



Marko Toivanen

OpenDataPlane (ODP) as a Part of a Linux Operating System Image Built with Yocto Project

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

31 May 2022

Abstract

Author: Marko Toivanen
Title: OpenDataPlane (ODP) as a Part of a Linux Operating System Image Built with Yocto Project
Number of Pages: 30 pages
Date: 31 May 2022

Degree: Bachelor of Engineering
Degree Programme: Information Technology
Professional Major: Internet of Things and Cloud Computing
Supervisors: Janne Salonen, Head of School

The purpose of the thesis project was to provide a simple way to include OpenDataPlane in operating system images built from scratch. The goal was to create a build environment, in which only minimal changes are needed for when the target platform changes.

For the base operating system and the build environment, the Yocto project and its reference Linux distribution was chosen. The implementation was tested on three different platforms, from which two were physical systems on a chip and one being an emulated target system.

The expected outcome was met. However, the two physical target systems were not compatible to be used with ODP with full functionality out of the box. In the virtualized target system, the ODP was functional to the expected extent.

In conclusion, the project goal was met. While the physical target platforms were not compatible with ODP, it was proven that the build environment created is able to compile and include an operating system image with the functional ODP included.

Keywords: Yocto Project, OpenDataPlane, Embedded System, Operating System, Linux

Tiivistelmä

Tekijä:	Marko Toivanen
Otsikko:	OpenDataPlane (ODP) osana Linux käyttöjärjestelmäkuvaan Yocto Projectin avulla
Sivumäärä:	30 sivua
Aika:	31.5.2022
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Internet of Things and Cloud Computing
Ohjaajat:	Osaamisaluepäällikkö Janne Salonen

Insinööriyön tarkoituksena oli luoda ympäristö, jonka työkaluilla mahdollistetaan OpenDataPlane kiihdytysrajapinnan sulauttaminen rakennettuun käyttöjärjestelmäkuvaan. Tavoitteena oli ympäristö, jossa käyttöjärjestelmäkuvan luominen eri alustoille onnistuu mahdollisimman vähillä ympäristön konfiguraatiomuutoksilla.

Käyttöjärjestelmän rakennusympäristöksi valikoitui Yocto Project, jossa käyttöjärjestelmä on Linuxiin pohjautuva Yocto Projectin referenssidistribuutio, Poky. Lopullinen implementaatio testattiin kolmella kohdealustalla, kahdella sulautetulla järjestelmällä ja yhdellä virtualisoidulla kohdeympäristöllä.

Varsinaisena insinööriyönä luotiin käyttöjärjestelmän rakennusympäristö, jossa käyttöjärjestelmä rakennettiin. Luotuun käyttöjärjestelmäkuvaan sulautettiin OpenDataPlane kiihdytysrajapinta Yocto reseptin avulla, jonka kautta haluttu ohjelmisto käännettiin kohdealustalle käyttöjärjestelmän rakennuksen yhteydessä.

Insinööriyössä saavutettiin odotettu tavoite. Käyttöjärjestelmäkuva luotiin onnistuneesti kolmelle erilliselle kohdealustalle. OpenDataPlane rajapinnan toiminnallisuus testattiin kaikilla eri alustoilla. Vaikka testauksessa todettiin, että OpenDataPlane ohjelmisto ei ole täysin yhteensopiva kahden käytetyn sulautetun järjestelmän prosessoriarkkitehtuurin kanssa, projektin varsinainen tavoite silti saavutettiin.

Työn lopputulosta voidaan käyttää viitteenä samankaltaisen ympäristön luomiseen. Käyttöjärjestelmäkuvaan liitettävien ohjelmistojen kirjo on käytännössä loputon, ja valmiiseen ympäristöön on helppo lisätä haluttuja ohjelmistoja Yocto reseptien muodossa.

Avainsanat:	Yocto Project, OpenDataPlane, Sulautettu järjestelmä, Käyttöjärjestelmä, Linux
-------------	--

Contents

1	Introduction	1
2	OpenDataPlane	2
3	Yocto Project	4
4	Build infrastructure implementation	6
4.1	Virtual machine	6
4.2	Yocto build system	7
4.3	Build configuration and recipes	11
4.4	BeagleBone Black and TFTP boot	15
4.5	Raspberry Pi and micro-SD card boot	22
4.6	Nested QEMU virtual machine boot	25
5	Conclusion	27
	References	29

1 Introduction

A typical embedded system requires an operating system with a minimal overhead. Basically this means that the operating system in question should take as little resources as possible from the application the system is being used for. Building a minimal operating system with only the needed components can be laborious, especially if the same image is to be provisioned for multiple target devices and processor architectures.

The goal of the thesis project is to build an environment for providing minimal Linux operating system images. The operating system image built is to include the OpenDataPlane hardware acceleration framework. The environment should be capable of building the operating system image in question for multiple different target architectures with minimum number of configuration changes.

This project takes different boot flows and procedures into an account, as setting up the boot environment for each system is also an important part of a practical build environment. The boot file delivery infrastructure is a requirement for a functional build system, and these methods and configurations will be described in detail.

The requirement for a successful implementation is to be able to build an operating system image with the ODP framework included for various target platforms. With the requirements met, one could automate software integration process of the software delivery by referring to the build environment implementation made.

2 OpenDataPlane

The OpenDataPlane project, ODP for short, was introduced in 2013 by a non-profit engineering organization Linaro. The initial goal was to create a data plane application framework to support software mobility across different networking SoCs. [1.]

ODP is aiming to provide a hardware acceleration layer for high performance networking applications. It is a library built specifically to the target ISA with GNU build tools. The implementation is intended to be run on top of Linux, therefore rendering it cross compatible between different end-system architectures. The ODP implements an API to be integrated into the software for it to achieve the best possible portability, scalability and maintainability. One of the focus areas of the development is the scalability of scheduling across the full range of usable CPU resources, along with low memory footprint. [2.]

The ODP API has all the needed parts for manipulating and managing the data flow. The overall flow is depicted in figure 1.

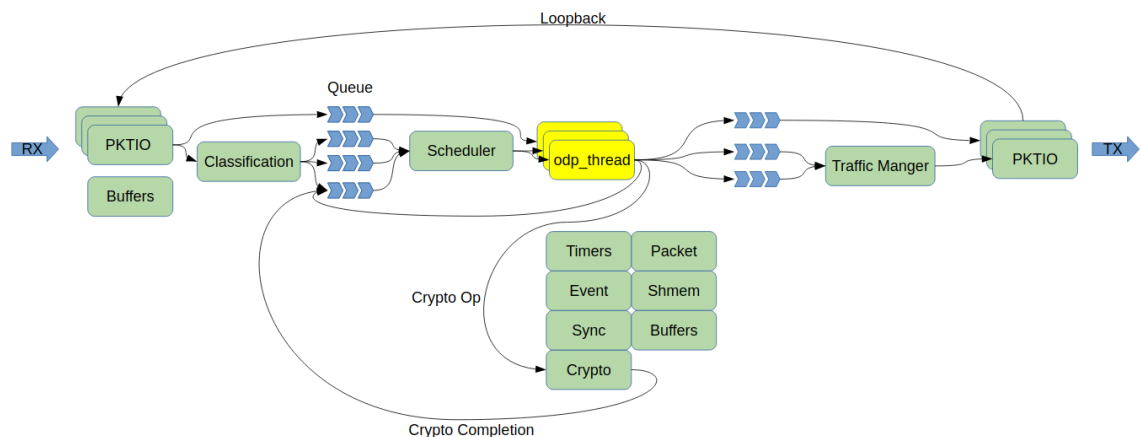


Figure 1: Typical data flow of an ODP application. [3].

As seen in the figure 1, the data is processed through the classification and scheduling, along with traffic managing for a Quality of Service feature. The overall system consists of different types of queues and scheduling methods. The data sent to the flow may loop around the system multiple times before it is sent

for output. [2.] Parts of the flow can also be replaced with specialized hardware to offload the processing from the main CPU. [4]

ODP is an open-source project with a BSD-3 license. The included BSD-3 license means that the source code and built binaries can be distributed freely, as long as the license is carried over. [5] By the definition of open-source, the source code is accessible to anyone and is licensed so that it does not restrict its usage to a specific product or platform, nor does it require a monetary compensation for its use or redistribution. [6]

3 Yocto Project

The Yocto Project was launched in April 2011 by a non-profit organization, The Linux Foundation. The goal of the Yocto Project is to provide a toolchain to develop an embedded Linux operating system distributions. [7.]

The reference Linux distribution of Yocto Project is Poky. Poky is not a ready-made Linux distribution one can simply download and run. Poky is a set of instructions, known as metadata layers and configuration files, for the toolchain to build and compile a Linux distribution. There are various types of predefined images that can be built, from *core-image-minimal*, for a minimal bootable distribution, up to *core-image-sato-sdk*, for a Linux Standard Base compliant operating system with development environment and graphical user interface capabilities. The images can be further extended with additional packages from metadata layers by appending the wanted packages to the image build configuration. The packages in the metadata layers are typically in a form of a source code and its compilation instructions. The package is compiled at the build time from the source code to the target architecture. The system is also capable of downloading the source code files required from, for example, remote Git repositories. [8.]

A typical metadata layer consist of configuration files and package recipes. The configuration files define the build process, along with the default tasks for certain recipe instructions. The recipes are a concept of the build instructions for a package. These recipes, identified with the file extension of *.bb*, contain the information, build time and runtime dependencies, source code location, along with the configuration, compilation, and install instructions of the package in question. A single recipe can have multiple flavors, which can be used to change the way the package is to be handled. A recipe can also require other recipes to be built and included in the resulting image. [9, section 1.3.]

Along with Poky, another core component of the Yocto build system is Bitbake. Bitbake is a tool responsible for fetching, scheduling, compiling, and verifying the

build recipes. Bitbake is capable of handling the cross-compilation of the source files for another architecture, along with different variations of the package in question. Basically, Bitbake takes the metadata layers from the configuration file, ranks those by priority level, and utilizes the recipes within the layer for a build. This introduces an ability to extend and override existing metadata layers with, for example, additional platform specific metadata layers. [9, sections 1.1, 1.2.]

4 Build infrastructure implementation

The goal of the project was to implement a Yocto build environment, in which a Linux image is built with ODP. The image is then to be run and tested to verify ODP functionality.

The Yocto build environment will be set up in a virtual machine. The host system for the virtual machine is a 2013 manufactured high-end laptop with four core AMD APU processor, eight gigabytes of DDR3 RAM and a 256 gigabyte SSD. The laptop is running a Pop!_OS 21.04 Linux distribution as its operating system.

As the target systems, two embedded systems are used, along with an emulated target system. The two embedded systems are a BeagleBone Black and a Raspberry Pi model B. Both of these are equipped with 32-bit ARM processors. The virtualized target system is to use an emulated 64-bit processor.

4.1 Virtual machine

The hypervisor for the virtual machine provision is QEMU with KVM. This is managed through libvirt based graphical user interface, Virtual Machine Manager also known as virt-manager.

The virtual machine operating system was chosen to be Ubuntu 20.04.3 LTS Linux distribution. Ubuntu has a large variety of software to install in its software repositories, and it is extendable with the Personal Package Archives, PPAs. Ubuntu 20.04 LTS is also one of the officially supported Linux distributions of the Yocto Project [10, chapter 1.1].

This report does not go through the virtual machine creation process, as it is optional, and the same results can be achieved without the virtualized environment. The virtualization of the build system was done only for the author's convenience, as it provides needed resource management and virtual machine

snapshots.

The Ubuntu 20.04 LTS virtual machine intended for Yocto build system was created with two virtual network interface cards. First one was mapped as a NAT to the wireless network interface of the host laptop to provide internet access for the virtual machine. The second virtual NIC was created as a Macvtap device for it to hook itself to the physical Ethernet port of the host laptop.

4.2 Yocto build system

The Yocto build system requires its components to be cloned from Git repositories. For this, the *git* package is needed. The *git* was installed to the Ubuntu virtual machine with a command `sudo apt install git`, which calls the *apt* package manager as superuser with arguments to install the *git* package.

The build system files are to be placed to the path */yocto* for convenience. The directory is created with a `sudo mkdir /yocto` command. As the location is at the root of the file system, the directory permissions need to be adjusted for a normal user to be able to utilize it. This was done with `sudo chown -R user:user /yocto` command, which instructs to change the owner to user *user* of *user* group recursively for the directory */yocto*.

In the */yocto* directory, the *Poky* reference Linux distribution is downloaded. This is done with `git clone git://git.yoctoproject.org/poky` command, instructing to use *git* to clone the contents of the "poky" Git repository. With this, there now is */yocto/poky* directory containing the basic build files and scripts.

By using the directory structure above, the environment is easy to manage. The root directory name *yocto* represents the intention of the directory. Having directory *poky* under */yocto* is a future-proof solution in case multiple different build environments would be needed. With this approach, the different build environments would be situated under */yocto* next to the *poky* subdirectory.

A build environment is initialized by sourcing the *oe-init-build-env* script with

source oe-init-build-env command. This command runs the script and keeps the environment variables persistent for the current shell session. The output of the *source oe-init-build-env* can be seen in figure 2.

```

yoctouser@yoctovm: /yocto/poky/build x yoctouser@yoctovm: /yocto/poky/build x
Receiving objects: 100% (560466/560466), 182.03 MiB | 12.24 MiB/s, done.
Resolving deltas: 100% (406820/406820), done.
yoctouser@yoctovm: /yocto$ cd poky
yoctouser@yoctovm: /yocto/poky$ source oe-init-build-env
You had no conf/local.conf file. This configuration file has therefore been
created for you with some default values. You may wish to edit it to, for
example, select a different MACHINE (target hardware). See conf/local.conf
for more information as common configuration options are commented.

You had no conf/bblayers.conf file. This configuration file has therefore been
created for you with some default values. To add additional metadata layers
into your configuration please add entries to conf/bblayers.conf.

The Yocto Project has extensive documentation about OE including a reference
manual which can be found at:
  https://docs.yoctoproject.org

For more information about OpenEmbedded see the website:
  https://www.openembedded.org/

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

You can now run 'bitbake <target>'

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

You can also run generated qemu images with a command like 'runqemu qemu86-64'.

Other commonly useful commands are:
- 'devtool' and 'recipetool' handle common recipe tasks
- 'bitbake-layers' handles common layer tasks
- 'oe-pkgdata-util' handles common target package tasks
yoctouser@yoctovm: /yocto/poky/build$

```

Figure 2: Output of the `source oe-init-build-env` command.

By running the `source oe-init-build-env` script, two files are created in the `conf` directory of the build environment, as can be seen in figure 2. The `local.conf` is used for configuration of the build of the Linux distribution, defining for example the target system this build is targeted to. The `bblayers.conf` is used to append the metadata, such as packages, to the image to be built.

With the build environment and the default template settings set, a build can be attempted. This is done by issuing a `bitbake core-image-minimal` command. This is a command for building a minimal image with necessary components for a Linux distribution to run. Running this for the first time in a system with no additional configuration done may lead to a situation, in which some dependencies are missing. This kind of situation is to be seen in the figure 3.

```

yoctouser@yoctovm:/yocto/poky$ source oe-init-build-env
### Shell environment set up for builds. ###
You can now run 'bitbake <target>'

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

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

Other commonly useful commands are:
- 'devtool' and 'recipetool' handle common recipe tasks
- 'bitbake-layers' handles common layer tasks
- 'oe-pkgdata-util' handles common target package tasks
yoctouser@yoctovm:/yocto/poky/build$ bitbake core-image-minimal
ERROR: The following required tools (as specified by HOSTTOOLS) appear to be unavailable in PATH, please
install them in order to proceed:
  ar as chrpath diffstat g++ gawk gcc ld make nm objcopy objdump pzdtd ranlib readelf rpcgen strings str
ip zstd
yoctouser@yoctovm:/yocto/poky/build$

```

Figure 3: Output of the *bitbake core-image-minimal* command being run at the first time.

As seen in the figure 3, multiple required packages are missing from the system attempting the build. Most of these tools are available in the *build-essential* package, which is a toolkit for compiling C and C++ software. From the list of these missing required packages, *chrpatch*, *diffstat*, *gawk*, and *zstd* are not in the *build-essential* package and need to be installed individually. With this, the command to install all the missing requirements mentioned is *sudo apt install build-essential chrpatch diffstat gawk zstd*. With these installed, running the *bitbake core-image-minimal* reports that the *distutils* module is further required for the Python 3, this was also installed with *sudo apt install python3-distutils* command. An expected *bitbake core-image-minimal* output can be seen in figure 4.

```

yoctouser@yoctovm:/yocto/poky/build$ bitbake core-image-minimal
Loading cache: 100% |#####| Time: 0:00:00
Loaded 98 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:01:20
Parsing of 829 .bb files complete (64 cached, 765 parsed). 1472 targets, 44 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "1.52.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING     = "ubuntu-20.04"
TARGET_SYS          = "x86_64-poky-linux"
MACHINE              = "qemux86-64"
DISTRO               = "poky"
DISTRO_VERSION       = "3.4"
TUNE_FEATURES        = "m64 core2"
TARGET_FPU           = ""
meta
meta-poky
meta-yocto-bsp       = "master:12ca5f7209126ea6e07b3ba93fbd31065971454"

NOTE: Fetching univariate binary shim http://downloads.yoctoproject.org/releases/univariate/3.4/x86_64-native-sdk-libc.tar.xz;sha256sum=126f4f7f6f21084ee140dac3eb4c536b963837826b7c38599db0b512c3377ba2 (will check PREMIRRORS first)
Initialising tasks: 100% |#####| Time: 0:00:05
Sstate summary: Wanted 1186 Local 0 Network 0 Missed 1186 Current 0 (0% match, 0% complete)
NOTE: Executing Tasks
Currently 3 running tasks (1186 of 1186/210 of 3103) 6% |##
0: libtool-native-2.4.6-r0 do_configure - 38s (pid 61091)
1: openssl-native-1.1.1l-r0 do_compile - 29s (pid 61403)
2: perl-native-5.34.0-r0 do_unpack - 0s (pid 67566)

```

Figure 4: Output of the `bitbake core-image-minimal` command when a build is in progress.

After the required dependencies are installed, a build with `bitbake core-image-minimal` is again tested. As can be seen in figure 4, the build is working and the build tasks are being processed in the output. The initial build with a default configuration took approximately nine hours.

4.3 Build configuration and recipes

To include an ODP compilation and further the built binaries to the Yocto image, a Yocto recipe for the process is needed. For the recipe, a new metadata layer will be created. A typical metadata-layer creation process is depicted in figure 5.


```

yoctouser@yoctovm:/yocto/poky$ source oe-init-build-env

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

You can now run 'bitbake <target>'

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

You can also run generated qemu images with a command like 'runqemu qemu86'

Other commonly useful commands are:
- 'devtool' and 'recipetool' handle common recipe tasks
- 'bitbake-layers' handles common layer tasks
- 'oe-pkgdata-util' handles common target package tasks
yoctouser@yoctovm:/yocto/poky/build$ bitbake-layers create-layer ../meta-odp
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer ../meta-odp'
yoctouser@yoctovm:/yocto/poky/build$ bitbake-layers add-layer ../meta-odp
NOTE: Starting bitbake server...
yoctouser@yoctovm:/yocto/poky/build$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                path                                priority
=====
meta                 /yocto/poky/meta                   5
meta-poky            /yocto/poky/meta-poky              5
meta-yocto-bsp       /yocto/poky/meta-yocto-bsp         5
meta-odp             /yocto/poky/meta-odp               6
yoctouser@yoctovm:/yocto/poky/build$ █

```

Figure 5: A metadata layer implementation process.

As the build environment is already set with `source oe-init-build-env` command, the command `bitbake-layers` can be used to create the new metadata layer for ODP and its dependencies. This is done with a `bitbake-layers create-layer ../meta-odp` command. After the layer is created, it can be added to the build with `bitbake-layers add-layer ../meta-odp` command. With `bitbake-layers show-layers` command, it can be verified that the new layer was created, and has now been added to the Yocto build. As can be seen in the figure 5, the new `meta-odp` layer took a higher priority in comparison, as in to ensure that its contents are preferred over the default layers.

The ODP build recipe can now be implemented to the `meta-odp` metadata layer. In the `meta-odp` metadata layer directory, being `/yocto/poky/meta-odp` in this case, a new directory `odp` is created to contain the recipes. In this

`/yocto/poky/meta-odp/odp` directory, a new file is created with a name of `odp_git.bb` which is the recipe for the ODP.

```

1 DESCRIPTION = "Pull and compile ODP"
2 LICENSE = "CLOSED"
3
4 RDEPENDS:${PN} += "openssl libconfig packagegroup-core-buildessential
   flex zlib coreutils bash"
5 DEPENDS += "openssl libconfig packagegroup-core-buildessential
   flex-native zlib coreutils-native bash"
6
7 PV = "1.32.0.0+git${SRCPV}"
8 SRCREV = "ec78e5305c387cef47be7479d2b9f85482e20b40"
9 SRC_URI = "git://github.com/OpenDataPlane/odp"
10
11 S = "${WORKDIR}/git"
12
13 inherit autotools pkgconfig
14
15 EXTRA_OECONF = 'CFLAGS="-O0 -Wno-error"'

```

Listing 1: Recipe to download ODP from GitHub, compile, and install it.

On line number two in listing 1, it can be seen that the value of `"LICENSE"` is set to `"CLOSED"`. This is to avoid possible errors caused by the checksum verification in the recipe in the development environment. On line number four, there is `"RDEPENDS"` variable, which contains the runtime dependencies of the recipe, as in the additional packages required by the ODP to be present in the built image. As can be seen the build time dependencies, the value of `DEPENDS`, the build time and runtime dependencies are very similar. Some build time dependencies however use the `"-native"` flag to indicate that the packages can be built with host machine architecture to speed up the build process by avoiding build time cross compilation. All but package `libconfig` can be built from the metadata layers included, the `libconfig` package needs its own recipe to be implemented.

Unfolding the listing 1 further, there are variables to define where the recipe needs to get the build files from. Starting from line number seven, there is the version to be downloaded (the value of `PV`), the defining commit hash for verification (the value of `SRCREV`) and the URL of the remote repository in question (being the value of `SRC_URI`). Downloading a Git repository inside the recipe, it is needed to

set the value of build source variable *S* as the *git* subdirectory of the working directory, which is where the source files are downloaded to by the recipe.

For the build instructions in listing 1, the packages *autotools* and *pkgconfig* are used. These will take care of the build automatically per the *Makefile* build file contents of the ODP repository. As additional build instructions, the value of *EXTRA_OECONF* passed with the *CFLAGS* compilation environment arguments, there are *-O0* to disable the compilation optimization and *-Wno-error* to ignore all *WARNING* level errors while compiling. Disabling the optimization ensures that the compiler does not implement its own optimizations in attempt of making the ODP faster, as this might potentially cause unwanted behavior. Ignoring compilation warnings ensures that the ODP is build even in case there would be build time warnings, which would potentially be caused by compiling different architectures while using some native packages in build time.

```

1 DESCRIPTION = "Pull and compile libconfig"
2 LICENSE = "CLOSED"
3
4 DEPENDS += "bison-native"
5
6 SRCREV = "${AUTOREV}"
7 SRC_URI = "git://github.com/hyperrealm/libconfig"
8
9 S = "${WORKDIR}/git"
10
11 inherit autotools-brokensep pkgconfig

```

Listing 2: The recipe for *libconfig* package.

The recipe for *libconfig*, which can be seen in the listing 2, is very similar to the ODP recipe in listing 1. The *libconfig* only requires package *bison* during build time. The *bison* package is available in the metadata layers already. For the source revision of the remote Git repository, a variable of *\${AUTOREV}* is used for the recipe to automatically use the latest version available. Also, instead of *autotools*, the *autotools-brokensep* is used to work around the broken out-of-tree build of the *libconfig* [10, chapter 5.3].

To prepare a build for the BeagleBone Black, the packages wanted into the image,

along with the target system machine, are to be appended in the *local.conf* configuration file in the *conf* directory. The value of *MACHINE* variable is set as *MACHINE ?= "beaglebone-yocto"* in the configuration file. Along with that, as new packages need to be installed to the built operating system image, a line *IMAGE_INSTALL:append = " openssl libconfig packagegroup-core-buildessential flex zlib coreutils odp"* is also needed to include all the runtime dependencies of the ODP. After this, the *bitbake core-image-minimal* command is again used to build the operating system image for BeagleBone Black.

4.4 BeagleBone Black and TFTP boot

The BeagleBone Black requires three cables for required functionality for the project. The USB-UART cable (FTDI TTL232R-3V3) was connected between the computer and the UART header of the board. Also, a generic miniUSB-USB cable was connected between the miniUSB port of the board and the host machine. Along with that, a CAT6 Ethernet cable was connected between the board and the host machine. The UART provides console interface, miniUSB is a for power and USB data connection, while the Ethernet is used for the TFTP boot.

To test out the serial connection from the virtual machine to the BeagleBone Black board, the program *minicom* was installed with the package manager. For this to work, the current user of the virtual machine needs to be in the *dialout* group, this can be achieved with a *sudo adduser \$USER dialout* command. From the settings of *minicom*, the *Hardware Flow Control* settings needs to be disabled for the output to work properly. This was changed within the *minicom* by pressing *CTRL+A* and then pressing *O* to open the *Configuration* screen, and selecting the option in *Hardware Flow Control* under *Serial Port Setup*. The serial connection can be tested with *minicom -D /dev/ttyUSB0* command.

```
1 bootdelay=3
2 client_ip=192.168.100.141
3 server_ip=192.168.100.140
4 gw_ip=192.168.100.1
5 netmask=255.255.255.0
6 hostname=bbb
7 device=eth0
8 autoconf=off
9 root_dir=/exports/rootfs
10 nfs_options=,vers=3
```

Listing 3: The new contents of /uEnv.txt in BeagleBone Black Debian file system.

The BeagleBone Black has a preinstalled Debian Linux distribution in its eMMC persistent storage. To be able to perform a TFTP boot, as in fetching the boot files needed over a network connection, a TFTP boot environment needs to be configured in the uBoot bootloader of the BeagleBone Black. The uBoot environment variables are set in the *uEnv.txt* file in the Debian file system root directory. The variables are set per the listing 3. The build environment acts as the server, and its network configuration is depicted in the figure 6.

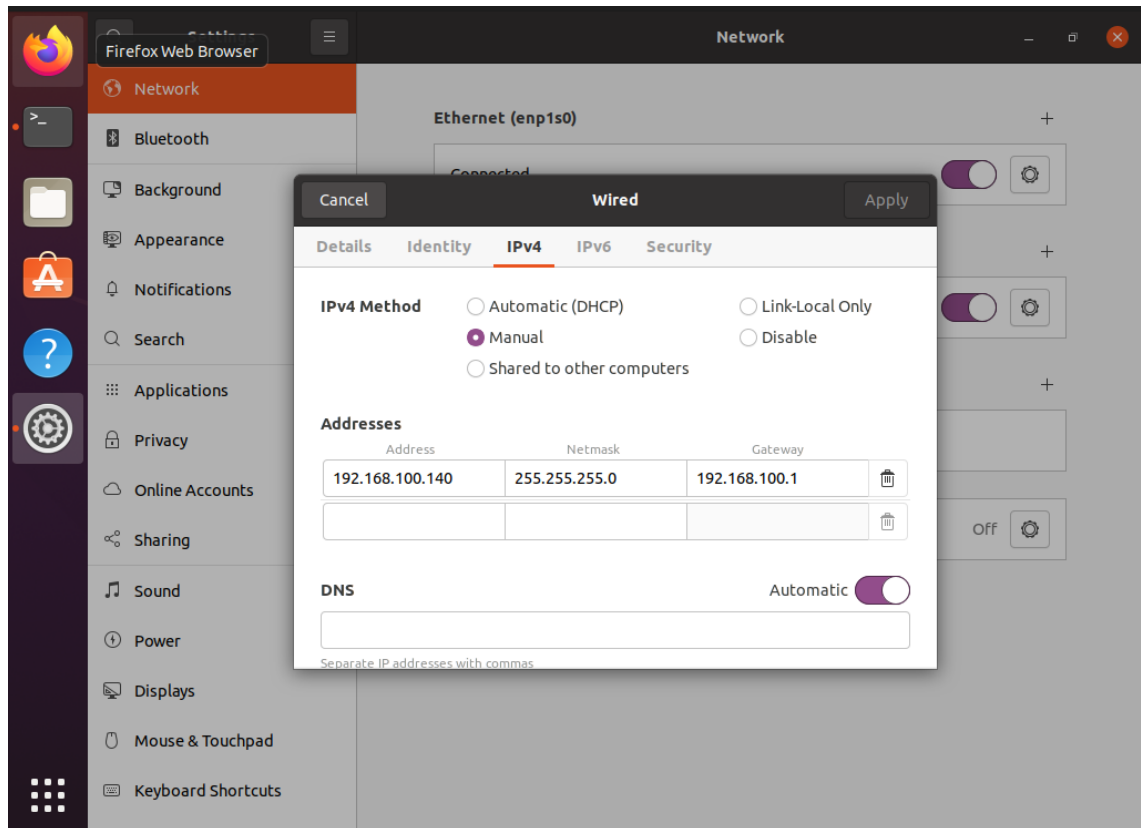


Figure 6: The Ethernet network interface settings of Gnome in the virtual machine. Values are set per the settings defined in `uEnv.txt` of the BeagleBone Black.

As seen in figure 6, the virtual network card of the virtual machine is to be given a static address. The *IPv4 Method* setting is set to *Manual*. The IP address is set as `192.168.100.140` with the netmask of `255.255.255.0`. This is set as it is for the BeagleBone Black to be able to connect to the virtual machine network address. Note that the address used here is the address which was defined in the `uEnv.txt` in the listing 3 for `server_ip` value. The value of gateway is set as `192.168.100.1`, also to match the corresponding `uEnv.txt` value of `gw_ip`. With this, the network for the TFTP is established.

Next step is to set up the locations for file serving to the virtual machine. Using the `mkdir` command, directories up to `/tftpboot/dtbs` and `/exports/rootfs` are created. For both of these, the permissions are changed so that the user, which in this case is `yoctouser`, is set as the owner of these directories recursively. The change of owner was done with the `sudo chown -R yoctouser /tftpboot` and `sudo chown -R yoctouser /exports/rootfs` commands.

From the build done earlier, the files needed for booting up the BeagleBone Black are located in `/yocto/poky/build/tmp/deploy/images/beaglebone-yocto` directory. From there, the needed content is copied to the respective locations for the TFTP server and the NFS service to deliver those to the BeagleBone Black. The kernel image, as in the `zImage` file, is to be situated under `/tftpboot` directory, and the device tree file, `am335x-boneblack.dtb`, is transferred to the `/tftpboot/dtbs` directory. Also, the root file system itself, which is packed and compressed as `core-image-minimal-beaglebone-yocto.tar.bz2`, is extracted and unpacked into `/exports/rootfs/` directory using `sudo tar xf core-image-minimal-beaglebone-yocto.tar.bz2 -C /exports/rootfs/` command. As the BeagleBone Black already has a bootloader, this is all it needs to boot into the Linux distribution built.

The actual file serving system consists of two services. A `tftpd-hpa` is the TFTP server, and `nfs-kernel-server` acts as the NFS provider. While the Linux kernel and the device tree can be downloaded to the memory of BeagleBone Black once and used further from there, the lockstep style TFTP server can be used for those. As the root file system contains of constantly changing files which need to be synchronized for persistence, an NFS mount is a suitable solution.

The TFTP server was installed with a `sudo apt install tftpd-hpa` command, and the NFS was installed with a `sudo apt install nfs-kernel-server` command. As for the configuration for the TFTP server, simply pointing to the location of the files to be shared is enough. In the file `/etc/defaults/tftpd-hpa`, the `TFTP_DIRECTORY` variable was given a value of `/tftpboot`, resulting in the said line in the configuration being `TFTP_DIRECTORY="/tftpboot"`. For the NFS, the line `/exports/rootfs 192.168.100.141 (rw,no_root_squash,no_subtree_check)` was appended to the `/etc/exports` file, for it to share the `/exports/rootfs` directory with the given IP address and options. With this, the configuration of TFTP server and NFS is now done. The boot process of the BeagleBone Black, along with its utilization of the TFTP for the retrieving the boot files, is depicted in figure 7.

Poky reference Linux distribution image built.

To test that the ODP has been compiled and included in the Yocto image properly, the shared memory virtual device needs to be implemented and mounted as */dev/shm* block device. In the serial console of the BeagleBone Black, a command *mount -t tmpfs shmfs -o size=400M /dev/shm* is used. To unfold this command, a *shmfs* style of block is mounted as *tmpfs* type. The *tmpfs* is a partition type which uses the RAM of the system for the storage allocation. This mount is given the storage allocation size of 400 megabytes and is instructed to be mounted as */dev/shm*, a location which is a typical for shared memory block device. [11.]

After setting up the shared memory block device, the ODP functionality can be tested. As the ODP has been compiled into the image, simply running one of the included examples, *odp_hello*, is enough for a test. Running the *odp_hello* on the BeagleBone Black serial console, the terminal window is filled with the command output. The output of the *odp_hello* command is depicted in the figure 8.

```

yoctouser@yoctovm: /yocto/poky/build
root@beaglebone-yocto:~# odp_hello
WARN: cpu[0] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[1] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[2] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[3] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[4] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[5] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[6] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[7] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[8] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[9] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[10] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[11] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[12] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[13] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[14] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[15] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[16] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[17] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[18] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[19] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[20] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[21] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[22] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[23] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[24] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[25] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[26] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[27] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[28] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[29] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[30] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[31] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[32] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[33] uses default max frequency of 1400000000 Hz from config file
WARN: cpu[34] uses default max frequency of 1400000000 Hz from config file

```

Figure 8: Initial output of running `odp_hello` on the BeagleBone Black.

As can be seen in figure 8, first thing to appear in the output is 256 lines of warning messages. As the BeagleBone Black has a 32-bit ARM processor, an architecture which is not officially supported by ODP, this is to be expected. The ODP is unable to poll for the number of CPU cores and their frequencies, which results the warning messages to be printed for the ODP configuration default value of 256 CPU cores. For each of these cores, it is stated that the frequency of the said core cannot be determined, and the ODP configuration default value of 1.4 GHz is to be used.

Even with the `odp_hello` outputs warnings about the CPU, the basic ODP functionality is working as it should. The continuation of the output presented in figure 8 can be seen in the figure 9 below.

```

yoctouser@yoctovm: /yocto/poky/build
yoctouser@yoctovm: /yocto/po... x yoctouser@yoctovm: /yocto/po... x yoctouser@yoctovm: /yocto/po... x
Terminal t.base_align: 64
yoctouser@yoctovm: /yocto/po... x yoctouser@yoctovm: /yocto/po... x yoctouser@yoctovm: /yocto/po... x
Queue config:
queue_basic.max_queue_size: 8192
queue_basic.default_queue_size: 4096

Using scheduler 'basic'
Scheduler config:
sched_basic.prio_spread: 4
sched_basic.prio_spread_weight: 63
sched_basic.load_balance: 1
sched_basic.burst_size_default[] = 32 32 32 32 32 16 8 4
sched_basic.burst_size_max[] = 255 255 255 255 255 16 16 8
sched_basic.group_enable.all: 1
sched_basic.group_enable.worker: 1
sched_basic.group_enable.control: 1
dynamic load balance: ON

Packet I/O config:
pktio.pktin_frame_offset: 0

PKTIO: initialized loop interface.
PKTIO: initialized null interface.
PKTIO: initialized socket mmap, use export ODP_PKTIO_DISABLE_SOCKET_MMAP=1 to disable.
PKTIO: initialized socket mmsg, use export ODP_PKTIO_DISABLE_SOCKET_MMSG=1 to disable.
Hello world from CPU 0!
root@beaglebone-yocto:~# df -h
Filesystem                Size      Used Avail Use% Mounted on
192.168.100.140:/exports/rootfs  98G   56G   38G  60% /
devtmpfs                   236M     0   236M   0% /dev
tmpfs                       244M    76K   244M   1% /run
tmpfs                       244M    92K   244M   1% /var/volatile
shmfs                       400M     0   400M   0% /dev/shm
root@beaglebone-yocto:~#
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Offline | ttyUSB0

```

Figure 9: Working output of the `odp_hello` command, printing the usable features of the ODP, along with the hello message from CPU of index 0. The `df` command in the end shows that the `/dev/shm` is correctly mounted.

As seen in figure 9, the `odp_hello` prints the basic ODP features usable. Along with the features available, the `odp_hello` prints a hello message from the CPU of index 0. Typically, if the `odp_hello` binary would be executed multiple times, the CPU index the hello is coming from would change. Yet in the case of the ODP not being able to correctly poll the CPUs, the hello is always sent from the CPU of index 0 as the ODP interprets this being the only CPU available in this 32-bit ARM system.

4.5 Raspberry Pi and micro-SD card boot

To build a Yocto image for the Raspberry Pi, a `meta-raspberrypi` metadata layer is needed. As with getting the original `Poky` metadata layer, the `meta-raspberrypi` metadata layer can also be downloaded from the `git.yoctoproject.org` repository management system. This is done with `git clone git://git.yoctoproject.org/meta-raspberrypi` command. As with other metadata layers, the `meta-raspberrypi` metadata layer directory is to be placed in

/yocto/poky directory.

With *meta-raspberrypi* in its correct place, the metadata layer needs to be added to the bitbake layer configuration. As with the *meta-odp* metadata layer, this is done with *bitbake-layers add-layer* command. The addition can be verified with the *bitbake-layers show-layers* command. Also, in the instruction file in the *meta-raspberrypi* repository it is said that the *meta-openembedded* metadata layer is required for building an image for the Raspberry Pi. Therefore, the *meta-openembedded* metadata layer is downloaded with *git clone https://git.openembedded.org/meta-openembedded* command. Along with that, the branch of the local *meta-openembedded* repository needs to be changed to match the Poky distribution version used in the project, being *honister*. The branch is changed by running *git checkout honister* in the local copy of the *meta-openembedded* repository directory. This metadata layer also needs to be added to the build layers with the *bitbake-layers add-layer* command.

The target, as in the *MACHINE* variable, also needs to be changed to match the target the built Yocto image is going to be used in. In the *local.conf* configuration file of the build environment, the value of *MACHINE* variable is changed as *raspberrypi3* value. While the project board actually is *Raspberry Pi model 2 B*, the board is similar enough for this target to work.

With the layers and configuration ready for the build, the build can be started with *bitbake core-image-minimal* command. After the build finishes, the image can be found in *tmp/deploy/images/raspberrypi3* directory as *core-image-minimal-raspberrypi3.wic.bz2* file. This is a compressed container file, and needs to be decompressed. The container can be decompressed with the *bzip2* tool, with *bzip2 -d -f core-image-minimal-raspberrypi3.wic.bz2* command, which results in a *core-image-minimal-raspberrypi3.wic* file.

The image is then to be written in a micro SD card, which is attached to the micro SD card reader of the virtual machine host laptop and then redirected to the virtual machine. The image can be written to the micro SD card with the *rpi-imager* tool.

For the *Ubuntu 20.04.3 LTS*, the tool was installed with `snap install rpi-imager` command. After this, running `rpi-imager` command starts the graphical Raspberry Pi Imager program. The graphical user interface and an ongoing writing operation is demonstrated in the figure 10.

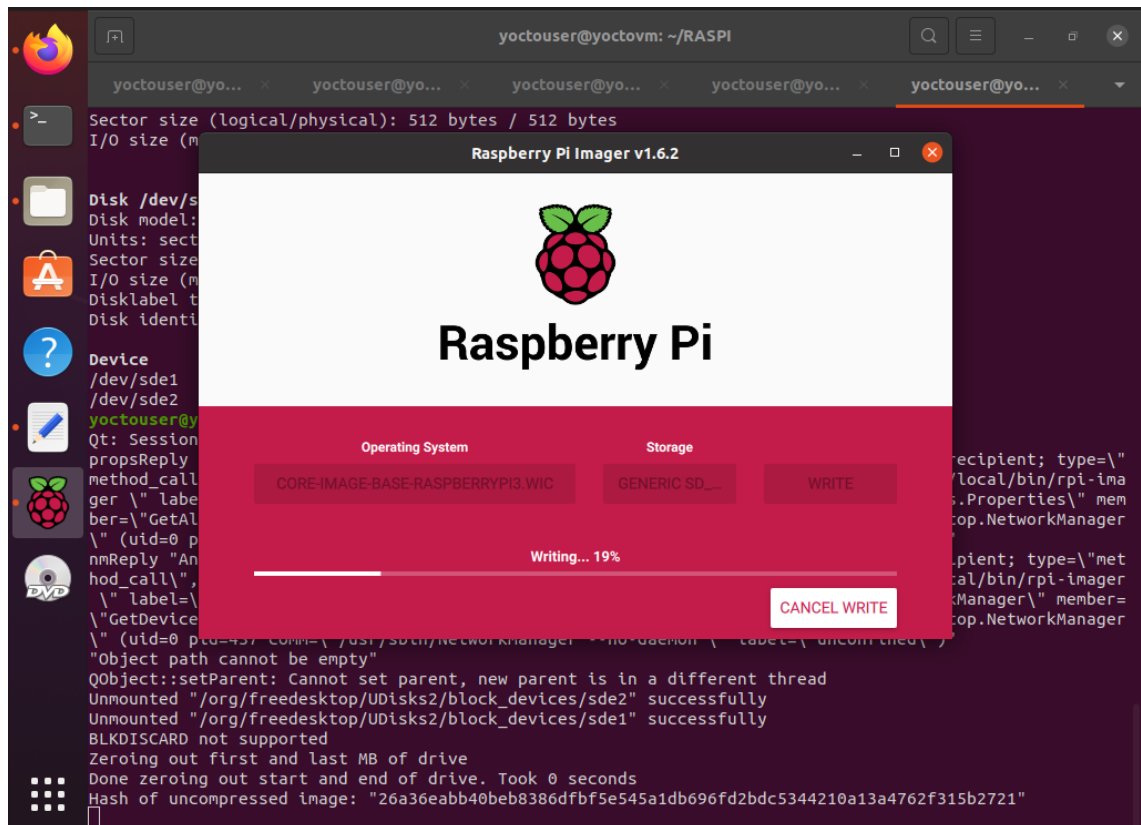


Figure 10: The graphical user interface of the `rpi-imager` tool.

In the Raspberry Pi Imager tool, the *Operating System* is set as the `core-image-minimal-raspberrypi3.wic` Yocto image generated, and the *Storage* is set as the micro SD card to be written. Clicking `WRITE` button writes the *Operating System* Yocto image to the micro SD card in question. A functional process of writing can be seen in figure 10.

With the image in the micro SD card, the micro SD card can be inserted to the Raspberry Pi 2 model B project board. After plugging in the power, the board boots up to the Yocto image. First thing to do is to again mount the `/dev/shm` with the `mount -t tmpfs shmfs -o size=400M /dev/shm` command. Running the `odp_hello` command results in the output similar to the case with BeagleBone Black, with warnings about using the configured CPU frequency default values for

not being able to properly poll the actual CPU. This also was to be expected, as similarly to BeagleBone Black, the Raspberry Pi 2 model B also has a 32-bit ARM processor. With this, it is safe to say that the ODP indeed does not seem to work properly on the 32-bit ARM boards, or at least not with the two boards tried.

4.6 Nested QEMU virtual machine boot

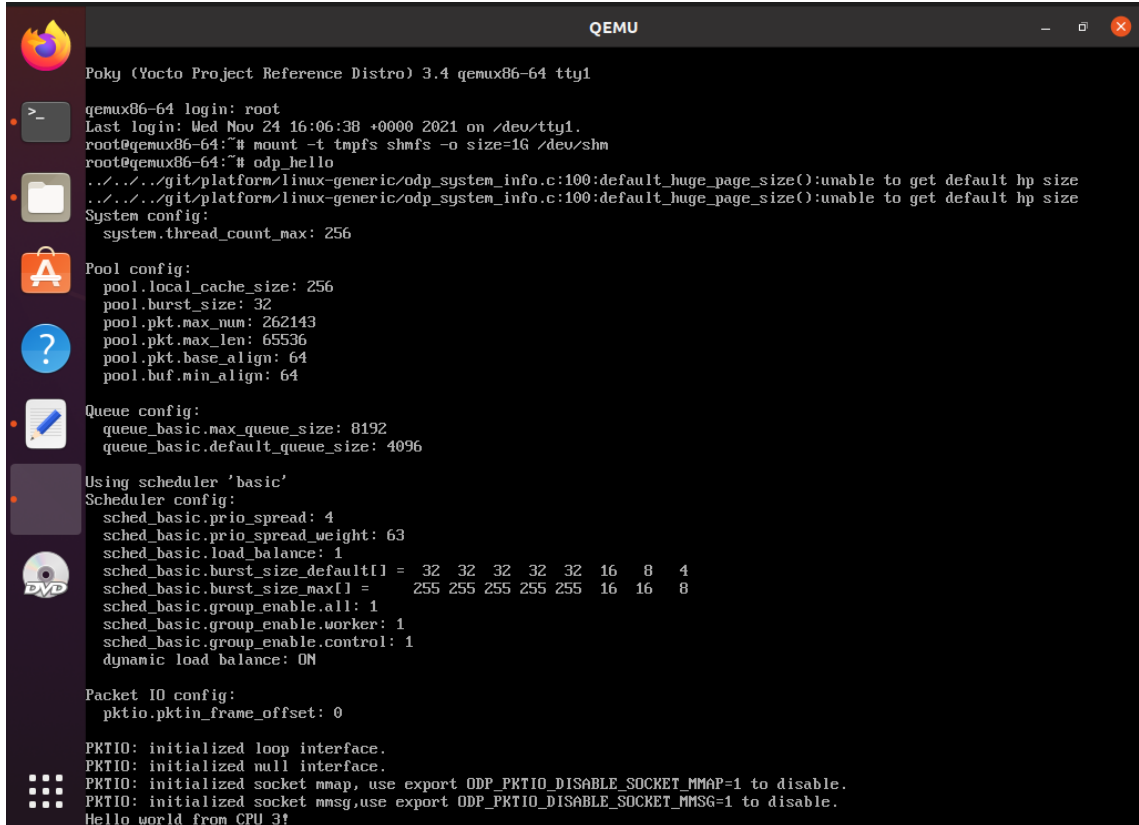
As the CPU polling failed on both of the 32-bit ARM devices, the BeagleBone Black and the Raspberry Pi, at least one fully functioning implementation is needed to show that the project implementation is actually working. This is achieved by using a QEMU virtual machine in the Yocto build environment virtual machine. This kind of virtualization in a virtual machine is known as nested virtualization. In this section, a nested virtual machine with an emulated 64-bit processor is deployed, and the built operating system image is tested with that.

The Yocto image needs to be again separately built for this machine configuration. This can be done by simply changing the value of *MACHINE* variable in the *local.conf* to *qemux86-64* before running the *bitbake* command. After the build is done, some changes are needed for the nested virtual machine configuration. In *build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.qemuboot.conf* the *qb_mem* variable is changed to *-m 2048* value, being *qb_mem = -m 2048* in the configuration file. This is done to set the nested virtual machine memory allocation to two gigabytes.

The nested virtual machine is started with *runqemu qemux86-64* command. This is a command introduced by the Yocto build environment. The command in question runs a virtual machine by the configuration created by the Yocto build, along with automatically including the *qemux86-64* Yocto image built.

After starting the nested *qemux86-64* virtual machine, a separate serial console window opens for the nested virtual machine. Logging in with a user *root*, the */dev/shm* needs to be mounted with *mount -t tmpfs shmfs -o size=1G /dev/shm* command.

Unlike with the 32-bit ARM architecture target devices, the ODP is able to poll the CPU properly in the *qemux86-64* target environment. Full output of the *odp_hello* command issued in the *qemux86-64* target is depicted in the figure 11.



```

QEMU
Poky (Yocto Project Reference Distro) 3.4 qemux86-64 tty1
qemux86-64 login: root
Last login: Wed Nov 24 16:06:38 +0000 2021 on /dev/tty1.
root@qemux86-64:~# mount -t tmpfs shmfs -o size=1G /dev/shm
root@qemux86-64:~# odp_hello
../../../../git/platform/linux-generic/odp_system_info.c:100:default_huge_page_size():unable to get default hp size
../../../../git/platform/linux-generic/odp_system_info.c:100:default_huge_page_size():unable to get default hp size
System config:
  system.thread_count_max: 256

Pool config:
  pool.local_cache_size: 256
  pool.burst_size: 32
  pool.pkt_max_num: 262143
  pool.pkt_max_len: 65536
  pool.pkt_base_align: 64
  pool.buf_min_align: 64

Queue config:
  queue_basic.max_queue_size: 8192
  queue_basic.default_queue_size: 4096

Using scheduler 'basic'
Scheduler config:
  sched_basic.prio_spread: 4
  sched_basic.prio_spread_weight: 63
  sched_basic.load_balance: 1
  sched_basic.burst_size_default[] = 32 32 32 32 32 16 8 4
  sched_basic.burst_size_max[] = 255 255 255 255 255 16 16 8
  sched_basic.group_enable.all: 1
  sched_basic.group_enable.worker: 1
  sched_basic.group_enable.control: 1
  dynamic load balance: ON

Packet IO config:
  pktio.pktin_frame_offset: 0

PKTIO: initialized loop interface.
PKTIO: initialized null interface.
PKTIO: initialized socket mmap, use export ODP_PKTIO_DISABLE_SOCKET_MMAP=1 to disable.
PKTIO: initialized socket msg,use export ODP_PKTIO_DISABLE_SOCKET_MSG=1 to disable.
Hello world from CPU 3!

```

Figure 11: The output of *odp_hello* command in a nested *qemux86-64* virtual machine running the project Yocto image.

The resulting output of the *odp_hello* can be seen in the figure 11. First thing to note is that the ODP is unable to get the default size of the hugepages. This is because the hugepages are not used nor even configured for the Yocto image. The hugepages are not required for basic ODP functionality. After these warnings about the hugepages, the expected output of *odp_hello* is printed, being the features utilizable by the ODP. In the end of the output in figure 11, it can be seen that the *odp_hello* prints a hello message from CPU of index 3. This hello message being from other CPU than index 0, along with now absent warning messages about failing to poll for the number of CPUs and their frequencies, indicate that the CPU can be polled properly in this *qemux86-64* nested virtual machine. With this, it is proven that the ODP in the project Yocto image is properly implemented and functional.

5 Conclusion

The goal of the project was to establish a build environment, in which it would be possible to provision operating system images, for example, for embedded systems. The images in question would be required to include ODP framework compiled for the target architecture.

While the project build environment implementation did work, the ODP did not work on all the end-systems tested. That, however, was due to the ODP being incompatible with the tested 32-bit ARM architectures.

The operating system images were successfully built with ODP for all tested platforms. With this, the goal of the project was met. There was no need to change the Yocto recipes for building the ODP, as all the additional changes were basically to meet the target platform requirements. Once the basic boot file delivery flow was set up, the boot file delivery environment was also easy to use for multiple test builds for each target system. With this, it is concluded that the environment for building the operating system images with ODP included is functional.

This project utilized as a reference for software development chain would be beneficial in, for example, testing a software implementation on an actual hardware or in an emulated environment. As the project results show, the exactly same software can fail when compiled on different architectures. If this sort of environment would be implemented as a part of the software testing, the functional failure witnessed caused by different platform architectures could be caught early on.

If the reference implementation is to be taken into use as a part of a software development chain, the changes in build environment target variables could be automated, along with adding the additional metadata layers needed. An ideal implementation would require little to no manual interaction from the developer. One could, for example, create a script to change the target infrastructure

variables and getting the metadata layer repositories needed, along with setting the proper *bitbake-layers* to be used with the build target. Copying the boot files after the build to the respective file serving locations could also be done by an automated script after a build is successful.

References

- 1 Linaro launches OpenDataPlane™ (ODP) project to deliver open-source, cross-platform interoperability for networking platforms. 2013. Online. <<https://www.linaro.org/news/linaro-launches-opendataplane-odp-project-deliver-open-source-cross-platform-interoperability-networking-platforms>>. Visited on 09/01/2021.
- 2 Technical Overview. 2021. Online. <<https://opendataplane.org/index.php/services/technical-overview/>>. Visited on 09/01/2021.
- 3 OpenDataPlane (ODP) Users-Guide. 2021. Online. <<https://opendataplane.github.io/odp/users-guide/>>. Visited on 11/17/2021.
- 4 Monkman, Bob. 2015. The Emergence of the OpenDataPlane™ Standard. Online. <<https://community.arm.com/arm-community-blogs/b/internet-of-things-blog/posts/the-emergence-of-the-opendataplane-standard>>. Visited on 11/20/2021.
- 5 LICENSE 2018. Online. <<https://github.com/OpenDataPlane/odp/blob/master/LICENSE>>. Visited on 09/01/2021.
- 6 The Open Source Definition. 2007. Online. <<https://opensource.org/docs/osd>>. Visited on 09/01/2021.
- 7 The Linux Foundation Announces Yocto Project Steering Group and Release 1.0. 2011. Online. <<https://www.linuxfoundation.org/press-release/the-linux-foundation-announces-yocto-project-steering-group-and-release-1-0/>>. Visited on 11/05/2021.
- 8 SOFTWARE : REFERENCE DISTRIBUTION. 2021. Online. <<https://www.yoctoproject.org/software-overview/reference-distribution/>>. Visited on 11/05/2021.
- 9 Purdie, Richard; Larson, Chris & Blundell, Phil. 2021. BitBake User Manual. Online. <<https://docs.yoctoproject.org/bitbake/index.html>>. Visited on 11/19/2021.
- 10 Yocto Project Reference Manual. 2021. Online. <<https://docs.yoctoproject.org/current/ref-manual/index.html>>. Visited on 11/25/2021.
- 11 Kerrisk, Michael. 2016. Linux Programmer's Manual. shm_overview manpage. Version 7.

<https://man7.org/linux/man-pages/man7/shm_overview.7.html>. Visited on 05/05/2022.