jądro Linux'a z przeznaczeniem na urządzenia wbudowane. Wszystkie z nich mają otwarty kod źródłowy i korzystają z licencji wolnego oprogramowania.

W rozdziale 1 podawane są definicje najważniejszych pojęć występujących w pracy, takich jak "system operacyjny", "jądro systemu operacyjnego" oraz "urządzenie wbudowane". Jako przegląd literatury zostały przedstawione najważniejsze strony internetowe oraz organizacje kształtujące systemy implementacji wbudowanego Linuxa. Ze względu na inżynierski charakter oraz bardzo dynamiczny rozwój tej dziedziny, materiały z ogólnopolskich oraz międzynarodowych konferencji środowisk związanych z Linuxem i wolnym oprogramowaniem przeważają nad książkami oraz artykułami naukowymi. Na postawie zgromadzonej wiedzy zaproponowany jest podział systemów budowania na logiczne części.

W rozdziale 2 zaprezentowane są urządzenia wbudowane, które będą wykorzystywane jako cele kross-kompilacji Linuxa. Są to zestawy deweloperskie służące do zapoznania się z danym mikrokontrolerem oraz do celów szybkiego prototypowania. Najbardziej popularna, a zatem posiadająca najlepsze wsparcie od społeczności jest seria Raspberry Pi, natomiast dla zapewnienia różnorodności wykorzystywane są także BeagleBoard, PandaBoard, WandBoard oraz klasyczny laptop Asus EeePC 1215n z procesorem Intel Atom (x86_64). Wobec tego reprezentowane są trzy najpopularniejsze mikroarchitektury ARM (Cortex-A7, Cortex-A8, Cortex-A9). Ze względu na posiadanie dużego zbioru wspólnych interfejsów, jak karta SD, USB oraz Ethernet, możliwe jest ich dokładne porównanie.

W rozdziale 3 opisane są tytułowe systemy budowania: Buildroot, OpenWrt, LTIB, PTXdist oraz Yocto Project. Po krótkim wstępie dla danego narzędzia podana jest lista poleceń, która wywołuje pełen proces budowy od pobierania źródeł z internetu po przygotowanie gotowego obrazu na kartę SD. Każda komenda jest omówiona nawiązując do podziału zaproponowanego w rozdziale 1. Bez przykładów uruchomienia zaprezentowane są także bardziej oryginalne projekty spełniające definicje, takie jak CLFS (Cross Linux From Scratch) oraz Debian Multistrap.

W rozdziale 4 przedstawione są wyniki eksperymentalnego procesu budowy. Ze względu na wielką ilość obliczeń oraz konieczność zapewnienia niezawodnego łącza o stałej prędkości, wszystkie procesy są wykonywane na specjalnie wynajętym do tego celu dedykowanym serwerze z procesorami Intel Xeon. Zbierane są metryki takie jak czas i całkowita wielkość folderu budowy, rozmiar systemu operacyjnego i czas od włączenia urządzenia do pojawienia się zachęty logowania. Eksperymenty podzielone są na dwie części: w pierwszej dla stałej mocy obliczeniowej wykonywane są procesy budowania dla różnych systemów i

urządzeń wbudowanych, natomiast w drugiej pokazana jest zależność między szybkością maszyny a czasem trwania budowania.

W rozdziale 5 znajdują się praktyczne przykłady zastosowania systemów budowania. Najpierw omówiona jest aplikacja przygotowana przez autora wcześniej w ramach projektu inżynierskiego oraz metoda jej wdrożenia na urządzenie wbudowane. Ponieważ głównym wymaganiem jest zainstalowanie środowiska uruchomieniowego Node.js, w związku z tym przedstawiono sposoby na dodanie go poprzez każdy system budowania. Tak przygotowane urządzenie może być wykorzystywane jako brama Internetu Rzeczy. Następnie ukazany jest sposób tworzenia kontenerów przy użyciu wyników z dowolnego omówionego systemu budowania. Kontenery, opisywane jako lekkie maszyny wirtualne, są obecnie coraz częściej wykorzystywane w zastosowaniach produkcyjnych, szczególnie chmurowych. Przygotowany obraz systemu operacyjnego jest kompatybilny z popularnymi narzędziami orkiestracji, w tym przede wszystkim z Kubernetesem.

W rozdziale 6 wysnute są wnioski i rekomendacje na dalszy rozwój tej publikacji oraz samych narzędzi budowania dystrybucji Linuxa. Przede wszystkim zawartość musi być aktualizowana przynajmniej raz do roku ze względu na częste zmiany, jakie zachodzą w wolnym oprogramowaniu. Finalnie wyrażony jest niepokój, że mimo posiadania szerokiej gamy uniwersalnych narzędzi, wciąż obsługują one jedynie jądra Linuxa. Jak w każdym obszarze brak konkurencji może w dłuższej perspektywie doprowadzić do spadku jakości. Autor wyraża nadzieję, że wykorzystując te same narzędzia będzie można w przyszłości zbudować systemy oparte o jądra MINIX, BSD, Darvin oraz HURD.