

Juhani Takalo

# **A BUILD ENVIRONMENT FOR CREATING A CUSTOM LINUX DISTRIBUTION FOR TRAIN INFORMATION SYSTEM DEVICES**

Creating a Linux distribution with the Yocto Project

Master's thesis  
Faculty of Information Technology and Communication Sciences  
Examiners: Prof. Karri Palovuori  
Prof. Jukka Vanhala  
March 2021

# ABSTRACT

Juhani Takalo: A build environment for creating a custom Linux distribution for train information system devices  
Master's thesis  
Tampere University  
Master's Programme in Electrical Engineering  
March 2021

---

The objective of this work was to investigate the possibility of using the open source tools provided by The Yocto Project, especially the task and build engine BitBake, to replace the old device image toolchain called Imagetools, which is created by Teleste for its train information system devices. While Imagetools mainly use Debian Linux distribution as the base for device images, the build environment created in this work is capable of building a fully working custom Linux distribution named Teleste Sky Blue for Teleste's embedded device DCU40, which is the main controller unit for Teleste's train passenger information systems.

The work begun by inspecting the Yocto recipes and instructions for DCU40's processor card by Seco, which is the manufacturer of the card. These recipes covered the GPIO expanders, SPI and I<sup>2</sup>C. First of all, these few recipes were updated to support the newest stable Yocto Project version 3.1, codenamed Dunfell. Builds were tested by installing them on DCU40's internal eMMC memory with the live USB flash drive image, which was generated with the BitBake, with network boot and in virtual environments. An automation script was written for pushing the cache items generated by the BitBake builds to Teleste's company network file share which made subsequent builds faster.

In the next phase, more features were added to the build to produce an image which would have the same base functionalities than the "base image" produced by Imagetools. These include kernel and software configurations, and Teleste's platform packages which enabled a full support for DCU40's features. The base software of a "base image" includes a Python interpreter with necessary packages, Java Runtime Environment with public and Teleste's libraries, PostgreSQL with default database, and Lighttpd HTTP-server with Teleste's Update Manager Web interface. New BitBake recipes were written for Teleste's platform components and existing recipes from Yocto's reference distribution named Poky and OpenEmbedded Layer Index were modified to match the existing Imagetools "base image".

After the base features of DCU40 were added to the build, recipes were written for the rest of Teleste's platform components, which are used in customer projects. Customer device images are also created with Imagetools, together with update packages uploaded from the Update Manager. However, with BitBake it is possible to automate the whole customer image creation process so that no other extra steps are needed to be taken in production for the customer project images. Also, the rest of the Imagetools features were added to the build environment, such as fully automated live USB flash drive image creation and VirtualBox image creation. In addition, more features were implemented to the build environment including Qemu virtualisation on command line for automated software tests, network booting of DCU40 with iPXE, and virtual machine images with nested virtualization support for virtual machine development and usage on DCU40.

During the work, the environment was used in investigation of DCU40's processor card's Intel microcode update performance degradation related to Meltdown and Spectre vulnerability mitigations. Eventually more and more thought and planning was put in to the build environment itself so that it would be scalable and useful on developers PCs to CI servers. The build environment created in this work is a clean and portable Git repository, with documented features which could eventually supersede Imagetools and overcome the challenges and problems it has.

Keywords: Yocto Project, Linux distribution, BitBake, build environment, embedded system

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

# TIIVISTELMÄ

Juhani Takalo: Käännösympäristö mukautetun Linux-jakelun luomiseksi junainformaatiojärjestelmälaitteille  
Diplomityö  
Tampereen yliopisto  
Sähkötekniikan DI-tutkinto-ohjelma  
Maaliskuu 2021

---

Tämän työn tavoitteena oli tutkia mahdollisuutta käyttää Yocto projektin tarjoamia työkaluja, kuten tehtävä- ja käännöshallintaohjelma BitBakea, korvaamaan Telesten vanha junainformaatiolaitteiden levykuvien luontijärjestelmä, jonka nimi on Imagetools. Imagetools käyttää pääosin Debian Linux-jakelua levykuvien pohjana mutta tässä työssä luotu käännösympäristö pystyy tuottamaan täysin toimivan mukautetun Linux-jakelun nimeltään Teleste Sky Blue Telesten sulautetulle laitteelle DCU40:lle, joka toimii pääkeskustietokoneena Telesten junien matkustajainformaatiojärjestelmissä.

Työ alkoi DCU40:n prosessorikortille tehtyjen Yocto reseptien ja ohjeiden tutkimisella, jotka kortin valmistaja Seco oli kirjoittanut. Nämä reseptit kattoivat ainoastaan GPIO-laajennuspiirit, SPI:n ja I<sup>2</sup>C:n. Ensiksi nämä reseptit päivitettiin tukemaan uusinta vakaata Yocto:n versiota 3.1, jonka koodinimi on Dunfell. Käännöksiä testattiin asentamalla niitä DCU40:n eMMC-muistille BitBakella tehdyllä live USB levykuvalla, verkkokäynnistyksen avulla ja virtuaalisissa ympäristöissä. BitBaken välimuistin hallintaa varten luotiin automaatiokripti, jolla pystyy kopioimaan BitBaken tuottamia ja käyttämiä välimuistiobjekteja Telesten tiedostopalvelimelle, mikä nopeuttaa seuraavia käännöksiä.

Seuraavassa vaiheessa käännöksiin lisättiin lisää ominaisuuksia, jotta tuotettu levykuva vastaisi toiminnallisuudeltaan Imagetoolssin tuottamaa "pohjalevykuvaa". Näihin ominaisuuksiin kuuluu kernelin ja ohjelmistojen asetuksia ja Telesten sovellusalustan ohjelmia, jotka mahdollistavat täyden tuen DCU40:n ominaisuuksille. Pohjalevykuvan kantaohjelmistoja ovat muun muassa Python tulkki tarvittavilla paketeilla, Java Runtime Environment julkisten ja Telesten kirjastojen kanssa, PostgreSQL vakiotietokannalla ja Lighttpd HTTP-palvelin Telesten Update Manager Web käyttöliittymällä. Uusia BitBake reseptejä kirjoitettiin Telesten ohjelmistokomponenteille ja olemassa olevia reseptejä Yocton referenssi linux-jakelulusta Poky:sta ja OpenEmbedded:n Layer Index:stä muokattiin, jotta toiminnallisuus olisi identtinen Imagetoolssin pohjalevykuvan kanssa.

DCU40:n perusominaisuuksien lisäämisen jälkeen reseptit kirjoitettiin myös muille Telesten ohjelmistoalustan komponenteille, jotka ovat käytössä asiakasprojekteissa. Asiakkaiden levykuvat tehdään myös Imagetoolssilla, joihin lisäksi asennetaan tarvittavat päivityspaketit Update Managerin kautta. BitBaken avulla on kuitenkin mahdollista automatisoida asiakaalle menevien levykuvien luonti, jolloin tuotannossa ei tarvita lisävaiheita niiden asennuksessa. Imagetoolssin muutkin ominaisuudet kuten automaattinen live USB levykuvien ja VirtualBox virtuaalikoneiden luonti lisättiin käännösympäristöön. Tämän lisäksi käännösympäristöön lisättiin uusia ominaisuuksia, joita ovat esimerkiksi Qemu:lla virtualisointi komentorivillä automaattisille ohjelmistotesteille, DCU40:n verkkokäynnistys iPXE:n avulla ja tuki sisäkkäisille virtuaalikoneille virtuaalisten ympäristöjen kehittämiseen DCU40:lle.

Työn aikana käännösympäristöä käytettiin DCU40:n prosessorikortin Intelin mikrokoodipäivitysten aiheuttaman tehoaleneman selvitykseen, joka liittyy Meltdown ja Spectre haavoittuvuuksien korjauksiin. Lopulta yhä enemmän aikaa käytettiin itse käännösympäristön kehittämiseen, jotta se olisi skaalautuva ja käyttökelpoinen kehittäjien tietokoneilla ja jatkuvan iteraation käännöspalvelimilla. Tässä työssä luotu käännösympäristö on siisti ja siirrettävä Git projekti dokumentoiduilla ominaisuuksilla, joka voisi tulevaisuudessa korvata Imagetoolssin ratkaisten sen haasteet sekä ongelmat.

Avainsanat: Yocto Project, Linux-jakelu, BitBake, käännösympäristö, sulautettu järjestelmä

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

## PREFACE

This work was carried out at Teleste, Tampere from spring 2020 to spring 2021.

When I started at Teleste as a summer trainee 2017, I hardly knew anything about Linux and couldn't have imagined that I would be creating a custom Linux distribution build system for Teleste as a Master's thesis. While repairing devices at the service, I came across a device image building toolchain called Imagetools, towards with I have created a love-hate relationship: it and its quirks got me interested to the internal Unix tools it uses, and with the help of my colleagues, I started to improve the features and functionalities of Imagetools. Eventually I was in a team with my student colleague friends, and together we did some actual developing for Imagetools. Eventually, this team shrunk when my friends moved forward seeking new challenges. However, as the author of this thesis, and current developer of Imagetools, I can confidently say that I managed to create something which could replace Imagetools in the future.

I wish to thank all of my friends and colleagues who I have been seeing and working with during this COVID-19 pandemic, which has given me energy to push through with this work. I also look forward seeing my colleagues again on daily basis, which is a thing I have missed a lot. Also, I want to thank my examiners: Prof. Jukka Vanhala, whose course about microcontrollers was one the most educational courses I had, and Prof. Karri Palovuori, whose courses were the most interesting at the university. I still have a fully working thermal camera smartphone which was designed by Karri. Lastly, I want to thank my boss Jukka Saari, who always listened my ideas and raised Teleste's software development to the next level.

Tampere, 26th March 2021

Juhani Takalo

# CONTENTS

1	Introduction . . . . .	1
2	Linux and The Yocto Project . . . . .	3
2.1	Linux in general . . . . .	3
2.2	Debian . . . . .	3
2.3	The Yocto Project . . . . .	4
2.4	BitBake . . . . .	4
2.5	Poky and The OpenEmbedded Project . . . . .	6
2.6	The Yocto Project Layer Model . . . . .	6
2.7	BitBake recipes . . . . .	7
2.8	BitBake build configuration files . . . . .	7
2.9	Recipe classes . . . . .	8
2.10	Recipe append files . . . . .	9
2.11	BitBake tasks . . . . .	9
2.12	The Yocto Project Extensible SDK . . . . .	10
2.13	OpenEmbedded Image Creator Wic . . . . .	10
2.14	Toaster web interface . . . . .	11
3	Using the build environment . . . . .	12
3.1	Setting up the project as a Git repository . . . . .	12
3.2	Building for DCU40 . . . . .	12
3.3	Cache management . . . . .	13
3.4	Directory structure of the project environment . . . . .	15
3.5	Git management guidelines . . . . .	16
3.6	Bash and Shell scrips . . . . .	17
4	Configurations for the DCU40 . . . . .	19
4.1	Target hardware device . . . . .	19
4.2	Introduction to dcu40-image . . . . .	20
4.3	Target machine configuration . . . . .	20
4.4	Distribution configuration . . . . .	21
4.5	Systemd as init manager . . . . .	22
4.6	Seco packages and kernel modules . . . . .	22
4.7	Package manager and Debian package format . . . . .	23
4.8	Keyboard layout . . . . .	23
4.9	Systemd-boot or GNU GRUB as bootloader . . . . .	24
4.10	BusyBox and GNU Core Utilities . . . . .	24
4.11	Kernel configurations . . . . .	24

4.11.1	Analyzing the build size of individual kernel features . . . . .	26
4.12	Disabling serial-getty . . . . .	26
4.13	Enabling magic SysRq key . . . . .	27
4.14	Java 8 . . . . .	27
4.15	Python 2 . . . . .	28
4.16	PostgreSQL 12 . . . . .	28
4.17	Live USB image . . . . .	29
4.18	Uninative settings . . . . .	30
4.19	Audio and graphics . . . . .	30
5	Deploying the build on the DCU40 and virtual environments . . . . .	31
5.1	Build artifacts . . . . .	31
5.2	Running the build with runqemu command . . . . .	32
5.3	Running the build in VirtualBox . . . . .	33
5.4	Using the live image . . . . .	35
5.5	Network boot of DCU40 . . . . .	36
5.5.1	Setting up a TFTP server . . . . .	36
5.5.2	Setting up a DHCP server . . . . .	37
5.5.3	Chainloading iPXE . . . . .	38
5.5.4	NFS root filesystem . . . . .	39
6	A comparison of BitBake and Teleste's Imagetools . . . . .	40
6.1	History of build automation at Teleste . . . . .	40
6.1.1	Rocket Tools . . . . .	40
6.1.2	Maven . . . . .	41
6.1.3	Jenkins . . . . .	41
6.1.4	Imagetools . . . . .	42
6.2	BitBake compared to Imagetools . . . . .	46
6.3	BitBake workflow . . . . .	48
6.3.1	Local meta-layers . . . . .	48
6.3.2	Remote meta-layers and Poky reference distribution . . . . .	48
6.3.3	Build configuration for DCU40 . . . . .	49
7	Testing . . . . .	51
7.1	Boot-up times . . . . .	51
7.1.1	USB flash drive flashing speed . . . . .	51
7.2	Meltdown and Spectre mitigations performance impact . . . . .	52
7.2.1	Meltdown and Spectre vulnerabilities . . . . .	53
7.2.2	Mitigating Meltdown and Spectre in DCU40 . . . . .	54
7.2.3	Analyzing test results . . . . .	55
8	Conclusions and outlook . . . . .	58
	References . . . . .	60

## LIST OF SYMBOLS AND ABBREVIATIONS

ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
APT	Advanced Package Tool
ARM	Advanced RISC Machines
AVR	Atmel AVR Microcontroller
Bash	Bourne again shell
BIOS	Basic Input/Output System
BitBake	Yocto Project's build and task execution engine
BSP	Board Support Package
CI	Continuous Integration
CPU	Central Processing Unit
CSM	Compatibility Support Module
DCU40	Diagnostic and Controller plug-in Unit, series-40
DHCP	Dynamic Host Configuration Protocol
EFI	Extensible Firmware Interface
ELF	Executable and Linkable Format
eMMC	Embedded Multimedia Card
FPGA	Field Programmable Gate Array
GNSS	Global Navigation Satellite System
GPG	GNU Privacy Guard
GPIO	General Purpose Input/Output
GPS	Global Positioning System
GRUB	GRand Unified Bootloader
GUI	Graphical User Interface
HDD	Hard Disc Drive
HTTP	Hypertext Transfer Protocol
I <sup>2</sup> C	Inter-Integrated Circuit
IC	Integrated Circuit
IP	Internet Protocol

KVM	Kernel-based Virtual Machine
LED	Light-Emitting Diode
LTE	Long-Term Evolution
LTS	Long-Term Support
MAC	Media Access Control
make	GNU Make build automation tool
MBR	Master Boot Record
NFS	Network File System
OS	Operating System
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCM	Pulse-Code Modulation
pip	Python package manager
PIS	Passenger Information System
pITX	Pico Information Technology eXtended
Poky	Reference distribution of the Yocto Project
PXE	Preboot Execution Environment
RAM	Random-Access Memory
RPC	Remote Procedure Call
RTC	Real-Time Clock
SoC	System on a Chip
SPI	Serial Peripheral Interface
SSD	Solid-State Drive
SSH	Secure Shell
TFT	Thin-Film Transistor
TFTP	Trivial File Transfer Protocol
Toaster	Web interface for BitBake
UART	Universal asynchronous receiver-transmitter
UEFI	Unified Extensible Firmware Interface
USB	Universal Serial Bus
VCS	Version Control System
VLAN	Virtual Local Area Network
WWAN	Wireless Wide Area Network
XML	Extensible Markup Language



# 1 INTRODUCTION

The initial objective of this work was to create a proof-of-concept custom Linux distribution with the Yocto Project for Teleste's Diagnostic and Controller plug-in Unit, DCU40, which is one of the main components in Teleste's train passenger information systems. Before this, device images have been created with Teleste's image creation tool called Imagetools, which uses Linux Debian distribution as a base image on top of which the full image is built on. There has been attempts to use the open source tools provided by the Yocto Project to create a new build toolchain to replace Imagetools, which have mainly been unsuccessful because of problems introduced by the new and more complex environment the Yocto Project provides compared to Imagetools. According to the interviews conducted during this work, there were a few main reasons for these unsuccessful attempts:

- The initial complexity and steep learning curve of the The Yocto Project
- Long rebuild times without proper cache handling
- A lack of an example build toolchain
- Complexity of Teleste's embedded devices which have multiple features built into them
- Completely different working principle of the Yocto Project compared to Imagetools

After the initial objective was reached, the image was developed further to match the base image created with Imagetools. After this, more features were added to the image and to the build environment. Also, more build target devices were added to the build environment, which eventually changed the core objective of this work from just a single device image made for DCU40, to a build environment capable of building a custom Linux distribution for Teleste's devices with different architectures.

Chapter 2, *Linux and Yocto Project*, explains what are Linux, Debian and the Yocto Project including the tools it uses such as BitBake together with terms and features related to it. Chapter 3, *Using the build environment*, describes how the build environment is set up as a Git superproject with other Git repositories as submodules, and how to start using the environment for running builds. Final sections of the chapter are dedicated for guidelines about Git management and Shell scripting in the build environment.

Chapter 4, *Configurations for DCU40*, introduces the device hardware and the build configurations for the device divided to sub sections. These configurations include features which are usually found in the base image created with Imagetools, like the support for all

hardware features, a set of Linux utilities and programs, system tweaks and configurations. Chapter 5, *Deploying the build on the DCU40 and virtual environments*, explains how to deploy the build on the actual hardware in different ways by installing it on the internal eMMC memory of DCU40 or by using network booting with iPXE. Instructions are also given on how to run the build virtualized with Qemu and VirtualBox.

Chapter 6, *Comparison of BitBake and Teleste's Imagetools*, dives into the history of build automation at Teleste, how the used tools have evolved and how the current build tool Imagetools compares to Yocto Project's BitBake. Visual illustrations about both tools are provided.

Chapter 7, *Testing*, includes tests related to the environment and since the newest BIOS versions for DCU40's processor card introduces Meltdown and Spectre vulnerability mitigations as Intel microcode updates, the possible performance degradation caused by them was investigated.

Finally, Chapter 8, *Conclusions and outlook*, finishes this thesis and gives an outlook on the future of this build environment.

## 2 LINUX AND THE YOCTO PROJECT

### 2.1 Linux in general

Linux is an open source operating system which is based on the Linux kernel. The kernel is a computer program which acts as a bridge between applications and hardware. Linux is usually used as a distribution, which is a ready made operating system with Linux kernel, system software and libraries. Popular Linux distributions include Debian, Ubuntu, Fedora, Red Hat and CentOS which can have different use cases, but can still be interchangeable. Even though they might share the same major Linux kernel version, the set of system software, libraries and general working principles are usually different. These distributions can be installed on different kinds of target devices with different architectures and hardware. For example, Ubuntu is a popular desktop distribution but there is also a version of it which is suitable for servers, and another one for ARM based devices. However, Ubuntu is actually based on Debian, and it differs from Debian by having a newer kernel and software compared to it.

In this work a custom Linux distribution named Teleste Sky Blue was built by using the Yocto Project version 3.1, codenamed Dunfell, which includes Yocto's reference distribution Poky version 23.0 with Linux kernel version 5.4.

### 2.2 Debian

Debian is a free and open-source Linux distribution and it is commonly used in different types of machines, from desktop PCs to embedded system devices and servers. This popular distribution is tested before releases, and Debian has at least three releases in active maintenance: stable, testing and unstable. The stable distribution of Debian is the latest official production release, which is recommended to be used in normal use cases. At the time of writing the current stable release version of Debian is 10, codenamed "buster".

The testing distribution of Debian has a newer kernel version and it contains newer versions of packages which have not been yet accepted to the stable release. The current testing distribution is codenamed "bullseye". The unstable Debian distribution is meant for active development, and it is codenamed always as "sid". During the past decade, there has been a stable Debian distribution release roughly every two years.

Every stable Debian release has 3 years of full support and 2 years of extra *Long Term Support*. This means that during those 3 years, the Debian security team will actively handle the security updates of that stable release. Debian LTS is a project to extend the lifetime of all stable Debian releases to at least 5 years and it is handled by volunteering individuals and companies.

Since Debian is the most used distribution in Teleste's embedded devices, Debian features were added to the Sky Blue to make future transition to it easier. These features include:

- Debian packages and dpkg package manager
- SystemD init system with SysVinit script compatibility

Debian package format is the most used archive format for software on Teleste's devices and many software components rely on SystemD units or SysVinit type init scripts.

## 2.3 The Yocto Project

The Yocto Project is not a Linux distribution, but an umbrella project including Poky reference distribution, task execution and build engine BitBake, OpenEmbedded-Core with package recipes, and several other components and tools for developing custom Linux distributions. With this collection of tools it is possible to customize every aspect of the build, from Linux kernel to included system software and libraries. It is very suitable for embedded system devices since the user has a total control on everything that is installed on the device. Compared to a full desktop distribution, a Yocto build can be configured to include just the things that are needed for the target embedded device. One of the important aspects of the Yocto Project is, that it builds a custom Linux distribution and its packages from source code rather than using prebuilt packages and binary files. Because of this, the source code of every component can be modified and patched in a modular way, so that the the source code can be always reverted to its original state.

Teleste has already used Yocto Linux in TFT and LED display devices, because it can be a lightweight Linux distribution for small form factor embedded devices. Normally in these applications, the display devices do not need to run many applications but just a few related to displaying the information and reporting diagnostics. Also, having a Linux environment compared to a bare metal implementation offers better code re-usability, wider feature base, and a familiar environment to develop for.

## 2.4 BitBake

The Yocto Project uses a parallel task execution and build engine written with Python 3 called BitBake, which automates the process of creating customized Linux distributions. BitBake is sometimes compared to GNU make, however, during building BitBake actually calls GNU make when the source code compilation is handled with Makefiles. According to the Yocto Project Mega-Manual[1], some key points for BitBake are:

- BitBake uses metadata from recipes denoted by filename extension `.bb`, configurations from `.conf` files and inheritable `.bbclass` files which provide instructions for BitBake on what tasks to run and dependencies between them. On GNU make, the counterpart for these files would be Makefiles
- BitBake can run Shell and Python functions within a single recipe file, and they can use the same variables
- BitBake has a built-in fetcher library for obtaining source code from various places like source control systems such as Git (with submodule support) and Subversion
- BitBake includes a client-server abstraction, and it can be used from a command line or used as a service over XML-RPC. It also has multiple user interfaces like the Toaster web interface
- BitBake does automatic syntax check for files it uses, and it detects changes in them between the builds
- BitBake runs a set of default tasks for recipes which can be modified or overwritten and more tasks can be added

Originally BitBake was part of the OpenEmbedded project, and it was inspired by the Portage package management system from Gentoo Linux distribution, which also builds packages from source code rather than using prebuilt packages. In 2004 OpenEmbedded was split into two parts:

- BitBake task execution engine
- OpenEmbedded metadata set used by the BitBake

Now BitBake is used as the primary basis for OpenEmbedded project and Yocto Project. According to the history section of Yocto Project's BitBake User Manual, before BitBake there wasn't any adequate system for building embedded Linux distributions with a feature set as comprehensive as BitBake's. Some of the important features and original goals for BitBake, according to the BitBake User Manual, were [2]:

- Cross-compilation support
- Dependencies management during build and runtime
- Support for running wide variety of tasks including fetching upstream sources, unpacking them, patching them, configuring them, building and installing them
- New tasks can be written for it
- It is Linux distribution agnostic for both build and target systems
- It is architecture agnostic
- It has support for multiple build and target operating systems
- It is self-contained and not tightly integrated into the build host's filesystem

- It can handle conditional metadata depending on target architecture, operating system, distribution and machine
- It includes tools to handle, modify and create source code
- It calculates a checksums for tasks, which allows the use of a shared state cache which accelerates subsequent builds

## 2.5 Poky and The OpenEmbedded Project

Poky is Yocto Project's reference distribution, which can be used to build a working Linux distribution without any extra configuration [3]. As a Git repository it consists of BitBake task execution engine, OpenEmbedded-Core with over 700 recipes and metadata such as configuration files, patches, and shared classes. Poky is not a production level Linux distribution but it is a good starting point for building one. It began as an open-source project initially developed by OpenedHand, and after Intel Corporation acquired OpenedHand, the Poky project became the reference distribution for the Yocto Project's build system.

Poky uses a six-month release cycle, and major releases occur at the same time major releases occur for the Yocto Project. The Yocto Project Version used for this work is 3.1, codenamed *Dunfell*, and the Poky version is 23.0. *Dunfell* was released in April 2020, and it will have long term support provided by Yocto Project at least for 3 years after which it will move to be community managed. The Linux kernel used for *Dunfell* is Yocto's *linux-yocto* kernel version 5.4.

The OpenEmbedded Project is a build automation framework consisting of BitBake build engine and other tools, which are used to create Linux distributions which Yocto Project adopted as a default build system in March 2011. The project also hosts a layer index with a searchable database of layers, recipes and machines [4].

## 2.6 The Yocto Project Layer Model

The Yocto Project uses a "Layer Model" type of development model which distinguishes it from other simple build systems [3]. These layers provide possibility to logically separate different parts of the system as reusable components such as Board Support Package (BSP), Graphical User Interface (GUI), Operating System (OS) configuration, middleware and application layers. Use of the Yocto is also made easier by silicon vendors such as Seco, Intel, Karo and Congatec by providing BSP layers for their products. This layer model makes it possible to do development in any layer regardless of the other layers which means that for example GUI development can be done at the same time as kernel development.

For example, at Teleste DCU40 has normally been equipped with a Debian "base image" with required hardware packages before Java developers can write applications for it,

and after the "base image" has been selected, modifying it can be difficult afterwards. Normally the only option has been to overwrite files of the "base image". The Layer Model introduces a solution to this by enabling the modifiability of the full software stack, from kernel configurations to individual software packages. Distinct layers also make it faster to transition from older hardware to newer hardware or port already implemented application layers for completely new hardware since layers such as the BSP layer and Teleste's platform component layer are separated and reusable.

## 2.7 BitBake recipes

BitBake recipes, which are denoted by the filename extension `.bb`, are the basic metadata files used by the BitBake. They provide following information and instructions:

- Descriptive information about the recipe such as version, author, homepage and license
- Build and runtime dependencies
- Location of source code and how to fetch it
- Possible patches to the source code and how to apply them
- How to configure and compile the source code
- How and where to install the generated build products

BitBake recipes support sophisticated variable syntax with hard, default and weak assignments and also appending, prepending and removal of values within variables. Variables can also be assigned with flags. Even more recipe features are provided with Python variable and function support, together with basic Shell functions. Python and Shell functions can access the same variables.

During BitBake runs, the recipes are parsed and syntax checked, which reduces the possibility of errors in the recipes. Poky reference distribution and OpenEmbedded layers provide good examples of complex recipes which often use more supported features such as the inheritance of class files (`.bbclass`), which are shared amongst multiple recipes, splitting of recipes to smaller parts as include files (`.inc`) and modifying existing recipes with `.bbappend` files. During this work, 70 new recipes were written for DCU40 and Teleste's platform components.

## 2.8 BitBake build configuration files

Most of the configuration files are denoted by the `.conf` filename extension, which define various types of configurations such as:

- `local.conf`, which resides in the build directory's `conf` folder. It is the most important configuration file for each build, and it describes packets, features, Yocto's built-in variables and many other features. The same `local.conf` can be used for different

machine targets such as DCU40 or Qemu build. Commenting the configurations in this file with a proper table of contents section makes it clear to read and edit afterwards. `local.conf` for DCU40 has 4 sections:

- Device related configurations, which define features and software highly related to DCU40 and its hardware
  - Core features, which are essential but not mandatory for DCU40, such as package management, SSH server, Java Runtime Environment, PostgreSQL and Lighttpd
  - Software such as Teleste's platform components, useful utilities, demos and testing software
  - Other settings related to the BitBake environment, cache locations, debugging tweaks and Qemu settings
- `site.conf`, which defines the locations of cache, downloads and univariate tarball
  - `layer.conf`, which has all the included meta layers for the build
  - `<machine>.conf`, like "intel-corei7-64.conf" is machine configuration file with architectural configuration, preferred library packages, preferred kernel provider and version.

## 2.9 Recipe classes

Class files, denoted by the `.bbclass` filename extension, contain inheritable instructions shareable among recipes. Poky reference distribution directory contains over 200 classes and some examples of them are:

- `base.bbclass`, which is always included for all recipes and classes and it has definitions for basic tasks such as fetching, unpacking, compiling and so on
- `image_types.bbclass`, which has all the image types BitBake can build and also descriptions for other build artifacts such as checksum files
- `qemuboot.bbclass`, which generates `qemuboot.conf` for `runqemu` Qemu wrapper
- `pypi.bbclass`, which automates installation of Python packages
- `package_deb.bbclass`, which automates Debian package creation, other package formats such as RPM and IPK are also supported with corresponding class files
- `useradd.bbclass`, which makes adding users easy
- `update-rc.d.bbclass`, for enabling init scripts
- `autotools.bbclass`, which automates Makefile creation, compiling and installation of supported source code collections

Usage of classes in recipes is not forced in any way, but with them complex yet clean recipes can be written. Recipes found within Poky reference distribution provide good examples of class inheritance within recipes.



## 2.10 Recipe append files

Append files with filename extension `.bbappend` modify or override existing recipe files. An append file must have a corresponding recipe file, and they must share the same base filename. The filenames can differ from the used suffix, which usually is the version number of the recipe. For example, the file `linux_yocto_%.bbappend` can have kernel configuration fragment information for all `linux_yocto` kernels. This way, the kernel version can be updated without losing the previous kernel configurations. Replacing the `"_%"` wildcard with a version number such as `"_5.4"` would apply this append file for only kernel version 5.4.

Append files are the recommended way of editing existing recipes. In this work, some of the use cases for them are:

- Overwriting default `php.ini` configuration file for PHP
- Patching a known bitmask bug in a Python serial port package
- Initializing Teleste's default database for PostgreSQL
- Enabling full support for system requests in the `procps` package
- Enabling Serial Getty only in virtualized environments, so that it does not block the hardware serial port of physical DCU40
- Modifying the installation script of live bootable USB image so that it does not require user interaction

Append files can also be disabled easily by adding a `BBMASK` configuration to `local.conf`.

## 2.11 BitBake tasks

While BitBake is a task execution and build engine, it does not actually build the source code, but it generates a `do_compile` task by using the recipe for that specific source code. In the recipe there is a `do_compile` function declaration, which uses BitBake's environmental variables to run e.g. `make` for the source code. However, many existing recipes, such as the recipe for Bash from OpenEmbedded-Core, are written in a way that just inheriting `autotools.bbclass` provides the `do_compile` function declaration, so that it is not declared in the recipe file itself. BitBake gathers and writes a task file, which is a Shell executable, under `<build_directory>/tmp/work/temp` which has all the variables expanded and which will be run by BitBake. The `do_compile` task and other tasks can also be run manually (without running dependent tasks) with BitBake's `devshell` task:

```
$ bitbake <recipe> -c devshell
$ ../temp/run.do_compile
```

This will show the output of the commands run in the task in `stdout`, which makes it easy to debug `make` errors, for example. It is to be noted that normally the compilation task

for Bash is nearly never run since its source code is not edited between the builds so it is just fetched automatically from the shared state cache. BitBake also creates logs for each individual task and the log file `log.task_order` shows all the executed tasks for the recipe. If the task `devshell` is replaced with the `do_compile` task in the bitbake command, then BitBake will also execute the dependent pretasks for compiling, such as source code fetching, patching and configuring.

## 2.12 The Yocto Project Extensible SDK

The Yocto Project Extensible SDK, or eSDK in short, is a collection of development tools which are part of the Openembedded Core. The eSDK can be used to add applications and libraries to the build, modify the source code of existing components, test changes on the target hardware, and also develop and test code that is designed to be run on the target system. One of the most used tools of the eSDK in this work is *devtool*, which can assist in adding new software to the build and editing existing sources. For example the source code of any recipe can be initialized in development environment with a command:

```
$ devtool modify <recipe>
```

This copies the source code from the `tmp` directory, which is inside of the build target directory to a `workspace` directory. The source code is set up as a Git repository, so any changes to the source files will be tracked by Git. If the source code is edited and BitBake build is called, then the edited source code will be used. To save the modified source code it must be committed in the Git repository and then `devtool` can be used to create a `.bbappend` file, with a patch to store the changes to some meta-layer with the `devtool`:

```
$ devtool update-recipe -a <layerpath> <recipename>
```

After this, the build source can be switched back to the original `tmp` directory by resetting the workspace target with a `devtool` command:

```
$ devtool reset <recipename>
```

## 2.13 OpenEmbedded Image Creator Wic

Partitioning in this environment can be done in two different ways: by OpenEmbedded Image Creator Wic, which is based on Kickstart partitioning commands from Fedora [5], or with live USB image's installation script as in Section 4.17. Wic is suitable for creating multiple partitions in a single image file which has a `.wic` filename extension, and it can be directly flashed to the target device or to a storage medium.

A live USB image was created with Wic, which has all the hardware dependencies of DCU40 so it can test the device and for example update the AVR or the BIOS. The partition file is located at:

```
yocto/meta-layers/local/meta-dcu40/wic/dcu40-tool.wks
```

and it has the following content:

```

part /boot --source bootimg-biosplusefi --sourceparams="loader=systemd-boot"
  --label boot --active --align 1024 --use-uuid
part / --source rootfs --fstype=ext4
  --label rootfs --align 1024 --use-uuid
part /mnt/usb-storage --fstype=vfat --size 128
  --label usb-storage --align 1024 --use-uuid
bootloader --ptable gpt --timeout=5
  --append="rootwait rootfstype=ext4 console=ttyS0,115200n8"

```

where the first bootloader partition for legacy and UEFI is created with a source plugin: `yocto/poky/scripts/lib/wic/plugins/source/bootimg-biosplusefi.py`. This automates the partition creation, and the same is done for the root filesystem partition. The usb storage partition is a FAT32 partition, which is accessible on a Windows host so it can be used for logging purposes or even scripts and updates, which could be run when the live USB flash drive would be plugged into the device. The last bootloader line defines the configuration for the bootloader and appends to the kernel command line.

## 2.14 Toaster web interface

Toaster is a web interface for OpenEmbedded and BitBake. It allows configuring and running builds, and it provides information and statistics about the build process. Some of its features are [6]:

- Inspect builds, what packages they have and what tasks were run
- Browse the root filesystem of the image
- See the values of all build variables and which files set them
- Debug errors, warnings and trace messages
- See dependency relationships between recipes, packages and tasks
- See performance statistics of the build
- Start builds
- Modify build variables and layers
- Browse and build any layers in the OpenEmbedded Metadata Index

While many of these tasks can be done on the command line interface, Toaster provides a fast and intuitive way of inspecting the builds. Toaster can also be set up as a shared hosted service, which is suitable for multiple users developing across many build hosts. To enable Toaster for BitBake builds, the BitBake environment has to be initialized first and then the Toaster can be sourced with a command:

```
$ source toaster start
```

which starts a local Toaster instance on a Django server.

## 3 USING THE BUILD ENVIRONMENT

### 3.1 Setting up the project as a Git repository

The build environment of this work was set up as a Git superproject, which makes managing Yocto's Poky reference distribution and remote meta-layers easy, as they can be added as Git submodules. Git submodules make it possible to add other Git repositories to the superproject while keeping commits between them separated [7]. Commits of the superproject track commits in the submodules so that if the superproject is pulled from a remote Git repository, it will checkout correct commits for the submodules. This causes submodules Git status to be "detached HEAD", which means that the submodule is not anymore attached to any branch but only to a single commit. This ensures that the cloned superproject with submodules is exactly same across all clients. The command to pull a Git repository with submodules is:

```
$ git clone -b <branch> --recurse-submodules <repository_url>
```

This does not mean, that the submodules would be locked in that specific branchless commit though. A command can be used to run a Git command for all submodules, for example to checkout a specific branch for all:

```
$ git submodule foreach 'git checkout -b <branch>'
```

Checking out branch *dunfell* for Poky and submodule layers conveniently updates them to the newest Dunfell releases. Having the Poky and others as Git submodules under the superproject makes it convenient to debug them by editing their files and after no more debugging is needed, the changes in the submodules can be reverted back to the original by checking out the modified files thus reverting the made changes.

### 3.2 Building for DCU40

The build host requires some dependency packages to be installed, which are listed in Chapter 1.2 of the Yocto's reference manual [5]. This chapter also lists the tested build host Linux distributions and it is mentioned that the Yocto Project could be also set up on version 2 of Windows Subsystem for Linux. Instructions are also given on how to deploy the Yocto Project on a Docker container.

After the dependencies have been installed, the build environment for DCU40 can be

initialized by going to a directory:

```
yocto/build/dcu40-image/
```

and running the OpenEmbedded's build environment initialization script:

```
$ source ../../poky/oe-init-build-environment .
```

If some other directory is given as an argument for the initialization script, a new build directory will be generated with default configuration files, which can be used to set up new build targets. No other initialization steps should be needed if the build is run in Teleste's network, since `site.conf` should be pointing to the right network share, where the downloads and shared state cache are located for fast rebuilds.

After the build environment has been initialized the build process for DCU40 can be started with a command:

```
$ bitbake dcu40-image
```

After starting the BitBake it starts to go through the recipes and variables defined in:

```
dcu40-image/conf/local.conf
```

and finds the recipes from the layers defined in:

```
dcu40-image/conf/layers.conf
```

finally generating build artifacts to a directory:

```
dcu40-image/tmp/deploy/images/intel-corei7-64/
```

The initial build time should be around 10 minutes on Teleste's company laptop (tested on Lenovo T490 running Debian 10), when using the download and sstate cache network share.

### 3.3 Cache management

Yocto's recipes have version dependent checksums for every source archive which are downloaded from the internet, and if the BitBake build system notices a mismatch between a checksum in a versioned recipe file and cached source archive, the source archive is marked as corrupted by adding suffix `_bad-checksum` to the filename of the archive after which BitBake tries to download it again from the original source address. By default BitBake stores the downloaded source code archives to the build configuration directory under a sub directory `downloads`. This cache is useful if the source code repository would be unreachable in the future, or the connection to it would have a low bandwidth.

Moreover, BitBake detects tasks that do not need to be rerun which can be saved to a *"Shared State Cache"* [1]. By default, this cache is stored in the build directory under `sstate-cache` subdirectory. This cache stores checksums equipped intermediate build artifacts produced by recipe tasks such as compiling, packaging, patching and generating root filesystem. Using this cache is essential for fast rebuilds since rerunning tasks for

recipes that have not been "touched" can be fetched from the cache instead. This cache decreases the build times of DCU40 images from tens of hours, to just around 10 minutes.

An automation script was written for copying the cache items from a local build host to the Teleste's network share so that they would be usable by other build hosts. The script uses the `rsync` file transfer program to compare the network share directory with the local cache directories so that it copies only the new files. BitBake can also create symbolic links inside of the `downloads` and `sstate-cache` directories to the cache items located in the Teleste's file share, if the file share is mounted as a directory to the build host. For BitBake to be able to do this, Teleste's file share mount directory it must be defined in the sources configuration file as a source address:

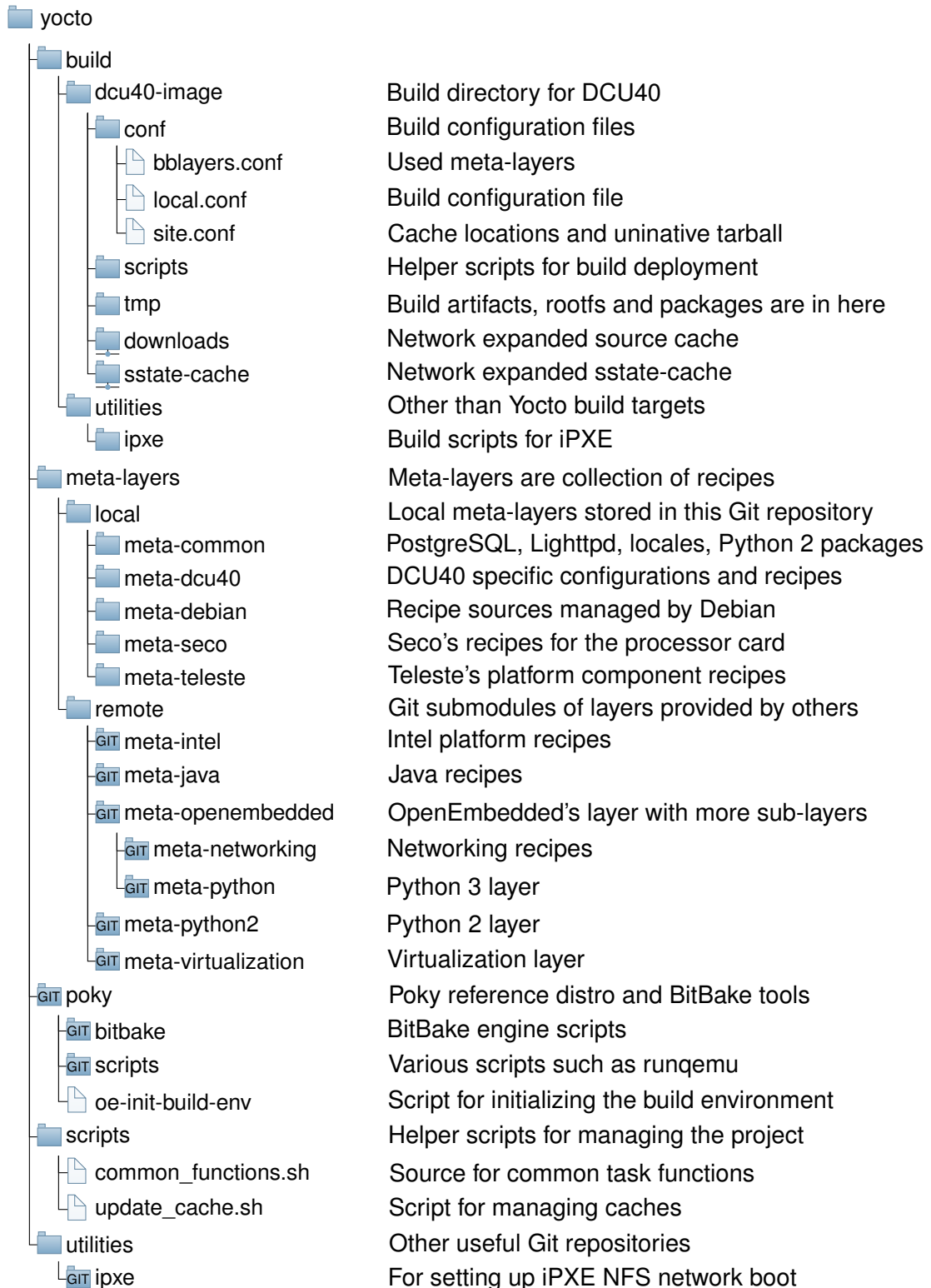
```
yocto/build/dcu40-image/conf/site.conf
```

If Teleste's file share is not found as a mounted directory, then the cache items are downloaded over HTTP protocol from the same Teleste's file share. The drawback of this is, that then the local build cache will be bigger in size, compared to if it would just have symbolic links to the needed cache files. In the end of this work, `downloads` cache was 40 GB and `sstate-cache` 80 GB in size on Teleste's network share.

It was noted during this work, that a single "worker" partition on a hard drive started to have corrupted files more and more after many build iterations. This was when the cache was stored under the build configuration directory, which also has the `tmp` directory for the actual build process. Moving the local `downloads` and `sstate` cache to a software RAID cache set up with `mdadm` utility on two SSDs solved the file corruption problems and also made subsequent builds faster because of increased read and write speeds. Majority of the actual work done during rebuilding is just comparing checksums to intermediate build artifacts, which are in the `sstate` cache and if the checksum matches those artifact archives are unpacked to the `tmp` directory.

### 3.4 Directory structure of the project environment

A simplified directory structure of the project Git environment can be seen in Figure 3.1, where the root folder is named `yocto`:



**Figure 3.1.** Directory structure of the Git project

Directory `build` has subdirectories for different build targets such as `dcu40-image`. Names for the build target directories are same as the build targets BitBake is called with, so it is easy to know what build target can be called in different directories. This arrangement is not forced in any way by the Yocto environment, and many build targets can be called from the same directory. The build directory also includes a subdirectory for other useful utilities such as `iPXE`.

Directory `meta-layers` has sub directories for local and remote meta-layers. Local meta layers are versioned by the Git superproject, and remote meta layers are fetched from other Git repositories and added as submodules. In the Figure 3.1 they are marked with the text "GIT" over the folders. Directory `poky` contains the reference distribution of the Yocto Project and the BitBake build tool. It could be considered as a remote meta layer, but having it separate does not cause any problems since the only place its location is referenced is in the builds configuration file `site.conf`.

### 3.5 Git management guidelines

Using Git in a company environment to manage source code of a product requires a set of guidelines on how to use the feature set of Git, in a manner which keeps the repository clean and uniform. Also, similarity between the Git repositories of company's products makes it easier to work with repositories maintained by other teams. Since Subversion has been Teleste's main VCS for many years, and Git has mainly been used for the newest projects and products, the best practises for using Git in versioning this build environment should follow some example. Because of this, a set of guidelines is proposed which are following Atlassian's Git tutorials [8] for Bitbucket, which is the Git software used at Teleste.

In the future, for this build environment Git branches should be created for five different purposes:

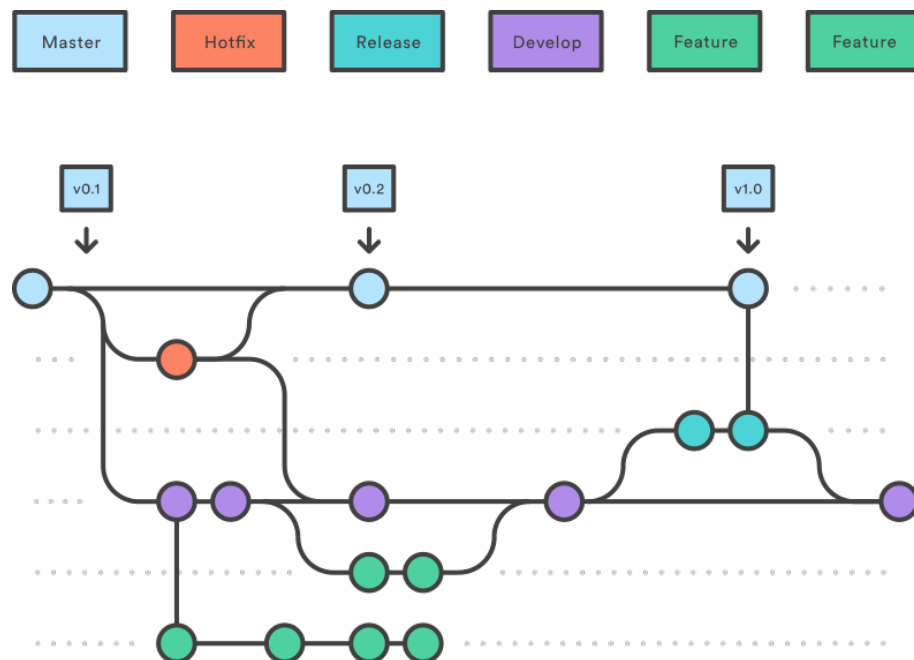
- Master branch for tagged releases of the environment. Merges to master branch should mostly be done from develop branch but changes from hotfix branches would also be accepted. Any tagged release of the master branch should be able to run all the build targets it consists of which could be automatically verified by a CI server such as Jenkins, which would be also handling the automatic tagging of the master branch.
- Develop branch for gathering the changes made in feature and hotfix branches. Any new feature should branch off from the develop branch and back to it. Also, a release should be branched off from the develop branch, merged in to the master branch and then back to the develop branch.
- Feature branches for actual feature development for the environment, which would eventually be merged to the develop branch. A nice practise at Teleste has been to create feature branches named like "feature/<Jira task ID><Jira task title>" where Jira by Atlassian is the task and project management software used by Teleste.



Specifying the tasks needed to be done to the environment before creating feature branches keeps the environment development process documented.

- Release branch for major releases of the environment, which are branched off from the develop branch and merged to the master branch preferably tagged with a major release number without a minor number (e.g. version 2.0). These major releases should definitely be able to run all the build targets they consist of, and the metadata and configurations in them should be written in a clean manner.
- Hotfix branches are branched off and to the master branch. A ready hotfix should be merged into both master and develop branches.

The Figure 3.2 by Atlassian illustrates these five different branches and how they interact with each other. During the initial development of this environment, these branches were not used since, for the first official release of the environment, Git rebase can be used to merge individual commits to a single one for clean initial release.



*Figure 3.2. Git workflow illustrated by Atlassian*

### 3.6 Bash and Shell scrips

Bash is an Unix shell and command language, which can used for simple automation scrips, but also for writing more complex software-like programs. It does not offer proper features for writing advanced code such as inheritable classes found in higher level languages, but many complex and elegant structures can still be written with it. One of the main problems of complex Bash scrips is, that the declared variables are global by default and usable in every subsequent scrip called and sourced by the main scrip. It does have support for functions but for example these functions' only return value is the status of the last statement executed in the function. To return any arbitrary value, the function must

declare a global variable or modify one which can be inspected in another statement.

Bash script collections can easily become hard to maintain when many people have written them, because similar functionality can be archived in multiple of ways. Without proper documentation and guidelines, scripts written in Bash by different people tend to differ from each other, and the elegance of implementations is highly proportional to the previous skill about Bash and overall Unix shells. Also, skills with other languages such as C and Java can affect the nature of code implementations when written in Bash.

In this work, Bash scripts were written for various automation tasks such as USB drive flashing, updating the cache, and some of Imagetools' Bash implementations were mimicked to make it easier for getting started with this build environment. New features were implemented, such as automated logging of calls to external programs with an execution wrapper, and a clean structure of scripts, where the main function calls subfunctions for initialization, argument parsing and sequential statement execution. Eventually these automation tasks could be integrated as task recipes for the BitBake so that the environment would stay as consistent as possible. Since BitBake supports Shell and Python function inside same tasks, and it has advanced debugging capabilities, the re-implementations of Imagetools' features could be done in elegant and robust way.

## 4 CONFIGURATIONS FOR THE DCU40

### 4.1 Target hardware device

In this work, the target hardware device is Teleste's embedded device *DCU-40 Diagnostic and Controller plug-in Unit, series-40* which is referred as DCU40. This rack unit is the main server in Teleste's Passenger Information Systems (PIS) in railway vehicles. DCU40 is equipped with Seco's pITX processor card, which is based on Intel Atom Q7 Bay Trail SoC. It has following technical specifications:

- Processor: Intel® Atom™ E3845 Quad Core 1.91 GHz
- Memory: 4 GB DDR3L
- Storage: 16 GB eMMC mass memory, SD card socket
- Interfaces: Gigabit Ethernet, I/O port, GNSS and 2 WWAN antenna ports, 2 SIM-card sockets, USB port, DisplayPort
- Features: LTE modem, AVR microcontroller, FPGA IC

The motherboard of the DCU40 is designed at Teleste and it has developed a lot since its first release in 2011. This device is responsible in controlling PIS devices such as LED screens, TFT screens, announcements, emergency phones, video surveillance system, route tracking, and many other other systems. DCU40 supports running Windows and Linux operating systems. Normally, if a Linux OS is chosen for the DCU40, the distribution is Debian. The process of creating a Debian image with Teleste's Imagetools for the DCU40 is explained in Chapter 6.1. DCU40 and its front interfaces can be seen in the Figure 4.1.



*Figure 4.1. DCU-40 Diagnostic and Controller plug-in Unit.*

## 4.2 Introduction to dcu40-image

**dcu40-image** is the main BitBake build target of this build environment, with hardware support for DCU40 together with virtual image creation. It is based on Seco's BSP guide's [9] few instructions from which it is expanded with additional layers, recipes, and configuration settings. The directory for DCU40 builds is located at:

```
yocto/build/dcu40-image/
```

In this directory, there is a subdirectory `conf/` with the main configuration file `local.conf`, which defines settings, package recipes and BitBake variables for the build. Other configuration files in `conf/` directory are `bblayers.conf`, which defines meta-layers used in this specific build and `site.conf` for downloads and shared state cache mirror addresses. In the build directory there is also a `scripts` subdirectory which has automation scripts for network boot artifact deployment and live USB flash drive creation.

## 4.3 Target machine configuration

Even though DCU40 uses Seco's processor card with Intel Atom E3800 family processor, the machine configurations for it come from Intel's general configuration set named `intel-corei7-64`. This is because Seco's BSP guide [9] also uses this configuration for

BitBake's `MACHINE` variable which specifies the target machine the image is built for. The `intel-corei7-64` configuration comes from a meta-intel layer, which *"supports moderately wide range of drivers that should boot and be usable on "typical" hardware."* according to its description. It is located at:

```
yocto/meta-layers/remote/meta-intel/conf/machine/intel-corei7-64.conf
```

This target machine is configured in the main configuration file `local.conf` with a line:

```
MACHINE = "intel-corei7-64"
```

This configuration includes other configurations from Intel's meta-layer, but also from Poky's machine configurations. Some of the configurations are:

- Basic machine features such as x86 architecture, PCI bus, ACPI, USB and EFI
- Preferred kernel provider, version and packaging type
- Graphics and X server
- Binary file tuning options
- Intel's microcode updates

## 4.4 Distribution configuration

While Poky is Yocto Project's reference distribution, and the default selection for the builds, a new distribution called Sky Blue was created for the DCU40 to emphasize this custom build environment tailored for Teleste's use cases. The distribution configuration is selected in the `local.conf` with:

```
DISTRO = "sky_blue"
```

and the actual configuration file is located at:

```
yocto/meta-layers/meta-teleste/conf/distro/sky_blue.conf
```

It defines at least the following settings and features for the distribution:

- Name, version and maintainer
- Default distribution features
- Include of `packagegroup-core-boot` for a minimal set of packages to boot the system
- Yocto Project Software Development Kit settings
- Mirror addresses of source code repositories hosted by The Yocto Project
- List of tested build host distributions
- Build error management
- Security related build options

In the future, more configurations can be added to the distribution such as Teleste's platform components. Figure 4.2 is Sky Blue's logo.



*Figure 4.2. Teleste Sky Blue logo*

## 4.5 Systemd as init manager

By default Yocto uses SysVinit as init manager but this can be changed to SystemD by adding following configurations to the `local.conf`:

```
DISTRO_FEATURES += "systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
VIRTUAL-RUNTIME_initscripts = "systemd-compat-units"
VIRTUAL-RUNTIME_syslog = "rsyslog"
VIRTUAL-RUNTIME_login_manager = "shadow-base"
DISTRO_FEATURES_BACKFILL_CONSIDERED = "sysvinit"
```

Even though there has been a debate between SysVinit and SystemD as init system for distributions such as Debian, Fedora and OpenSuSE, SystemD has replaced SysVinit in these distributions [10]. SystemD has also been the default init system for Debian since *Jessie*, which was initially released in 2015.

To achieve backwards compatibility to Debian based DCU40 image, SystemD was chosen as the init system for this Yocto build. However SystemD has support for SysVinit type init scripts, which were also enabled.

## 4.6 Seco packages and kernel modules

Seco provided packages and kernel modules for SPI, I<sup>2</sup>C and for internal components such as:

- FXL6408 and PCAL6408A, 8-bit I<sup>2</sup>C-controlled GPIO expanders
- PCA9655E, a 16-bit I<sup>2</sup>C-controlled GPIO expander

They also provided kernel module `seco-spi`, which is an implementation for interfacing with SPI devices from user space via the `spidev` linux kernel driver. Rest of the packages are:

- `minicom`, a tool used to connect serial devices
- `setserial`, a program for getting and setting Linux serial port information
- `net-tools`, a collection of base networking utilities

- spitools, a command line tool to help using spidev devices

These configurations are set in the `local.conf` with lines:

```
IMAGE_INSTALL_append = " seco-spi i2c-tools fxl6408 pcal6408 pca9655e \
                        minicom setserial net-tools spitools"
MACHINE_EXTRA_RRECOMMENDS += "kernel-module-seco-spi"
KERNEL_MODULE_AUTOLOAD += "spidev seco-spi"
```

## 4.7 Package manager and Debian package format

The Yocto Project's default package manager is RPM, which was originally made for Red Hat Linux. Since Debian package format is the default for Teleste's packages, and Imagetools created base image it, was chosen for the DCU40 image. Switching to Debian packages requires these configuration lines in the `local.conf`:

```
EXTRA_IMAGE_FEATURES += "package-management"
PACKAGE_CLASSES = "package_deb"
```

This enables the `dpkg` package manager, but it does not necessarily provide support for packages build especially for Debian. The OpenEmbedded Core layer also has a recipe for `apt`, which is an advanced front-end for `dpkg`. This can be used together with Teleste's own Debian package repository to install packages to DCU40 when developing projects.

Internally, BitBake will package all software as individual Debian packages, which are then installed to the filesystem. BitBake also splits the packages into smaller parts, such as source code and documentation which can be included, if needed, on the root filesystem with a `local.conf` line:

```
EXTRA_IMAGE_FEATURES += "src-pkgs doc-pkgs"
```

## 4.8 Keyboard layout

By default, the keyboard layout of the built image is US layout. To use Finnish keyboard layout, the `keyboard-fi` package from `meta-dcu40` layer is added to `local.conf`:

```
IMAGE_INSTALL_append = " keyboard-fi"
```

However this package does not provide keyboard support for Scandinavian letters such as "ä" and "ö". To enable full Finnish support following configurations can be added to the `local.conf`:

```
GLIBC_GENERATE_LOCALES = "en_US.UTF-8 fi_FI.UTF-8"
IMAGE_LINGUAS = "fi-fi"
```

These install the Finnish locales and set it as the default language for the image. However, this should not be necessary for Finnish, nor other languages, since customer projects' software does not need this kind of system wide support for multiple languages.

## 4.9 Systemd-boot or GNU GRUB as bootloader

The default bootloader for Yocto is systemd-boot, which is lighter and simpler than GNU GRUB. Systemd-boot supports only systems with UEFI firmware, and it loads boot entry information from the UEFI system partition which is usually mounted at `/efi/`. Systemd-boot makes it possible to change boot configurations such as timeout, default boot entry selection, kernel command line arguments, and others. It also integrates with systemd which implements features such as rebooting into a specific boot entry. To increase the reliability of the system, systemd implements boot counting and an automatic fallback to a working boot entry if enough failures are encountered [11]. The kernel must be configured with `EFISTUB` enabled for the systemd-boot.

GNU GRUB is more traditional bootloader for Linux systems, and now it refers to its second version while the first version is called Grub Legacy. GRUB 2 was rewritten from scratch since the first version could not keep up with the feature requirements and extensions written for it [12]. GNU GRUB offers more features than systemd-boot but that also makes it more complicated. However, it can also boot from MBR partitions which enables usage of Legacy BIOS, but it should not be considered as a viable option anymore since Intel is planning to remove support for CSM from client and data center platforms by 2020 which enables Legacy booting [13].

Systemd-boot is selected as bootloader in the `local.conf` with:

```
EFI_PROVIDER="systemd-boot"
```

## 4.10 BusyBox and GNU Core Utilities

By default, Yocto uses BusyBox to provide several Unix utilities in a single executable file. Even though BusyBox offers nearly all utilities provided in GNU coreutils, they are normally minimalist versions of their fully-featured GNU counterparts thus having fewer options. For embedded linux system with limited resources and size limits, BusyBox can be an optimal solution. However, DCU40 is equipped with enough memory and processing power to run the full versions of GNU coreutils, and early on it was noticed that even the simplest programs such as `ps` for listing processes did not have all the options which the GNU coreutils has. Because of this, BusyBox was replaced with GNU coreutils by adding following configuration lines to the `local.conf`:

```
PREFERRED_PROVIDER_virtual/base-utils = "coreutils"
VIRTUAL-RUNTIME_base-utils = "coreutils"
VIRTUAL-RUNTIME_base-utils-hwclock = "util-linux-hwclock"
```

## 4.11 Kernel configurations

The default kernel provider in Yocto is `linux-yocto`, which can be found at:  
`yocto/poky/meta/recipes-kernel/linux/linux-yocto_5.4.bb`



Another kernel is provided by meta-intel layer:

```
remote/meta-intel/recipes-kernel/linux/linux-intel_5.4.bb
```

To select linux-yocto kernel version 5.4.51, following configuration lines are added to the `local.conf`:

```
PREFERRED_PROVIDER_virtual/kernel="linux-yocto"
PREFERRED_VERSION_linux-yocto="5.4.51"
```

To append arguments to the kernel command line, the following line is added to the `local.conf`:

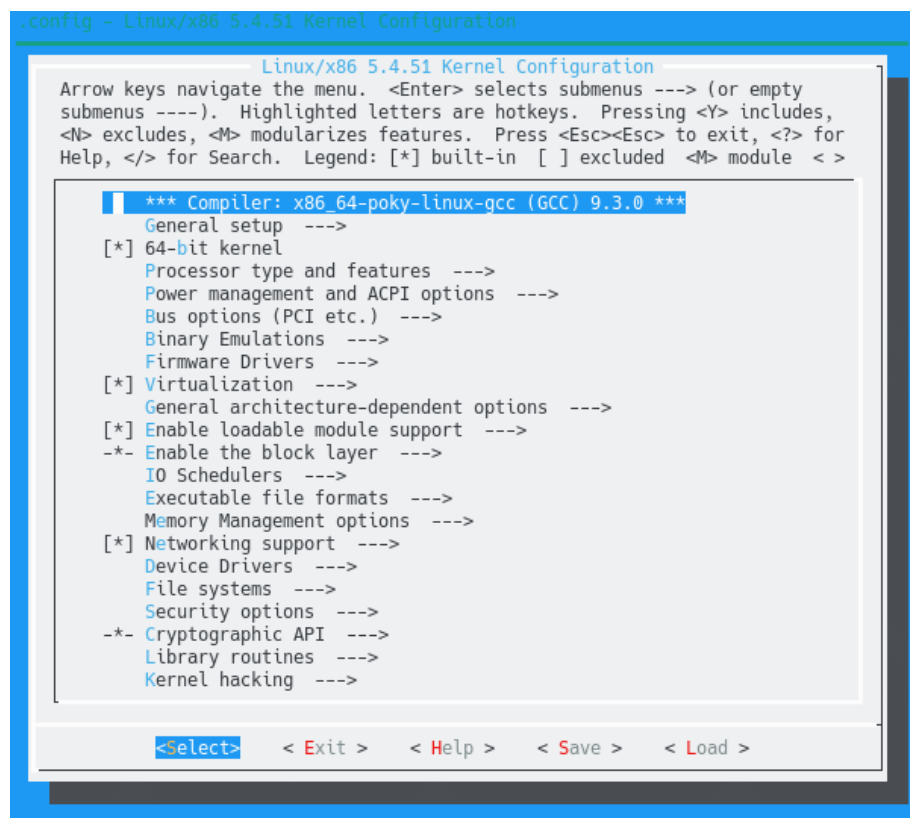
```
APPEND += "net.ifnames=0 mitigations=off"
```

which enables traditional network interface names and disables security mitigations discussed in Section 7.2.2.

Selected kernel can be configured with a graphical menuconfig interface with a command:

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

The menuconfig interface can be seen in the Figure 4.3.



**Figure 4.3.** Main menu of Linux kernel configuration menu.

Instead of creating a full kernel configuration file, the kernel can be configured with configuration fragments [14]. These are more flexible since they can be applied to other kernel versions, also. However, by using the search option in the Kernel configuration menu, the kernel configuration options are easy to find. Once the correct options have been found, they can be added to `meta-dcu40` as a kernel recipe, which will get

appended to linux-yocto kernel. The patch directory is located at:

```
yocto/meta-layers/local/meta-dcu40/recipes-kernel/linux-yocto
```

For example, there are separate kernel configuration fragments for GNSS, I<sup>2</sup>C, NFS and SPI.

### 4.11.1 Analyzing the build size of individual kernel features

When doing kernel build iterations for storage space limited devices, the size of the kernel binary can be an important factor. To inspect the build size of individual kernel features, the kernel has to be rebuilt without the sstate cache so that the build generates actual build artifacts rather than using previously build sstate cache items. The command to rebuild the kernel without sstate cache is:

```
$ bitbake linux-yocto -c do_build -f
```

The sizes of individual kernel build artifacts can be quickly checked with a command:

```
$ pushd `find ./tmp/work/ -regex ".*linux-yocto.*standard-build"` && \
  find . -name "*.o" | xargs size | sort -n -r -k 4 | less && popd
```

This command finds the build directory, temporally moves into it, finds all compiled object files, gets the sizes of them and sorts them in descending order in less, after which it moves back to the original directory.

## 4.12 Disabling serial-getty

The machine configuration in:

```
yocto/meta-layers/remote/meta-intel/conf/machine/intel-corei7-64.conf
```

sets serial devices ttyS0, ttyS1 and ttyS2 as serial consoles and creates a systemd target for the build rootfs:

```
/etc/systemd/system/getty.target.wants
```

This causes systemd to start serial-getty processes on those serial ports which floods DCU40's ttyS0 serial port. This port is used for the UART connection between the ATmega 3250A AVR microcontroller and Seco's processor card. If ttyS0 is used by serial-getty processes, the connection between user space programs and AVR becomes unreliable. However, serial-getty is needed when running the build with a command `runqemu` (see Section 5.2), which is why the serial consoles are not completely disabled with a `local.conf` line:

```
SERIAL_CONSOLES_intel-corei7-64 = ""
```

Since systemd offers many condition options for starting the services, it can be checked, if the system is executed in a virtualized environment [15]. For the serial-getty service this is done by adding a configuration line:

```
ConditionVirtualization=yes
```

to the unit configuration of `serial-getty@.service`. This configuration calls for command

`systemd-detect-virt`, which returns `none/qemu` depending on the virtualisation status of the environment the build is running on. A patch was created for the `systemd` to enable this modification.

### 4.13 Enabling magic SysRq key

The Magic System Request Key provides keyboard key combinations to directly send commands to the kernel [16]. These can be useful in developing and debugging situations, for example, if the target machine becomes unresponsive. To perform a safe reboot of a linux system, the following keyboard combination can be used:

Hold down: `Ctrl` + `Alt` + `SysRq`

and type these keys in this order:

- `r`, turns off keyboard raw mode and thus recovers from X server crashes
- `e`, sends a SIGTERM to all processes, except for `init`
- `i`, sends a SIGKILL to all processes, except for `init`
- `s`, attempts to sync all mounted filesystems
- `u`, attempts to remount all mounted filesystems read-only
- `b`, immediately reboots the system

Magic SysRq key was enabled with `bbappend` file located at:

```
layer/meta-dcu40/recipes-extended/procps/procps_%.bbappend
```

It patches the file `/etc/sysctl.conf` which is provided by the `procps` package. System requests can be disabled by adding a `BBMASK` configuration to the `local.conf`:

```
BBMASK += "procps.*\bbappend"
```

### 4.14 Java 8

Java 8 is the default Java on DCU40 and it can be added to the image with following configuration lines to the `local.conf`:

```
PREFERRED_PROVIDER_virtual/java-initial-native = "cacao-initial-native"
PREFERRED_PROVIDER_virtual/java-initial = "cacao-initial"
PREFERRED_PROVIDER_virtual/java-native = "cacao-native"
IMAGE_INSTALL_append = " openjre-8 libslf4j-java"
```

The first three configuration lines enable CACAO Java Virtual Machine, and the last line is the Java runtime library by OpenJDK. A few logging libraries, such as SLF4J, were added by writing a recipe for getting the required Java archive files from Maven's central archive which corresponded with the same library version as on Teleste's Debian based DCU40 base image.

## 4.15 Python 2

Even though Python 2 support by the Python Software Foundation ended in January 2020, and it is not receiving any security updates [17], it still has its use cases in running legacy software which has not been ported to Python 3 yet. Adding Python 2 can be done with OpenEmbedded community layer meta-python2, which has been added to:

```
yocto/meta-layers/remote/meta-python2
```

Python 2 and required packages are installed with `local.conf` lines:

```
IMAGE_INSTALL_append = " python python-pip python-spidev \  
                        python-cryptography python-serial \  
                        python-gps python-paramiko python-cffi"  
PREFERRED_VERSION_python-serial = "2.7"
```

Acquiring the correct Python 2 packages for pip requires writing recipes for them which are in:

```
yocto/meta-layers/local/meta-common/recipes-python/
```

While a newer version of python-serial exists (3.4), version 2.7 is used which is the same as in Teleste's Debian based DCU40 base image. This version has a known bug related to a certain bitmask so a `.bbappend` patch file was written for it.

## 4.16 PostgreSQL 12

DCU40 uses PostgreSQL on Debian so it was added to the image from meta-openembedded layer with `local.conf` lines:

```
IMAGE_INSTALL_append = " postgresql postgresql-client libpam \  
                        postgresql-dev python-pygresql"  
DISTRO_FEATURES_append = " pam"
```

Since the DCU40's Debian base image has a preinstalled SQL database, the same database was also set up with this build environment. Initializing of the database can be done during BitBake image generation by extracting a pre-generated database archive to a directory:

```
/var/lib/postgresql/data/
```

on the target filesystem. To disable the installation of the default database a mask for `.bbappend` file installing the database can be added to the `local.conf`:

```
BBMASK += "postgresql.*\bbappend"
```

To re-generate the database archive file from the database dump file (`sql.db`), following commands can be used on the target machine or virtual image:

1. Stop the PostgreSQL server:

```
$ systemctl stop postgresql
```

2. Remove old database and generate new:

```
$ rm -rf /var/lib/postgresql/data
$ su -l postgres -c "/usr/bin/initdb \
  --pgdata='/var/lib/postgresql/data' --auth='ident'"
```

3. Start the PostgreSQL server and restore db.sql:

```
$ systemctl start postgresql
$ sudo -u postgres psql -U postgres -f /var/lib/postgresql/db.sql
```

4. Generate a database archive

```
$ systemctl stop postgresql
$ tar -cf db.tar /var/lib/postgresql/data .
```

After this, the database can be transferred back to the build host so that BitBake can handle its installation.

## 4.17 Live USB image

To build a live image, which can be used to live boot the system or automatically install it from USB flash drive to DCU40, this configuration line is needed to be added to the `local.conf`:

```
IMAGE_FSTYPES += "iso"
```

This generates an `.iso` artifact (see Section 5.1), which can be flashed to a USB flash drive and the image can be booted from it or installed to DCU40's internal eMMC memory. See Section 5.4 for live USB flash drive usage.

The live image is actually a `initramfs` (initial ram filesystem) image, which means that it is loaded to the RAM of DCU40 from where it can load the actual image from the USB flash drive, or install it to the device's internal eMMC memory. The installing script can be found at:

```
yocto/meta-layers/local/meta-dcu40/recipes-core/initrdscripts/
  initramfs-module-install-efi/init-install-efi.sh
```

By default, this script prompts user input for selecting the target hard drive, but since by default DCU40 only has one, this step has been automated in the script so that booting from USB flash drive will automatically install the image to DCU40 if option `boot` is not selected during the `systemd-boot` prompt. It will also automatically partition the target device by using the `parted` partition tool. If a

specific partitioning scheme is needed to be used then the OpenEmbedded Image Creator Wic can be used, which is explained in Section 2.13.

## 4.18 Uninative settings

The Yocto Project's uninative prebuild glibc is designed to remove the differences between the host distributions, meaning that the native sstate objects can be shared between the build hosts. This is done by isolating the build system from the host distribution's C library [1]. Uninative settings are in `site.conf` starting with a `UNINATIVE_` prefix. These settings include filenames with checksums for different architecture tarballs, and the locations of them. The uninative tarball can be also build with a BitBake command:

```
$ bitbake uninative-tarball
```

## 4.19 Audio and graphics

Even though DCU40 does not have a traditional audio output and it is not designed to show any graphics from the DisplayPort output, the correct kernel configurations were added as kernel fragments so that audio can be played through DCU40's DisplayPort output to a display supporting PCM audio playback. The needed `local.conf` settings are:

```
DISTRO_FEATURES += "alsa"  
IMAGE_INSTALL_append = " alsa-utils pulseaudio"
```

Enabling graphics requires installing the X Server, hardware acceleration codecs and Open Graphics Library. Glxgears from MESA OpenGL demonstration program can be used to verify that the graphics work. Required `local.conf` lines are:

```
DISTRO_FEATURES += "x11 x11-base hwcodecs opengl"  
IMAGE_INSTALL_append = " xserver-xorg xinit xauth xterm xrandr \  
                        xf86-video-fbdev mesa-demos"
```

## 5 DEPLOYING THE BUILD ON THE DCU40 AND VIRTUAL ENVIRONMENTS

### 5.1 Build artifacts

After BitBake has built the target, the artifacts generated for DCU40 can be found in directory:

`yocto/build/dcu40/tmp/deploy/images/intel-corei7-64/`. Important artifacts are:

- `bzImage` is the kernel image.
- `core-image-minimal-initramfs-intel-corei7-64.cpio.gz`, is the initramfs image which is the live boot image
- `dcu40-image-intel-corei7-64.ext4`, is the root filesystem of the built image
- `dcu40-image-intel-corei7-64.iso`, is the live image capable of live booting the DCU40 or flashing it
- `dcu40-image-intel-corei7-64.qemuboot.conf`, is the configuration file for `runqemu` command
- `dcu40-image-intel-corei7-64.tar.bz2`, is used with iPXE network boot as root filesystem for the NFS server
- `dcu40-image-intel-corei7-64.wic`, is an image container of all created partitions (boot, rootfs and swap) and it is used in VirtualBox image creation
- `dcu40-image-intel-corei7-64.wic.vdi`, is the VirtualBox image

Other artifacts are also generated: a microcode update file, certificate files, UEFI files and archived kernel modules. The root filesystem produced by the build can also be inspected by running a command:

```
$ bitbake dcu40-image -c do_rootfs -f
```

which forces the root filesystem generation without sstate cache, after which it can be found in the build directory at:

```
tmp/work/intel_corei7_64-poky-linux/dcu40-image/1.0-r0/rootfs
```

A command can be used to find the recipe which produces an individual root filesystem file:

```
$ oe-pkgdata-util find-path <full_file_path>
```

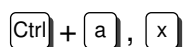
## 5.2 Running the build with runqemu command

Builds can be run immediately after successful build with Yocto's Qemu wrapper Python script with command:

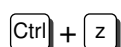
```
$ runqemu nographic kvm
```

The nographic argument disables the video output and enables a serial console. The kvm option enables the Kernel-based Virtual Machine, KVM, which is more reliable and faster than Qemu's emulated CPUs since it uses the host machine's CPU for hardware virtualization. For example, loading many Java classes seems to be unreliable when using an emulated CPU.

By default, this command does not use any specific virtual machine file, but it uses the kernel bzImage image together with a `rootfs.ext4` image from the build output directory. Additionally, this wrapper can pass custom parameters directly to Qemu or kernel. The basic runqemu command syntax is described in Yocto Project Mega-Manual section 8.9. [1]. To exit Qemu when the virtual machine is running the following keyboard combination can be used:

 `Ctrl + a, x`

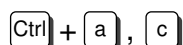
This way signals such as SIGINT can be used inside Qemu with a keyboard combination:

 `Ctrl + z`

Qemu also has a feature called Monitor, which can be used for various tasks such as:

- Removing and inserting media images
- Freezing and unfreezing the VM and saving/restoring its state
- Inspecting VM state without external debugger
- Limiting block device IO operations
- Inspecting CPU registers and memory

The Qemu Monitor can be opened and closed while the VM is running with a keyboard combination:

 `Ctrl + a, c`



### 5.3 Running the build in VirtualBox

VirtualBox by Oracle is a powerful virtual machine manager with a command line and a graphical user interface. It has lots of features such as:

- Ability to run multiple VMs at once
- Taking snapshots of VMs for testing and recovery purposes
- Portability, which means that VirtualBox can be run on many hosts and VMs can be shared among them
- Nested VT-x/AMD-v which enables running another VM inside the host VM
- Guest Additions, which can be installed inside of the VM for even more features
- Guest multiprocessing, where VirtualBox can present up to 32 virtual CPUs to the VM, irrespective of the host CPU core count

VirtualBox's nested VT-x feature was tested by creating a VirtualBox image for DCU40, which was equipped with Qemu, so that it could run a Debian 10 virtual machine. This kind of setup has its use cases when the customers want to run their own virtual machines in the DCU40. Firstly, Qemu was used to create an empty image with its own .qcow2 format, and then the Debian installation was started for the image:

```
$ qemu-img create -f qcow2 debian.qcow2 2G
$ qemu-system-x86_64 -hda debian.qcow \
  -cdrom debian-10.6.0-amd64-netinst.iso -boot d -m 1024
```

The installation won't have an internet connection, thus only the Debian Base System is installed for the minimal image size. Required packages can be installed afterwards with APT. Only a single ext4 partition is created without a swap. After the installation is finished, the kernel and the ramdisk are extracted from the .qcow2 image with a command:

```
$ virt-get-kernel --add debian.qcow2
```

After this, they can be provided to Qemu as individual arguments, which enables the possibility of setting the kernel command line arguments. This is useful since then the console output of the Debian 10 VM can be redirected to the host's terminal. This way all the boot messages are shown, which normally can only be done with the X11 window manager, which might not be installed on a headless server host such as DCU40.

In VirtualBox following settings are used:

- Settings -> Motherboard -> Enable EFI, since the image is only UEFI bootable

- Settings -> Processor -> 4 CPU, Enable Nested VT-x/AMD-v
- Settings -> Network -> Bridged Adapter Name: *VLAN with internet connection on host*, Promiscuous mode: Allow VMs

After the DCU40 image is booted up in VirtualBox, a network bridge is created for eth0 which is the internet enabled VLAN interface. Finally, a tap interface is attached to the bridge and an IP address is requested for it:

```
$ ip link add br0 type bridge
$ ip link set eth0 master br0
$ ip tuntap add tap0 mode tap user root
$ ip link set tap0 up
$ ip link set tap0 master br0
$ dhclient br0
```

After this, the Debian 10 VM can be started with Qemu:

```
$ qemu-system-x86_64 -kernel vmlinuz-4.19.0-11-amd64 \
  -append "root=/dev/sda1 console=ttyS0" \
  -initrd initrd.img-4.19.0-11-amd64 -hda debian_10.qcow2 \
  -m 1024 -cpu kvm64 -enable-kvm -smp 1 -nographic \
  -device virtio-net-pci,netdev=net0,mac=12:34:56:78:90:00 \
  -netdev tap,id=net0,ifname=tap0,script=no,downscript=no
```

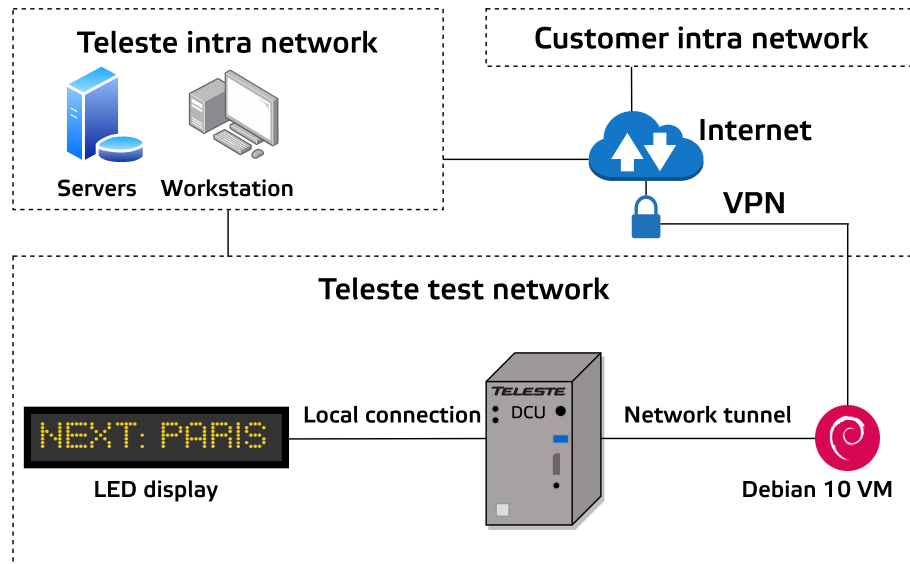
where:

- `-kernel` takes the kernel binary as an argument
- `-append` provides the kernel command line arguments
- `-initrd` points to the initial ramdisk
- `-hda` points to the image with root filesystem (it also has kernel and initrd inside)
- `-m` sets the RAM amount, `-cpu` CPU type, `-enable-kvm` enables KVM and `-smp` processor core count
- `-nographic` redirects input/output to the current terminal and disables graphical output
- `-device` sets the virtual network adapter
- `-netdev` uses previously generated tap0 to connect the the br0

The MAC address can be freely generated. After shutting down the virtual machine the bridge and tap interface can be cleared with:

```
$ ip link del tap0
$ ip link del br0
```

An automation script was written for this setup. Figure 5.1 is an illustration about the network structure, which was set up for the physical DCU40, after its functionality was verified in a virtualized environment. The setup allows the customer to write programs for the Debian 10 virtual machine inside DCU40, which can interact with Teleste's LED displays, for example.



**Figure 5.1.** Customer interactable Debian 10 virtual machine running on DCU40

## 5.4 Using the live image

The build artifact `dcu40-image-intel-corei7-64.iso` can be flashed to a USB flash drive with command-line utility *Bmaptool* by executing the following commands:

```
$ bmaptool create dcu40-image-intel-corei7-64.iso \
  dcu40-image-intel-corei7-64.iso.bmap
$ sudo bmaptool copy dcu40-image-intel-corei7-64.iso /dev/sdX
```

where `sdX` is the correct block device of the USB flash drive. The BIOS setup of DCU40 has to have the UEFI boot option enabled for the USB drive to boot up. After booting to the systemd-boot bootloader, there is a configurable 10 second timeout before the default boot entry of *install* is executed and the device is flashed. If *boot* is selected, then the image is only booted up from the USB flash drive. The performance and boot up time are not as fast as from a eMMC memory but that is expected behaviour from the live USB drive, since the read and write speeds over USB bus are not as fast as eMMC's speeds. Flashing speed performance of different utilities was tested in Section 7.1.1.

## 5.5 Network boot of DCU40

Because creation of a live USB flash drive and flashing the DCU40 with it takes some time, a flexible UEFI network boot environment was set up. With network boot, the DCU40's eMMC memory does not need to be flashed at all but it is booted up from TFTP and NFS servers which are running on the build host. This is achieved with iPXE, which is an open source network boot firmware implementation following Intel's PXE specification. This works by running a DHCP server on the build host which provides the address of a TFTP server to DCU40 which is hosting three files:

- `bzImage`, the kernel image
- `install.ipxe`, the iPXE menu configuration file
- `ipxe.efi`, the iPXE boot loader executable

First, the iPXE boot loader executable is downloaded from the TFTP server to DCU40 and executed, which then uses the iPXE menu configuration file to download the kernel image and set the NFS share as root filesystem.

iPXE was chosen for this environment because of its flexibility and features which include:

- Controlling the boot process with a menu script
- Support for UEFI network boot over different protocols
- Chainloading with DCU40's network card's PXE implementation by Intel

### 5.5.1 Setting up a TFTP server

`tftp-hpa` was chosen as the TFTP server on the build host, and the only configuration which was done for it was to change the default `/srv/tftp/` directory to be writable by everyone:

```
$ sudo chmod 755 /srv/tftp/
```

This is for the automated network boot deploying script, which is located at:

```
yocto/build/dcu40-image/scripts/network_boot_deploy.sh
```

The TFTP server can host multiple different kernels which can be configured in the `install.ipxe` iPXE menu configuration file. During the network boot, the wanted boot selection can be chosen on the DCU40 and default set in the iPXE menu configuration file.

## 5.5.2 Setting up a DHCP server

A DHCP server is needed to be configured for network boot, which tells the location of network boot files to the client DCU40.

On Debian and Ubuntu a DHCP server is installed and configured with:

```
$ sudo apt install isc-dhcp-server
$ echo 'INTERFACESv4="<interface>"' | sudo tee -a /etc/default/isc-dhcp-server

$ sudo tee -a /etc/dhcp/dhcpd.conf << EOF
# Network for DHCP server
subnet <subnet> netmask <netmask> {
    option routers <host_ip>;
    option domain-name-servers <host_ip>;
    option subnet-mask <netmask>;
    option broadcast-address <broadcast>;
}

# PXE network boot
allow booting;
next-server <host_ip>;
option client-arch code 93 = unsigned integer 16;
if option client-arch != 00:00 {
    filename "ipxe.efi";
}

# DCU40 static IP
host cu {
    hardware ethernet <dcu40_mac>;
    fixed-address <dcu40_ip>;
}
EOF
$ sudo service isc-dhcp-server restart
```

where:

- <subnet> is the subnet used for the connection
- <netmask> is the network mask for the subnet
- <broadcast> is the broadcast address for the connection
- <interface> is the name of the interface used for connecting to DCU40
- <host\_ip> is the host's IP address on the <interface>
- <dcu40\_mac> is the MAC address of the DCU40
- <dcu40\_ip> is the IP address wanted for the DCU40

It is assumed, that the interface does not have any other DHCP servers running and it is configured with static IP for the host. A convenient way, is to set up a separate VLAN for <interface>. The "PXE network boot" section of the configuration follows DHCP specification for PXE by defining "Client System Architecture Type Option Definition" [18]. This configuration allows booting of clients with different architecture and both BIOS and UEFI boot. Currently, DHCP returns the filename "ipxe.efi" if the client gives any other DHCP option 93 than 0, which is standard PC BIOS. For example DHCP options can be defined for ARM systems also.

### 5.5.3 Chainloading iPXE

Because DCU40 already has a basic PXE implementation on its network card iPXE's chainloading is used to first boot with PXE, and then chainload to iPXE's boot menu. Doing this requires building an EFI file with iPXE's source code. The build of ipxe.efi binary requires a chain.ipxe, which has the host's TFTP server IP address and menu.ipxe as a chain target.

A template for chain.ipxe:

```
#!/ipxe
dhcp
chain tftp://%HOST_IP%/menu.ipxe
```

and a template for menu.ipxe, which has the boot option for dcu40-image:

```
#!/ipxe
:start
menu Please choose boot image
item --gap DCU40
item dcu40-image dcu40-image
item --gap ipxe shell
item shell          Drop to iPXE shell
item reboot        Reboot machine

choose --default dcu40-image --timeout 5000 target \
    && goto ${target}

:shell
echo Type 'exit' to get the back to the menu
shell
goto start

:reboot
echo Rebooting
reboot

:failed
```

```

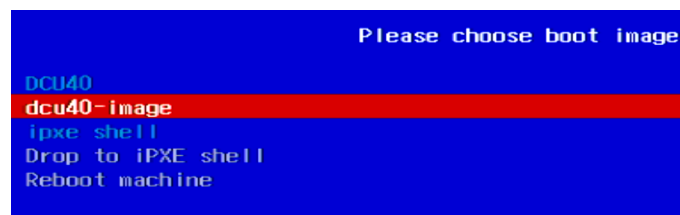
echo Booting failed, going back to start
goto start

# DCU40 image in the build directory
:dcu40-image
kernel tftp://%HOST_IP%/bzImage \
root=/dev/nfs nfsroot=%HOST_IP%:%YOCTO_DIR%/build/dcu40-image/nfs-
boot,nfsvers=3,port=3049,udp,mountport=3048 \
ip=dhcp raid=noautodetect rootwait console=tty0

boot || goto failed

```

For convenience, a script `yocto/scripts/make_ipxe_boot.sh` was written, which automates the file generation from templates by using `sed` stream editor, and building of the `ipxe.efi` binary with help of the iPXE GIT repository added as a submodule, and deploying the files to the TFTP server. As can be seen from the `goto` statements in `menu.ipxe`, the iPXE's command language is fairly primitive but it provides a configurable bootloader menu like in Figure 5.2. iPXE's command reference has all the available commands documented which can be used in iPXE shell or in scripts [19].



**Figure 5.2.** iPXE bootloader menu

## 5.5.4 NFS root filesystem

After the kernel has been downloaded to the DCU40 then the NFS server address is given as a kernel command line option. To deploy the NFS server in user space following commands can be user:

```

$ runqemu-extract-sdk <rootfs_tarball> <nfs_rootfs_dir>
$ runqemu-export-rootfs <nfs_rootfs_dir>

```

where:

- `rootfs_tarball` is the tar package of the root filesystem in the build output directory
- `nfs_rootfs_dir` is the directory which is to be used as a NFS share

The root filesystem can be interacted with and modified when the DCU40 is running from it.

## 6 A COMPARISON OF BITBAKE AND TELESTE'S IMAGETOOLS

### 6.1 History of build automation at Teleste

Build automation and reproducibility have been important aspects since the first DCU type devices were developed at Mitron Oy, which Teleste Oyj acquired in 2015. In 2012, a DCU type device was an acronym for *Display Control plug-in Unit*, which main role was to control TFT and LED displays on a railway vehicle. The main programming language on DCU was Java, which is still the most used language for Teleste's platform components.

#### 6.1.1 Rocket Tools

One of the first build automation tools was called *Rocket Tools*, which is a command line build system written for Bash. The main functionalities of Rocket tools are:

- Debian package management and building
- Dependencies management
- SVN checking out and committing
- Automatic changelog updating
- Virtualisation with VirtualBox
- Maven build system management, especially *Project Object Model* (.pom) files handling

However, Rocket Tools has its drawbacks. First of all, it was mainly written by one developer why the working principles of it were not so easy to understand for other developers. The main script of Rocket Tools is called `rt_release.sh`, and it is over 2000 lines of bash functions, inline file generation, and external program calls. It has to be run with `sudo`, which is said to cause problems on the build host since the build environment is not isolated, meaning, that the script can modify the system files of the build host. Nowadays, it is recommended to run Rocket Tools in a virtual machine for old legacy projects, because of possible host contamination.



### 6.1.2 Maven

Rocket Tools was eventually replaced with a pure Maven build system which can be run in developers' IDE, command line or on dedicated Jenkins build hosts. Maven is a building tool mainly for Java, but with its plugin system other languages are also supported. Its objectives are [20]:

- Making the build process easy by hiding many underlying mechanisms
- Providing a uniform build system with its *Project Object Model* so that every project has same the structure
- Providing project information such as changelog updating, sources' cross referencing, dependencies management and unit tests
- Encouraging better development practices with test source code separation, naming schemes and project layout guidelines

In Teleste's projects Maven is used with a collection of plugins:

- Release plugin, which can be used from Jenkins' web interface to create new tagged releases and update VCS
- Execution plugin for executing any available program on the build host such as make
- Dependency management
- Debian package building
- Update Manager compatible update package building

Maven is a modular build system where the individual components can be built separately or a full project build can be executed and if Maven's (.pom) files are written correctly which means, that higher level artifacts define sub-artifacts as modules. Artifacts also have dependencies which are downloaded from Teleste's Nexus repository which stores platform component build artifacts' (.pom) files and prebuilt (.jar) files. The platform components are versioned which makes it easy to choose the correct versions for each project.

### 6.1.3 Jenkins

Jenkins, an open source automation server, is the default continuous integration server for Teleste's projects. It can be used for all sorts of tasks, such as building, testing, delivering and deploying of software. At Teleste, Jenkins also creates update packages for the other PIS devices which are installed by using the Updatemager interface hosted on the DCU40.

Jenkins can also be used to run BitBake, which would make the continuous

integration of new image releases fast and easy. It is not practical for every developer to set up the Yocto build environment, but a centralized build host can be used to automatically start new builds once new commits are pushed to a Git repository. Other automatic tasks can also be set up such as refreshing the network bootable images, running tests in a virtual machine or on an actual hardware and updating the downloads and sstate-cache hosted on the Teleste's network share. The Yocto Project also provides guidelines for creating a team development environment [1].

### 6.1.4 Imagetools

Imagetools is the current image creation tool at Teleste. It is a collection of Bash and Python scripts and tools, and it is used to create USB installer flash drives for the most of Teleste's devices. Before the new line of Teleste's PIS devices started to use internal eMMC memory for the operating system, the devices normally used SSDs for that purpose. Imagetools can also flash SSDs and HDDs directly after which the flashed disks can be installed to the device. Imagetools has three main user initialized scripts:

- `imagemtools_flash_image`, which flashes an image to a storage medium
- `imagemtools_create_usb_installer`, which creates a USB installer flash drive for flashing internal storage mediums such as eMMC
- `imagemtools_build_virtual_image`, which can build VirtualBox, VMware or raw virtual machine images

Eventually, the main script `imagemtools_flash_image` is called in every case which does the actual image flashing. This script calls a sequential collection of numbered scripts which do the following in this order:

1. `100_setup_config`, reads the main configuration file `imagemtools.conf`, which can be used to set global variables for Imagetools
2. `200_partition_disks`, creates partitions according to a predefined partition scheme
3. `300_make_filesystem`, creates file systems for partitions
4. `400_mount_filesystem`, mounts the partitions to the build host
5. `500_unpack_data_to_filesystem`, unpacks the pre-base image
6. `600_addons`, installs Debian packages and miscellaneous configurations
7. `700_grub`, installs GRUB bootloader
8. `800_unmount_filesystem`, finishes the installation and cleans up the environment

More numbered custom scripts can be added, which are run in the number order. However, same things are done in a different way by different developers when writing these custom automation scripts. Normally, the extra scripts do not use Imagetools' logging and execution wrappers, but they just have the bare minimum statements to archive a goal, such as copying a configuration file or installing an extra package to the image. To change the behaviour of a predefined numbered script is has to be added to a project build with the same number and name so that it overwrites the predefined one. In BitBake, commands can be appended and prepended to individual tasks which is usually only what is needed even though tasks can be completely overwritten.

Imagetools workflow is pictured in the Figure 6.1. The workflow chart does not describe all the steps and processes involved in a full Imagetools run, but it shows all the major phases of a build. Gray and green colors are used to separate the automated processes from manual configurations. The workflow is split to four different phases which are:

**Phase 1: Pre-base image creation.** Building an image with Imagetools begins with creation of a *Pre-base image*, which is a process of installing a Linux distribution, Debian in this case, to the target device such as DCU40, for example. After the Debian installation, the DCU40 is either connected to the internet for Debian repository access, or needed packages are downloaded with another machine and transferred and installed to the device. Dependencies are added for hardware devices, Python interpreter and Java Runtime Environment are installed, and Teleste's basic platform components such as Update Manager are installed together with HTTP-server and SQL database. After the installation is configured and tested, a full disk image of the installation is taken by generating a tar archive out of it, which is usually 1 GB in size.

**Phase 2: Base image creation.** After the pre-base image is created, it is stored to the Teleste's network share, which has a specific directory structure set up for Imagetools. The base image directory is set up in a way that if Imagetools is run in this directory, it can directly flash external storage mediums or create a USB installer flash drive, which can install the pre-base image back to DCU40. Also, extra configuration files and Debian packages can be added to the base image directory, which is more flexible method of modifying the pre-base image than working with the tar archive. The base image is a stable base line for the project images thus it should not be modified later on.

**Phase 3: Project image creation.** As a phase, this is basically identical to phase 2, but the installation is done over the base image rather than over the pre-base image. The purpose of this phase is to modify and expand the features of the base image so that it fulfils the requirements of a project specific image. For

example, project specific GPG keys, configuration, and Debian packages are installed. Project related platform components can be installed in this phase or in the next one. As in the previous phase, the project image should not be modified after creation because the specific image version is used in production, where the devices are assembled, installed, and sent to the customers.

**Phase 4: Project package installation.** The iterative developing of a project is done in the last phase. Maven and Jenkins are used to build software update packages, which are installed to the device via Update Manager. In some projects the deployment of update packages can be done from a centralized update server, which sends the update packages to all vehicles and installs them automatically. Normally during project development, this phase is used for adding new features, fixing software bugs, and creating major software releases for the project.

As a process, the Imagetools workflow is highly sequential and the iterative developing is only done in the last phase. This means that the first 3 phases should be executed without any errors, so that the iterative work could be done on a stable Debian installation. However, since the build process is highly influenced by many individual scripts, written by many developers, non fatal errors normally happen during the build. Fixing these errors might be dangerous since the same base image is used for many projects where the erroneous behaviour might work in a different way because of different combination of scripts. If modifications are needed to be done to a base image, new versions of the base and the project images have to be created.

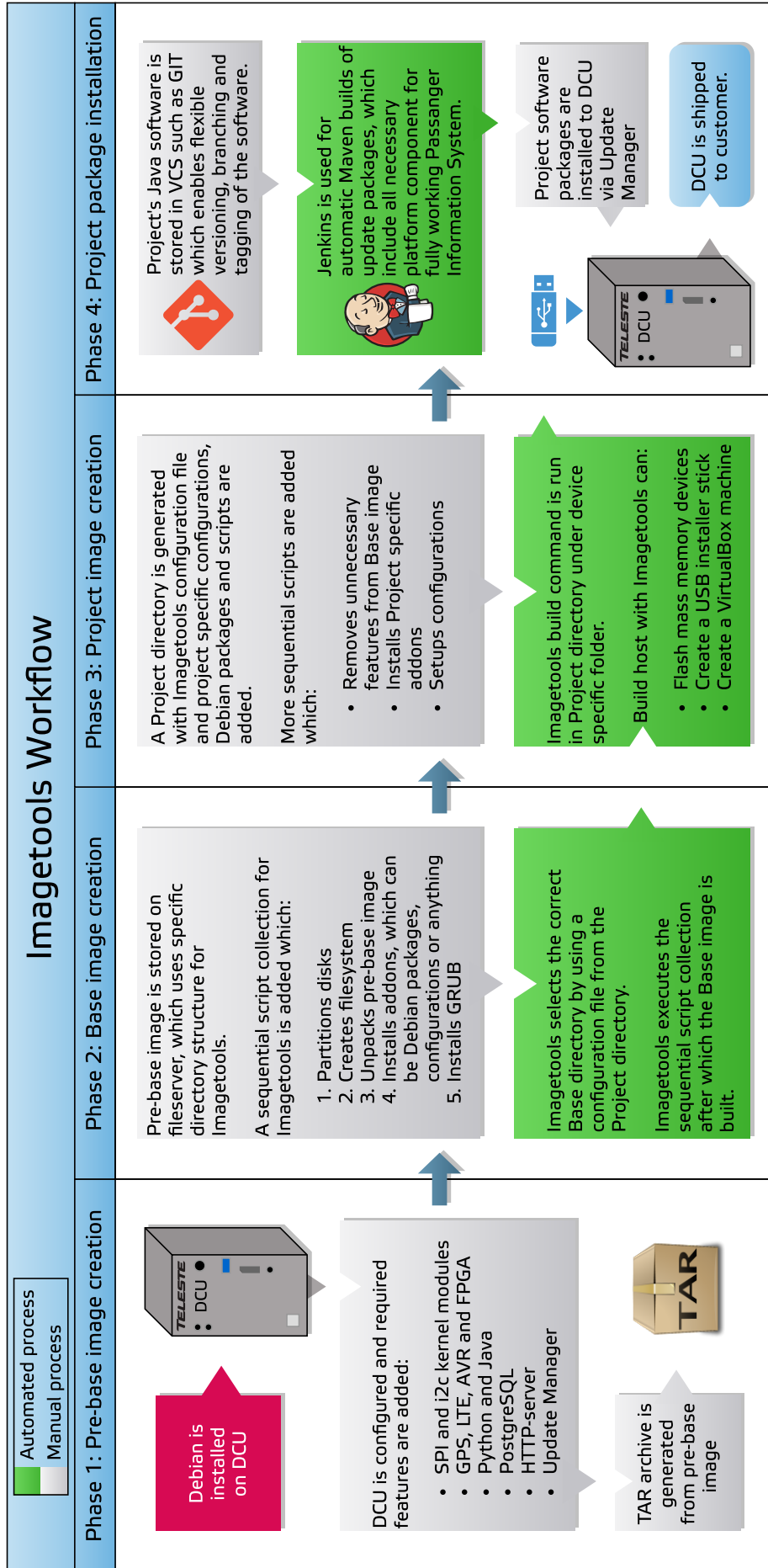


Figure 6.1. Workflow visualization of Imagetools

## 6.2 BitBake compared to Imagetools

Both tools can be thought as build automation and task execution engines which are user configurable. The main difference between the build artifacts produced by the tools is the distribution, which for Imagetools is stripped down Debian and for BitBake Yocto Project's reference distribution Poky. Imagetools uses prebuilt packages while BitBake compiles them from source code. The core idea, underlying principles, and the workflow of the tools are so different that comparing the tools to each other is hard.

Not all compared terms, such as a *task*, share any similarities due to the complexity of BitBake compared to Imagetools. As a good example, installing bootloader (see also Section 4.9) is a task in both tools: in Imagetools it is a single script which calls `grub-install` program to install the GRUB to the target device during image flashing. In BitBake, installing bootloader is done by including a recipe either for GRUB or `systemd-boot`, which BitBake parses and executes the tasks related to it. For example, recipe for `systemd-boot` is located in:

```
yocto/poky/meta/recipes-core/systemd/systemd-boot_244.3.bb
```

and it is 70 lines long. During image creation for DCU40 with BitBake, 17 different tasks are processed of which 7 tasks are cached thus not needed to be re-executed, and 9 tasks have been covered by other tasks. So nearly all tasks have already been previously executed and cached, thus they can just be fetched from the cache.

Some feature differences of BitBake and Imagetools have been collected to the Table 6.1. The compared features might not be exactly comparative, but the table provides some insight of the features for both tools.

**Table 6.1.** *BitBake and Imagetools features compared*

Feature	BitBake	Imagetools
Architecture agnostic builds	Yes, uses internal tools and C library	No, uses build host's tools
Automatic image testing support	Yes, on virtual and actual machine	No
Build cache	Yes, shareable downloads and intermediate build task caching	No
Build speed bottleneck	Amount of CPU threads and RAM	Read and write speeds
Continuous integration support	Yes	No
Custom scripts	Python 3 and Shell scripts	Bash scripts
Custom tasks	New tasks can be added to recipes	Numbered bash scripts
Distribution maintainer	Teleste	Debian
Image modifiability	Everything is built from source code	Uses prebuilt packages and binaries
Image versioning	Configurations, tools and metadata are in GIT	Version number for base and project image
Kernel management	Version selection and full configuration	Not implemented
Modularity	Multiple layers with recipes, inheritable classes and package groups	Base and project layer
Parallelism support	Yes, task and compiling parallelism	No, sequential task execution
Required privileges	Normal user	Root user
Syntax checking	Automatic for all metadata	No, runtime errors
Tool main language	Python 3	Bash and Python 2
Tool debuggability	Built-in solutions	Script forking
Tool maintainer	The Yocto Project	Teleste
Variable syntax and operators	Advanced in-built syntax and features	Bash variables

## 6.3 BitBake workflow

BitBake is also described under Section 2.4, but the workflow is described in here together with Imagetools' workflow. Figure 6.2 shows a workflow visualization of BitBake. Rather than having separate phases like in the Imagetools workflow the functionality is wrapped around the BitBake task execution engine. The next sections describe these functionalities in more detail.

### 6.3.1 Local meta-layers

As the project is structured as a Git superproject with other Git repositories as submodules, the local meta layers in this superproject is one part, which requires manual work. This work includes writing recipes for Teleste's platform components and for DCU40's hardware components together with device specific configurations. Also, common recipes are written for things such as SQL database initialization, python packages, and Sky Blue distribution configurations. For Teleste's platform components, the recipe version numbers match the component versions they install. However, the version is started from 1.0.0 for additional recipes such as virtual machine only scripts and configurations, which are written specifically for BitBake and DCU40.

Yocto's layer concept makes it possible to have some settings in many different places. For example, user configuration can come from a single recipe which needs that user, from the DCU40 image recipe, or from the Sky Blue distribution configurations. BitBake will notice if there are conflicting configurations across layers which are included to the build.

### 6.3.2 Remote meta-layers and Poky reference distribution

These layers are managed by other parties such as The Yocto Project, Intel, OpenEmbedded, or even individual people. They provide ready made recipes which can be included into the build by just declaring the wanted recipe in:

- `local.conf` which is the build specific configuration
- in Sky Blue's metadata
- as part of package group
- as feature of `dcu40-image` recipe
- as a dependency of some other recipe

Some of these recipes are quite simple but others make use of Yocto's framework in an advanced way which is why recipes by bigger organisations can be used as



guidelines when writing new recipes stored in local-meta layers. As the remote meta-layers are just Git submodules, they can be updated with a single command, see Section 3.5 for more details. To reduce the size of the build environment, some of the local meta-layers can also be set up as separate Git repositories, which can be added as submodules to the superproject. Even though the remote recipes are maintained by others, every aspect of them can still be modified with `.bbappend` files, which means that usually there is no need to copy a full recipe from a remote layer to a local one. However, if the recipe needs a lot of modifications, then it might be useful to copy the whole existing recipe: BitBake will require a layer selection if multiple layers provide the same recipe.

The Poky reference distribution is a special Git submodule in the project, since it has its own meta-layers for building many different Poky distribution versions for many architectures and it also includes all the tools, utilities and scripts such as BitBake, Devtool, and runqemu. The Poky Git repository does not require anything else for the build process. However, normally other layers are added for more pre-written recipes of packages.

### 6.3.3 Build configuration for DCU40

The build configuration consists of three configuration files, which have around 150 effective configuration lines, while the comments take close to 300 lines. The configuration is described in more detail in Section 4. The configuration file `local.conf` can be compared to Imagetools' `imagemetools.conf` since they are both tied to the build directory, and they both control the build by modifying the global build runtime variables used by both tools.

Imagetools does have a base and a project layers while BitBake can have any number of layers to be included in `bblayers.conf`. For example, a project specific meta-layer repository can be created as a Git repository and added as a submodule to the build environment. `site.conf` has the addresses for downloads and sstate cache directories, and univariate tarball location.

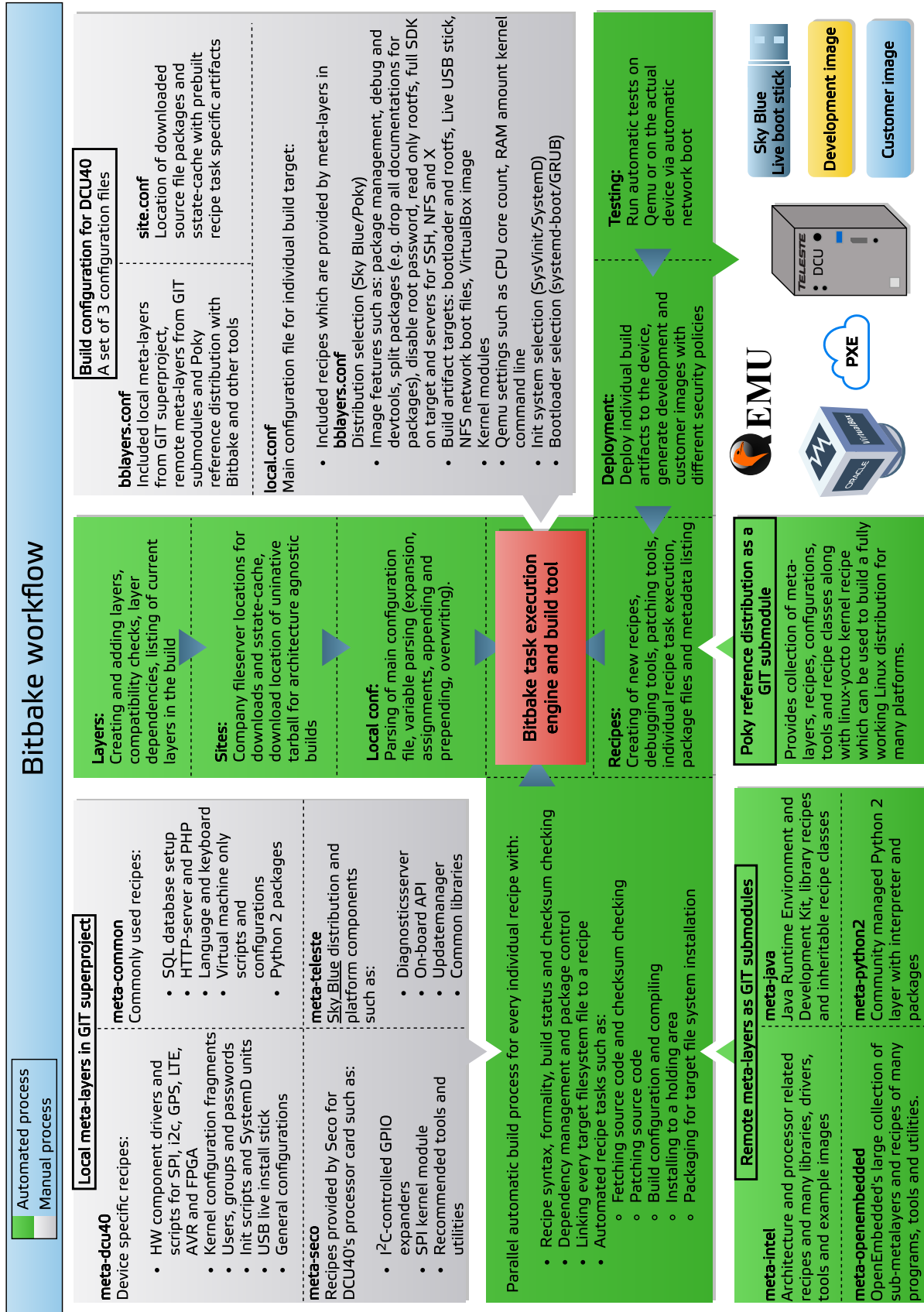


Figure 6.2. Workflow visualization of Imagetools

## 7 TESTING

### 7.1 Boot-up times

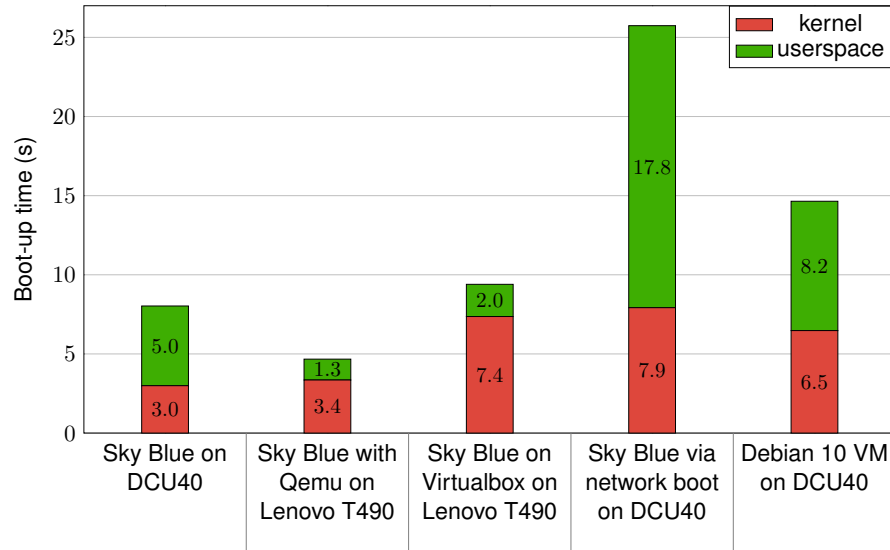
Boot-up times of the DCU40 image were measured in different situations. A command `systemd-analyze` was used to determine the exact boot-up time which also breaks it down to the time spend in kernel and after that in user space. It can be seen from the Figure 7.1, that the boot-up times are fairly fast and during testing and configuring a clean reboot of the whole system was used multiple times as a method of applying modified configurations without restarting individual services. Especially booting up the Sky Blue image with Qemu and KVM was fast, which made it easy to tweak the system and troubleshoot problems. The VirtualBox image with KVM was not as fast as the Qemu image but in some use cases using VirtualBox can be more intuitive and easier since it has a GUI with easy to access settings.

Network booting Sky Blue on DCU40 with iPXE was slower than booting it up from the internal eMMC memory but since no live USB flash drive has to be created nor the eMMC memory has to be flashed, the network boot was especially suitable for fast image development iterations on the DCU40 when the slower boot-up times were not an issue.

#### 7.1.1 USB flash drive flashing speed

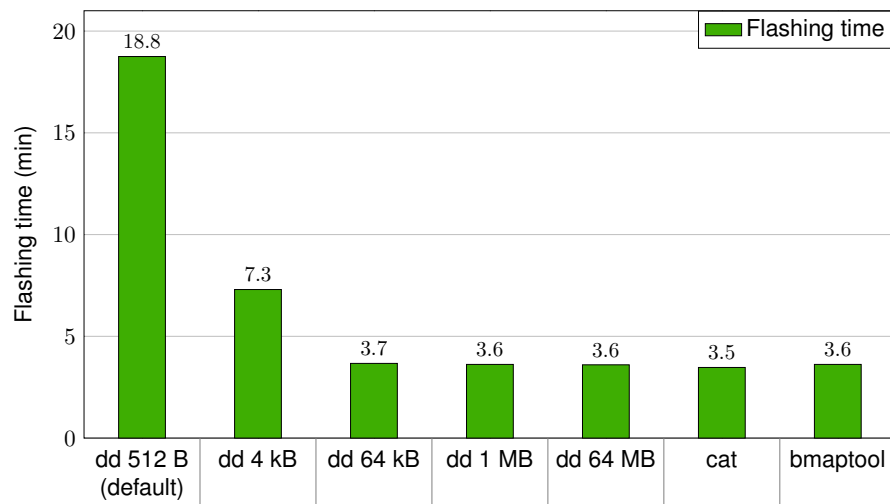
Yocto Project recommends using a tool called `bmptool` to flash images and notes, that it's generally 10 to 20 times faster than `dd` utility [1]. This was tested by flashing a BitBake generated 4.1 GB live boot image to a 16 GB Kingston USB 3 flash drive with `bmptool`, `dd` with different block sizes and also with `cat`. From the results in Figure 7.2 it can be seen, that `dd` is indeed slower than `cat` and `bmptool` if the default block size of 512 bytes is used. However, if the block size for `dd` is increased to 64 kB it will flash the USB flash drive at the same speed as `cat` and `bmptool`.

An automation script was written for `bmptool`, which prompts user to insert the USB flash drive thus mimicking the functionality of `Imagetools`. Other pros of the `bmptool` are that it constantly verifies data integrity while flashing, it can read



**Figure 7.1.** Boot-up times on different platforms

images from remote servers (which is useful in a company network) and it has protection mechanisms, such as the prevention of flashing mounted block devices [21]. It also shows the flashing speed and a progress bar during flashing. One key difference between Imagetools and BitBake is that when creating USB drives Imagetools writes many individual files to the USB drive in a chroot environment while BitBake generates a single file, which can then be flashed to the USB drive even with Windows tools such as the open source software Rufus.



**Figure 7.2.** Flashing time of 4.1 GB Sky Blue live image to a USB flash drive with different tools

## 7.2 Meltdown and Spectre mitigations performance impact

Spectre and Meltdown are critical security vulnerabilities found in January 2018 which affect most modern CPUs. They are not software vulnerabilities, but the core

problems are found in the CPU hardware which makes them hard to fix. Meltdown breaks the memory isolation between user applications and the operating system, and Spectre breaks the memory isolation between user applications. These isolation breaches allow malicious software to access the system memory and extract secrets, such as keys and passwords, of other applications and the operating system. Since these exploited features are commonly used in CPUs for better performance the fixes for these exploits, also called mitigations, can have a performance impact to the CPU which will be investigated in the next sections. [22]

### 7.2.1 Meltdown and Spectre vulnerabilities

Meltdown exploits the side effects of out-of-order execution found in modern processors. Out-of-order execution is a performance feature which enables execution of sequential processor instructions defined in a program in a parallel way, so that if the next instruction in the program does not depend on the previous ones, it can be executed before the previous instructions are completed. Meltdown enables privileged system memory access only available to the operating system from a user application which does not have access rights to that memory. [23]

Normally, if a program tries to access system memory addresses which are not allocated for it, the OS will usually react to this by sending an exception signal to the program forbidding the memory access. Languages like C provide low-level memory access methods so that the program can normally access memory addresses and their data, which are within the process memory space. For example, if the following pseudocode would try to access a memory location which is not allocated for it, the OS would forbid it and there would be an exception in the program:

```
secret = read_memory_value(forbidden_memory_address);
```

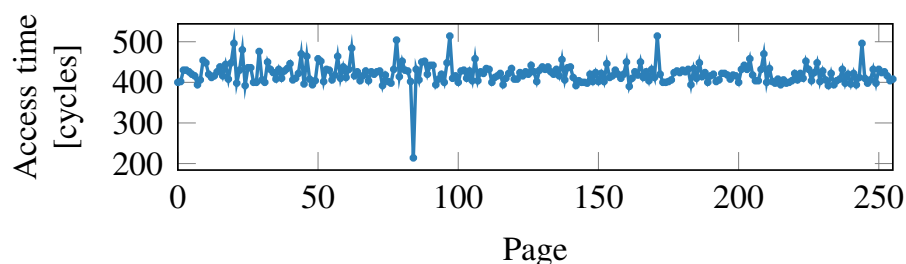
thus the secret would not be revealed.

However, in Meltdown, a probing array is constructed and the value of the secret is used as an index to probe the array:

```
probe_array = {0, 1, 2, ..., 256};  
probe_array[secret];
```

This will lead to a forbidden memory access error but because of out-of-order execution the value of the secret was used to access the probe array which causes the CPU to fetch that array value to the CPU cache. Even though the execution of the probing has to be discarded and register and memory contents are not committed, but the CPU cache is not flushed which makes it possible to use a

side channel timing attack to exploit the load time difference of an array value from slower RAM compared to a fast CPU cache. So when iterating the probe array, one value will be fetched significantly faster as seen in the Figure 7.3 than others, thus the secret is revealed. [23]



**Figure 7.3.** Access time of probe array's page fetching times in clock cycles

Spectre exploits branch prediction and speculative execution, where the CPU can guess which branch would most likely be executed before the branch conditions are validated. If the prediction was correct, the predicted execution is allowed to be committed and if it was incorrect the execution will be discarded and the correct branch will be executed instead. However, in this case the result of an incorrect execution will be stored into the CPU cache and it can be found out in a similar way than in Meltdown with a side channel timing attack. For example, the Spectre exploit can be used to read data from a browser tab to another. [24]

## 7.2.2 Mitigating Meltdown and Spectre in DCU40

DCU40's CPU card manufacturer Seco has provided BIOS updates for the card which includes Intel's CPU microcode updates, which mitigate the Meltdown and Spectre vulnerabilities. These BIOS updates also include other stability and feature updates, such as support for USB 3.1 and eMMC memory updates. Because these mitigations target the CPU's performance features they should have a degrading impact on overall CPU performance, which was tested by running performance benchmarks with **Phoronix Test Suite** with the default mitigations, and by disabling them with kernel command-line parameter:

```
mitigations=off.
```

This is available for kernel starting from 5.2 as a single parameter, and it disables all mitigations fully. Table 7.1 shows the four different test scenarios.

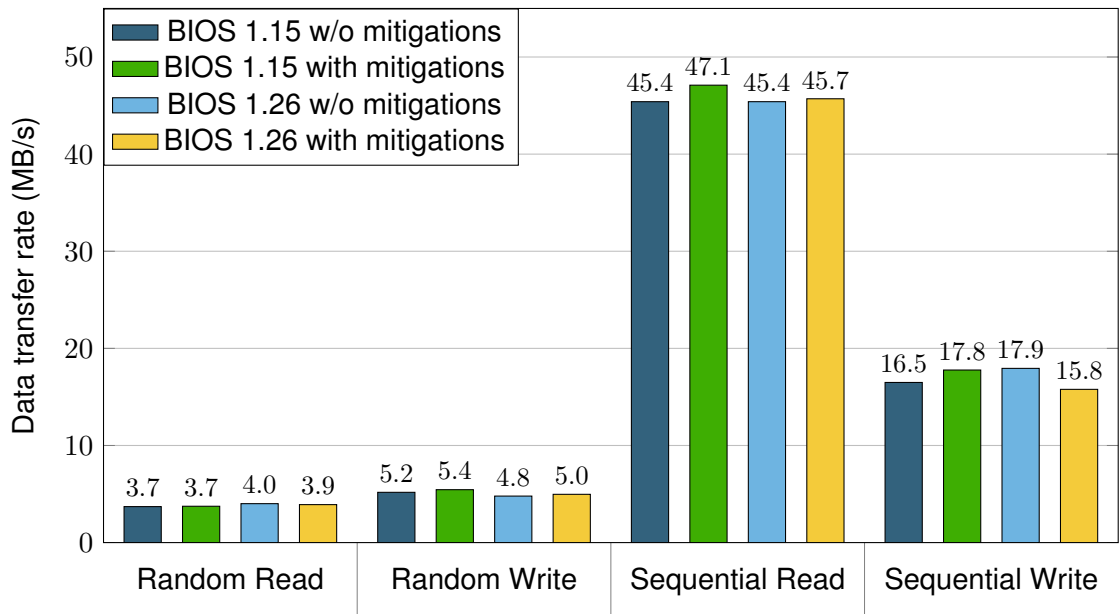
**Table 7.1.** Test configurations for DCU40 Meltdown and Spectre mitigations performance impact

Factor	Old BIOS without mitigations	Old BIOS with mitigations	New BIOS without mitigations	New BIOS with mitigations
Bios version	1.15	1.15	1.26	1.26
Microcode version	0x907	0x907	0x90A	0x90A
Kernel version	5.4.69	5.4.69	5.4.69	5.4.69
Kernel command-line	mitigations=off	mitigations=auto	mitigations=off	mitigations=auto

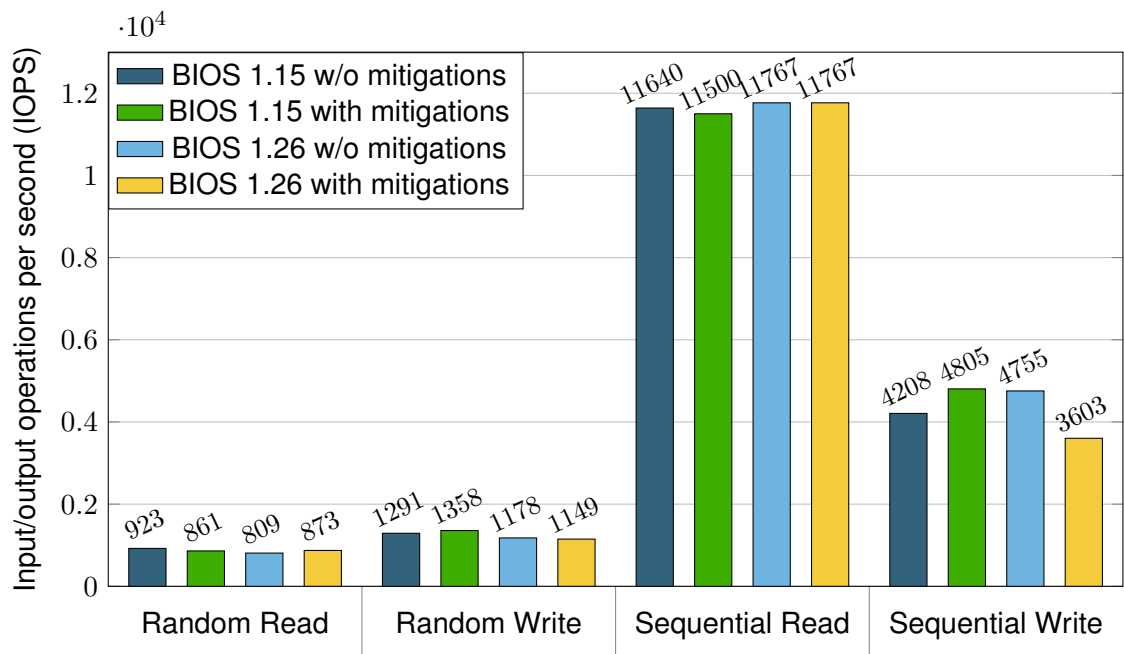
### 7.2.3 Analyzing test results

The initial assumption was that the BIOS version 1.26 with Intel microcode mitigations would have a performance impact on DCU40. The Figures 7.4 and 7.5 about disk I/O, and the Figure 7.6 about Java tests show that the newer BIOS version 1.26 indeed has a performance impact compared to bios 1.15. However, having mitigations enabled or disabled did not have any significant difference in the tests compared to performance degradation caused by the updated BIOS. The changelog for the BIOS lists many other changes other than the CPU microcode update for mitigating the Meltdown and Spectre vulnerabilities, which can be the reason for the performance differences since the kernel command line option to disable the mitigations did not seem to have an effect on system performance.

The BIOS changelog specifies that the version which has the vulnerability fixes is version 1.19, so that could be compared to the version 1.18 to better see the performance impact caused by the microcode update. The kernel version in this test setup is newer than in the Debian based DCU40 images, which probably gives different kinds of results with these two BIOS versions. Setting up the Phoronix Test Suite on the Yocto environment required manually installing the dependencies the tests needed, but an automation script was written for it so that subsequent tests could be run automatically in the future if more testing would be needed before the newest BIOS could be used.

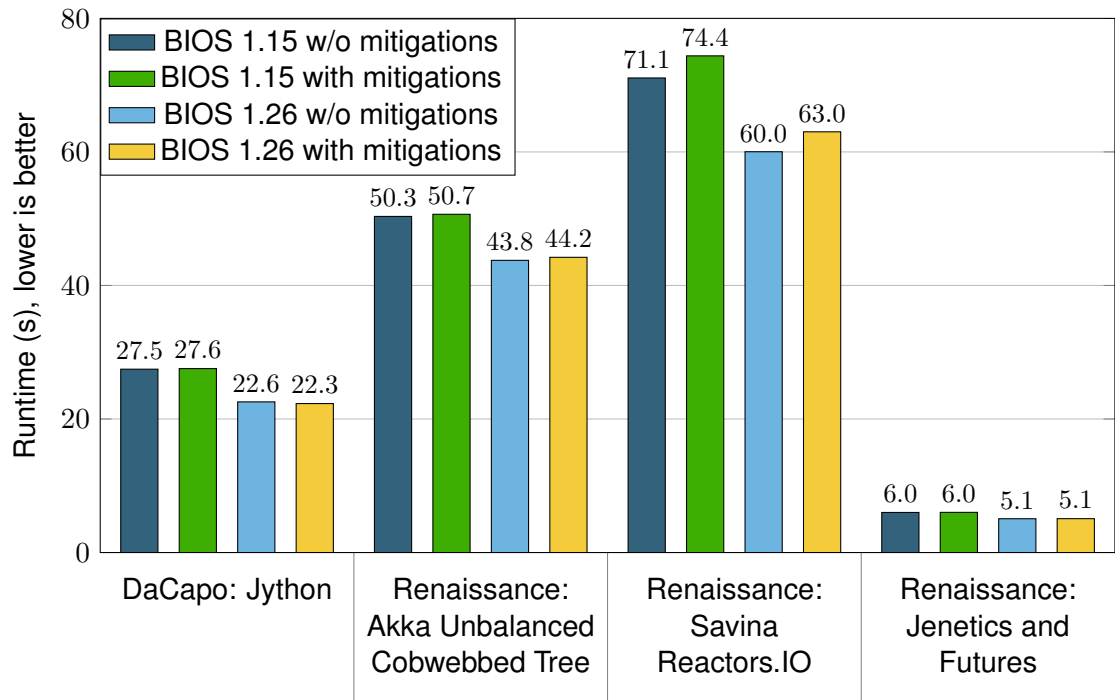


**Figure 7.4.** Random and sequential read and write speeds of DCU40's 16 GB eMMC memory with different BIOS versions, with and without mitigations



**Figure 7.5.** Random and sequential input and output operations of DCU40's 16 GB eMMC memory with different BIOS versions, with and without mitigations





**Figure 7.6.** Java tests with different BIOS versions, with and without mitigations

## 8 CONCLUSIONS AND OUTLOOK

In this work a build environment was set up, which is capable of creating a custom linux distribution for Teleste's train information system devices. The target device in this work was a single device called DCU40, however, at time of writing, work has already begun for setting up the Yocto environment for other Teleste's devices such as an ARM based network audio amplifier, which is the key component in announcement systems. The original objective of this work was a proof of concept image for DCU40, and after that more features were added to the build environment. The build environment itself is a Git superproject with other repositories as submodules, and it takes around 600 MB of disk space, and it consists of 14300 individual files. 70 new recipes were written for DCU40 and Teleste's platform components. The cache which BitBake uses to accelerate subsequent builds was set up on Teleste's company network share, and the disk space it takes is 40 GB for downloads and 80 GB for the shared state cache. In Teleste's company network, it is possible to run a full build for DCU40 in about 10 minutes on a company laptop when using the shared state cache. In the future, the build environment will be integrated to a CI server such as Jenkins.

With this build environment, it is possible to do more frequent build iterations than with Teleste's previous image creation tool Imagetools and be able to modify the source code of every software package. New features such as an automatic virtual machine creation for VirtualBox and Qemu were found to be useful in fast build iterations, which were related to non-hardware software running on DCU40. To further improve development on virtual platforms, software components could be written which would only load on virtualized environments so that DCU40's hardware component functionalities could be emulated. Network boot automation was a feature which made build and test iterations faster and this feature could also be developed further so that DCU40's internal eMMC memory could be flashed over the network. With the flexible partition scheme implementation of OpenEmbedded's Wic, more partitions could be set up on the internal eMMC memory of DCU40 for full remote updates of whole partitions while having a recovery partition, for example.

A comparison was done between Teleste's Imagetools and Yocto's BitBake even though they differ a lot. However, if this build environment would replace Imagetools, the created images would be fundamentally different since the used distribution would not anymore be Debian, but Teleste's own custom Linux distribution named Sky Blue. Testing was done related to image creation and boot-up times, and since the BIOS of Seco's processor card was updated, which enabled fast USB 3 for live USB drives, it also introduced Meltdown and Spectre vulnerability fixes, which can cause performance degradation. Testing was done to find out the severity of this performance degradation, but it was concluded that more testing would be needed to fully analyze the actual performance impact.

The conclusion for this work is that The Yocto Project can be successfully used in a company environment as a flexible build environment for Teleste's train information system devices.

## REFERENCES

- [1] *Yocto Project Mega-Manual*. Version 3.1. 2020. URL: <http://www.yoctoproject.org/docs/3.1/mega-manual/mega-manual.html> (visited on 06/13/2020).
- [2] Purdie, R., Larson, C. and Blundell, P. *BitBake User Manual*. Version 3.1. 2020. URL: <https://www.yoctoproject.org/docs/3.1/bitbake-user-manual/bitbake-user-manual.html> (visited on 08/17/2020).
- [3] *Yocto Project Overview and Concepts Manual*. Version 3.1. 2020. URL: <https://www.yoctoproject.org/docs/3.1/overview-manual/overview-manual.html> (visited on 08/17/2020).
- [4] Streif, R. J. *Embedded Linux systems with the Yocto project*. eng. Boston: Pearson Education, Inc., 2016. ISBN: 978-0-13-344324-0.
- [5] *Yocto Project Reference Manual*. Version 3.1. 2020. URL: <https://www.yoctoproject.org/docs/3.1/ref-manual/ref-manual.html> (visited on 10/06/2020).
- [6] *Toaster User Manual*. Version 3.1. 2020. URL: <https://www.yoctoproject.org/docs/3.1/toaster-manual/toaster-manual.html> (visited on 02/18/2021).
- [7] Chacon, S. and Straub, B. *Pro Git*. 2014. URL: <https://www.git-scm.com/book/en/v2>.
- [8] Atlassian. *Gitflow Workflow*. 2020. URL: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow> (visited on 01/26/2020).
- [9] *Seco BSP for Yocto*. Version 1.3. 2018. URL: [https://www.seco.com/Software/BayTrail/Yocto/Yocto\\_Pyro\\_Seco\\_BSP\\_v1.0.zip](https://www.seco.com/Software/BayTrail/Yocto/Yocto_Pyro_Seco_BSP_v1.0.zip) (visited on 08/05/2020).
- [10] *Debate Init System To Use*. 2015. URL: <https://wiki.debian.org/Debate/initssystem/> (visited on 08/05/2020).
- [11] *systemd-boot*. URL: <https://www.freedesktop.org/software/systemd/man/systemd-boot.html> (visited on 10/12/2020).
- [12] *GNU GRUB*. URL: <https://www.gnu.org/software/grub/manual/grub/grub.html> (visited on 10/12/2020).
- [13] Richardson, B. *"Last Mile" Barriers to Removing Legacy BIOS*. 2017. URL: [https://www.uefi.org/sites/default/files/resources/Brian\\_Richardson\\_Intel\\_Final.pdf](https://www.uefi.org/sites/default/files/resources/Brian_Richardson_Intel_Final.pdf) (visited on 10/12/2020).
- [14] *Yocto Project Linux Kernel Development Manual*. Version 3.1. 2020. URL: <https://www.yoctoproject.org/docs/3.1/kernel-dev/kernel-dev.html> (visited on 08/06/2020).
- [15] *systemd manual - Unit configuration*. 2020. URL: <https://www.freedesktop.org/software/systemd/man/systemd.unit.html> (visited on 08/14/2020).

- [16] *Linux Magic System Request Key Hacks*. 2020. URL: <https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html> (visited on 08/11/2020).
- [17] *Sunsetting Python 2*. 2020. URL: <https://www.python.org/doc/sunset-python-2/> (visited on 08/11/2020).
- [18] *Dynamic Host Configuration Protocol (DHCP) Options for the Intel Preboot eXecution Environment (PXE)*. 2006. URL: <https://tools.ietf.org/html/rfc4578> (visited on 11/02/2020).
- [19] *iPXE Command reference*. 2016. URL: <https://ipxe.org/cmd> (visited on 11/02/2020).
- [20] *What is Maven?* 2017. URL: <http://maven.apache.org/index.html> (visited on 10/20/2020).
- [21] Bityutskiy, A. *BMAP Tools*. 2020. URL: <https://github.com/intel/bmap-tools> (visited on 01/19/2021).
- [22] *Meltdown and Spectre, Vulnerabilities in modern computers leak passwords and sensitive data*. 2018. URL: <https://meltdownattack.com/> (visited on 01/05/2021).
- [23] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y. and Hamburg, M. *Meltdown: Reading Kernel Memory from User Space*. 2018. URL: <https://meltdownattack.com/meltdown.pdf> (visited on 01/05/2021).
- [24] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M. and Yarom, Y. *Spectre Attacks: Exploiting Speculative Execution*. 2019. URL: <https://spectreattack.com/spectre.pdf> (visited on 01/05/2021).