

**Disclaimer:** This material is a study notes, so for any technical issues please contact me to resolve it periodically.

# The Yocto Project - Up and Running

## Contents

<b>1</b>	<b>Introducing Yocto</b>	<b>1</b>
1.1	What is the Yocto Project ? . . . . .	1
1.2	What is Poky ? . . . . .	2
1.3	Poky workflow . . . . .	3
1.4	Building a minimal image using Poky . . . . .	4
1.5	Poky folder structure . . . . .	8
1.6	Yocto Project Layers . . . . .	11
1.7	Basic meta files types . . . . .	11
1.8	Basic configuration files . . . . .	12
<b>2</b>	<b>Writing Layers, Recipes and using the application SDKs</b>	<b>12</b>
2.1	Simple Custom Layer . . . . .	12
2.2	Writing recipes . . . . .	17
2.2.1	Writing recipes manually . . . . .	17
2.2.2	Using Devtool to automatically create recipes . . . . .	19
2.3	Dependencies . . . . .	24
2.4	Package Splitting . . . . .	24
2.5	Preparing the SDKs for your application development . . . . .	24
2.5.1	Types of SDKs . . . . .	30
<b>3</b>	<b>Packaging</b>	<b>31</b>
<b>4</b>	<b>building an image for Raspberry Pi Board</b>	<b>31</b>
<b>5</b>	<b>Appendix</b>	<b>37</b>
5.1	Debugging the build messages . . . . .	37

## 1 Introducing Yocto

### 1.1 What is the Yocto Project ?

"IT'S NOT AN EMBEDDED LINUX DISTRIBUTION, IT CREATES A CUSTOM ONE FOR YOU. " That's it!.

A Yocto release will have:

- Poky, the build system
- A build appliance; that is, a VMware image of a host system ready to use Yocto
- An Application Development Toolkit (ADT) installer for your host system
- And for the different supported platforms:
  - Prebuilt toolchains
  - Prebuilt packaged binaries
  - Prebuilt images

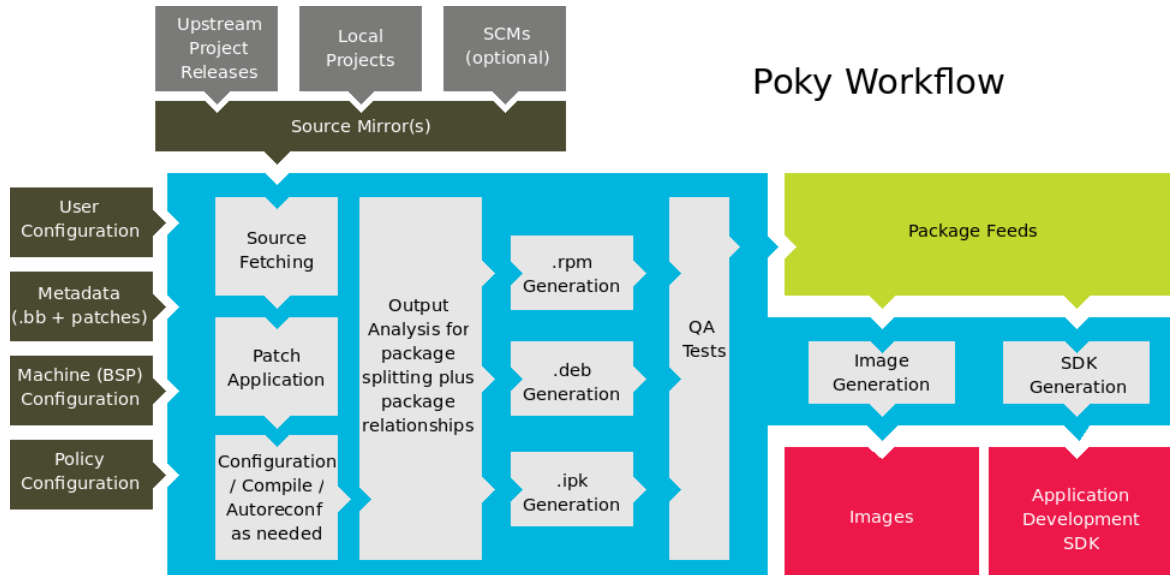
## 1.2 What is Poky ?

Poky is the Yocto project reference **build system**, it's originally a branch from OpenEmbedded build system called *bitbake*, and it contains already the metadata of the OE system.

The purpose of Poky is to build the components needed for an embedded Linux product, namely:

- A bootloader image
- A Linux kernel image
- A root filesystem image
- Toolchains and software development kits (SDKs) for application development.

### 1.3 Poky workflow



1. **do\_fetch** - Downloads the source files and additional artifacts needed to construct a specific SW component. The location can be a remote repository or even a local directory. This location is configured using the SRC\_URI variable of a recipe file.
2. **do\_unpack** - If the downloaded sources in the previous step are compressed, they are decompressed in this step.
3. **do\_patch** - If the source needs to be modified (e.g. due to a known bug), it can be patched in this step using the patch file indicated in the corresponding recipe.
4. **do\_configure** - In this phase, the build configuration is defined (e.g. build files are generated, directives are set, ) as a preparation for the coming task.
5. **do\_compile** - This is the task where the sources are built and linked to produce deployable binaries.
6. **do\_install** - In this task, the produced binaries and any additional files (e.g. images, database files, textual configuration files, ) are grouped together in a folder structure specified by the user (e.g. binaries go to `ipackage_i/bin`, config. files go to `ipackage_i/etc`).
7. **do\_package** - The packaging operation converts the previously generated folder into a known package format (e.g. .rpm, .deb, ...) to be used by package management applications.

And once all needed components are packaged, the following tasks are executed for the image:

8. **do\_rootfs** - Builds a root filesystem that includes the bootloader, the kernel binary, the device tree blob in addition to all packages that need to be deployed in the system.
9. **do\_image** - Finally the generated filesystem is converted into a predefined image format (e.g. `.sdcard` to be deployed onto a SD Card, hard drive image, tarball files, etc)

## 1.4 Building a minimal image using Poky

So, let's have a quick example on building a full Linux distro using Poky, it's as simple as the following four steps:

1. Get Poky Just fetch your own branch of Poky, I am using here *morty* branch with Ubuntu 18.04. You need to take care as some distributions of Yocto has already compatibility issues with a certain Linux images.

Listing 1: Install Poky

```
ahmed@Sabry:~/YoctoProject$ git clone git://git.yoctoproject.org/poky.git

ahmed@Sabry:~/YoctoProject/poky$ ls
bitbake LICENSE.MIT meta-skeleton README.poky
contrib MEMORIAM meta-yocto-bsp README.qemu
documentation meta oe-init-build-env scripts
LICENSE meta-poky README.hardware
LICENSE.GPL-2.0-only meta-selftest README.OE-Core

ahmed@Sabry:~/YoctoProject$ git fetch --tags
ahmed@Sabry:~/YoctoProject$ git tag
1.1.M1.final
1.1.M1.rc1
1.1.M1.rc2
1.1.M2.final
1.1.M2.rc1
.
.
.
yocto-2.5
yocto-2.5.1
yocto-2.5.2
yocto-2.6
yocto-2.6.1
yocto-2.6.2
yocto-2.7
yocto_1.5.M5.rc8
```

```
# Checkout a tag
ahmed@Sabry:~/YoctoProject$ git checkout tags/yocto-3.0
```

2. Configure your system After getting poky installed just run the **oe-init-build-env**, and see what happened ?

Listing 2: Config the environment

```
ahmed@Sabry:~/YoctoProject/poky$ source oe-init-build-env
```

```
ahmed@Sabry:~/YoctoProject/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:
    http://yoctoproject.org/documentation

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/

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

You can now run 'bitbake <target>'

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

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

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
ahmed@Sabry:~/YoctoProject/poky/build$ ls
conf
ahmed@Sabry:~/YoctoProject/poky/build$
```

it will setup the environment and create a build directory (with conf folder; contains the a build specific conf files, we will take about them later) for us in order to start, and finally here we are in that build directory to *bitbake* our image.

3. Build the distro using *bitbake*

Listing 3: bitbake a minimal image

```
$ bitbake core-image-minimal
```

Here we are trying to build a minimal Linux distro (something which can boot and run), and as you can see it fetches the stuff and building.

after some `do_compile` and `do_package` and so on, then you have your own linux image. you can use see the output here **"build/tmp/ deploy/images/qemux86/"**

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ cd tmp/deploy/images/qemux86/
ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/images/qemux86$ ls
bzImage
bzImage--4.8.26+git0+1c60e003c7_27efc3ba68-r0-qemux86-20200314095138.bin
bzImage-qemux86.bin
core-image-minimal-qemux86-20200314095138.qemuboot.conf
core-image-minimal-qemux86-20200314095138.rootfs.ext4
core-image-minimal-qemux86-20200314095138.rootfs.manifest
core-image-minimal-qemux86-20200314095138.rootfs.tar.bz2
core-image-minimal-qemux86.ext4
core-image-minimal-qemux86.manifest
core-image-minimal-qemux86.qemuboot.conf
core-image-minimal-qemux86.tar.bz2
modules--4.8.26+git0+1c60e003c7_27efc3ba68-r0-qemux86-20200314095138.tgz
modules-qemux86.tgz
ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/images/qemux86$
```

as you might noticed that it taken little bit of time for the first time to build.



### Note: where to find all the supported images ?

Note: you can check all the supported images by the following command:

Listing 4: Supported Images

```

ahmed@Sabry:~/TheYoctoProject/poky$ ls meta*/recipes*/images/*.bb
meta-mylayer/recipes-example/images/example-image.bb
meta/recipes-core/images/build-appliance-image_15.0.0.bb
meta/recipes-core/images/core-image-base.bb
meta/recipes-core/images/core-image-minimal.bb
meta/recipes-core/images/core-image-minimal-dev.bb
meta/recipes-core/images/core-image-minimal-initramfs.bb
meta/recipes-core/images/core-image-minimal-mtdutils.bb
meta/recipes-extended/images/core-image-full-cmdline.bb
meta/recipes-extended/images/core-image-kernel-dev.bb
meta/recipes-extended/images/core-image-lsb.bb
meta/recipes-extended/images/core-image-lsb-dev.bb
meta/recipes-extended/images/core-image-lsb-sdk.bb
meta/recipes-extended/images/core-image-testmaster.bb
meta/recipes-extended/images/core-image-testmaster-initramfs.bb
meta/recipes-graphics/images/core-image-clutter.bb
meta/recipes-graphics/images/core-image-weston.bb
meta/recipes-graphics/images/core-image-x11.bb
meta/recipes-rt/images/core-image-rt.bb
meta/recipes-rt/images/core-image-rt-sdk.bb
meta/recipes-sato/images/core-image-sato.bb
meta/recipes-sato/images/core-image-sato-dev.bb
meta/recipes-sato/images/core-image-sato-sdk.bb
meta/recipes-sato/images/core-image-sato-sdk-ptest.bb
meta-selftest/recipes-test/images/error-image.bb
meta-selftest/recipes-test/images/oe-selftest-image.bb
meta-selftest/recipes-test/images/test-empty-image.bb
meta-selftest/recipes-test/images/wic-image-minimal.bb
meta-skeleton/recipes-multilib/images/core-image-multilib-example.bb

```

4. Finally, run your first yocto image by using the *qemu* emulator.

Listing 5: runqemu command to load your image

```

ahmed@Sabry:~/TheYoctoProject/poky/build$ runqemu qemux86 nographic

#After some boot messages
#Here we are!
Poky (Yocto Project Reference Distro) 2.2.4 qemux86 /dev/ttyS0

qemux86 login: root
root@qemux86:~#

```

```
root@qemux86:~# uname -a
Linux qemux86 4.8.26-yocto-standard #1 SMP PREEMPT Sat Mar 14 14:07:45 EET 2020 i686 GNU/Linux
root@qemux86:~#
```

```
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 2.2.4 qemux86 /dev/ttyS0

qemux86 login: root
INIT: Id "S1" respawning too fast: disabled for 5 minutes

root@qemux86:~# uname -a
Linux qemux86 4.8.26-yocto-standard #1 SMP PREEMPT Sat Mar 14 14:07:45 EET 2020
i686 GNU/Linux
```

## 1.5 Poky folder structure

Now after a successful build of our image, let's have a look on *Poky* folder structure.

Listing 6: Yocto folder structure

```
poky
|--- bitbake
| |--- bin
| |--- contrib
| |--- doc
| |--- lib
|--- build
| |--- cache
| |--- conf
| |--- downloads
| |--- sstate-cache
| |--- tmp
| | |--- buildstats
| | |--- cache
| | |--- deploy
| | |--- log
| | |--- sstate-control
| | |--- stamps
| | |--- sysroots
| | |--- sysroots-components
| | |--- sysroots-uninative
| |--- work
|--- workspace
|--- appends
```



```

| |---- conf
| |---- recipes
| ---- sources
|---- documentation
|---- meta
| |---- classes
| |---- conf
| |---- files
| |---- lib
| |---- recipes-XX
| ---- site
|---- meta-poky
| |---- classes
| |---- conf
| ---- recipes-core
|---- meta-raspberrypi
| |---- classes
| |---- conf
| |---- files
| |---- recipes-XX
| ---- scripts
|---- meta-selftest
| |---- classes
| |---- conf
| |---- files
| |---- lib
| ---- recipes-test
|---- meta-skeleton
| |---- conf
| |---- recipes-XX
|---- meta-yocto
| ---- conf
|---- meta-yocto-bsp
| |---- conf
| |---- lib
| |---- recipes-XX
|---- scripts
| |---- contrib
| | |---- bb-perf
| |---- python
| |---- lib
| | |---- bsp
| | |---- devtool
| | |---- recipetool
| |---- wic
| |---- native-intercept
| |---- postinst-intercepts
| |---- pybootchartgui
| |---- pybootchartgui
|---- tiny

```

---

As you can see from the tree-like structure, *Yocto* has the following main folders:

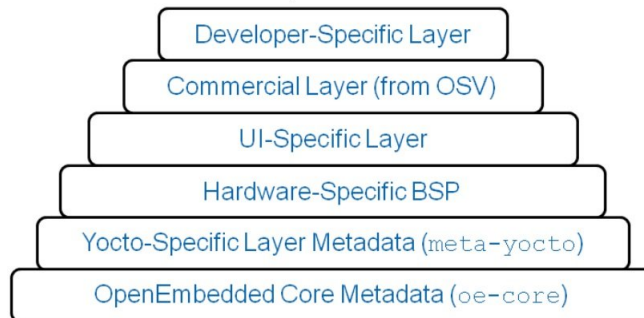
- **bitbake** folder  
It contains all the stuff related to the OpenEmbedded (OE) bitbake build system, which is a reference one for Poky.
- **meta-XX** folders (meta, meta-poky, meta-yocto, ...)  
Any folder annotated by *meta-*, it will be a layer to contain a certain yocto meta and configuration data. We have here a basic meta layers, and please refer to the layers section [Yocto Project Layers](#) for more details.

You might noticed that for every layer, there is a at least **conf** folder which contains a specific layer configuration file.

- **Build** folder:  
All the magic while building occurs on this folder, and all your output images and SDKs goes here, and as you might notice that this folder contains:
  - **downloads** folder; which is used while fetching the files before building
  - **conf** folder; contains configuration files for the current build, here mainly you will find:
    - \* **bblayers.conf** file: which contains the configured meta-layers used in the current build
    - \* **local.conf** file: contains some build variables/parameters to customize the buildand both files will be described later.
  - **tmp** folder: while building, and all the outputs (SDK, images, ...) will be placed here in this folder.
    - \* **images** folder: holds all the output images
    - \* **SDKs** folder: holds all the populated SDKs
- **scripts** It contains all the python scripts to automate the fetching, build, and packaging.
- **documentation**

Note: You can get all the tree-like structure by using Linux *tree* command, in case you want to explore more.

## 1.6 Yocto Project Layers



In any Poky version, you will find the following basic layers:

- **meta**: This directory contains the OpenEmbedded-Core metadata which have the base layers of all the build system, for example the basic tasks like **do\_fetch**, **do\_compile**, ... are defined in **base.bbclass** in the OE meta folders.
- **meta-yocto**: This contains Poky's distribution-specific metadata
- **meta-yocto-bsp**: This contains metadata for the reference hardware boards

The initial poky build system only supports the following BSP targets, and the virtualized emulators provided by OE-Core, and of course other layers can be included, or defined and customized, you can refer to creating [Simple Custom Layer](#) section.

Listing 7: Supported virtualized emulators

```
ahmed@Sabry:~/TheYoctoProject/poky$ ls meta/conf/machine/*.conf
meta/conf/machine/qemuarm64.conf meta/conf/machine/qemuppc.conf
meta/conf/machine/qemuarm.conf meta/conf/machine/qemux86-64.conf
meta/conf/machine/qemumips64.conf meta/conf/machine/qemux86.conf
meta/conf/machine/qemumips.conf
```

## 1.7 Basic meta files types

Yocto has several files types to support extending your layer:

- **.bbclass files**: Class files contain information that is useful to share between recipes files. An example is the autotools class, which contains common settings for any application that Autotools uses.
- **.bb recipe files**: the files that have the .bb suffix are "recipes" files. In general, a recipe contains information about a single piece of software, and each recipe describes a set of meta info and tasks to do deal with this software unit.

- **.bbappend files:** You can use a .bbappend file in your layer to make additions or changes to the content of another layer's recipe without having to copy the other layer's recipe into your layer. Your .bbappend file resides in your layer, while the main .bb recipe file to which you are appending Metadata resides in a different layer.
- **.conf files:** The configuration files define various configuration variables that govern the OpenEmbedded build process.

## 1.8 Basic configuration files

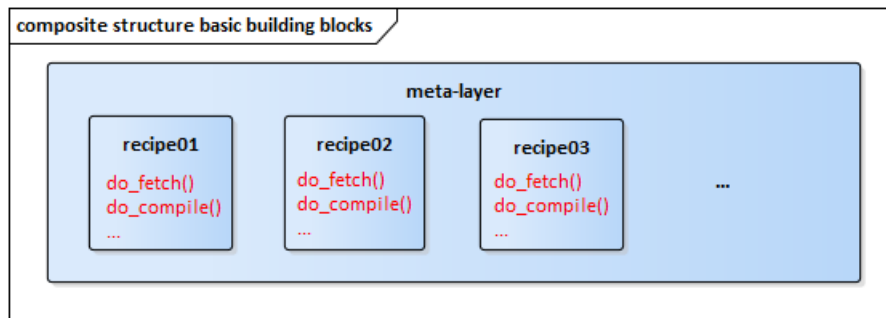
Finally as an introduction to *Yocto*, we can list some important configuration files which we will deal with a lot in our further sections.

- Layers Configuration files:  
**build/conf/bblayers.conf** : This file is used to find all the configured layers.  
**meta-xx/conf/layer.conf**: layer specific configuration file.
- Build Specific configuration:  
**build/conf/local.conf**: This file is used for any other configuration the user may have for the current build
- Core configuration files:  
**meta/conf/distro/<distro>.conf**: This file is the distribution policy; by default, this is the poky.conf file.  
**meta/conf/bitbake.conf**: this file used to set some default values for overall system, export some core variables, and assign some core configurations.  
**meta/conf/machine/<machine>.conf**: This file is the machine configuration; in our case.

## 2 Writing Layers, Recipes and using the application SDKs

### 2.1 Simple Custom Layer

A Yocto layer basically contains set of recipes to describe the workflow for your application building.



So in order to write some recipes, it's needed first to create our custom layer, we will create it in two different ways, and we will call it **my-layer**.

A. You can create a layer automatically by using **create-layer** sub-command, and you need respect the yocto project naming by creating your layer name as follow **meta-*<your-layer-name>***, as it helps bitbake while parsing your recipes.

Listing 8: bitbake-layers create-layer

```
ahmed@Sabry:~/TheYoctoProject/poky$ bitbake-layers create-layer meta-my-layer
```

By running this command, it will create your layer and append it to the different configuration files in your build system (i.e. append it to bblayers.conf file, and creates layer.conf file).

B. Or you can create a layer manually as follow:

1. Create a folder for your layer, and you need to respect yocto naming, create it as *meta-my-layer*

Listing 9: Create the layer folder

```
ahmed@Sabry:~/TheYoctoProject/poky$ mkdir meta-my-layer
ahmed@Sabry:~/TheYoctoProject/poky$ mkdir meta-my-layer/conf
ahmed@Sabry:~/TheYoctoProject/poky$ cd meta-my-layer/conf/
ahmed@Sabry:~/TheYoctoProject/poky/conf$ vim layer.conf
ahmed@Sabry:~/TheYoctoProject/poky/meta-my-layer/recipes-example/images$ vim example-image.bb
```

2. Create a layer config file, in **"/poky/meta-my-layer/conf/layer.conf"**

Listing 10: add layer config file

```
# conf and classes directory added to BBPATH
BBPATH += "${LAYERDIR}"

# recipes-* directories added to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipe-*/*/*.bbappend"
```

```
# this variable is used to set the layer name
BBFILE_COLLECTIONS += "mylayer"
#used to match files in BBFILE, here it's used to match within the layer path
BBFILE_PATTERN_mylayer = "^${LAYERDIR}/"

# Assign recipes, this is helpful when the same recipe in multiple layers and allows you to choose the layer that takes
BBFILE_PRIORITY_mylayer = "6"
```

3. Enabling your layer by adding it to your image, you can do that either by using *bitbake-layers add-layer* command or by adding it manually to **bblayers.conf** file

Listing 11: include your layer

```
ahmed@Sabry:~/TheYoctoProject/poky$ bitbake-layers add-layer meta-mylayer

# Or you can make it manually by appending your layer to bblayers.conf file
ahmed@Sabry:~/TheYoctoProject/poky$ vim build/conf/bblayers.conf

# then re-parse again
ahmed@Sabry:~/TheYoctoProject/poky$ bitbake example
```

```
ahmed@Sabry: ~/TheYoctoProject/poky
File Edit View Search Terminal Help
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

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

BBLAYERS ?= " \
    /home/ahmed/TheYoctoProject/poky/meta \
    /home/ahmed/TheYoctoProject/poky/meta-poky \
    /home/ahmed/TheYoctoProject/poky/meta-yocto-bsp \
    /home/ahmed/TheYoctoProject/poky/meta-mylayer \
"
```



### Note: Yocto meta-layer index site ?

- Before creating a specific layer, you need to look at the [Yocto metadata Index site](#) for already created layers, it might fit your needs.
- if you would like your layer to be global at the index, fill the application of yocto project compatible [here](#)

Finally let's create our own image; you need to create **recipes/images** as follow, then we will use a pre-existing defined image from **meta** layer (**core-image-minimal.bb**), then customize it to include the "mc" application.

Listing 12: create an image recipe

```
ahmed@Sabry:~/TheYoctoProject/poky/meta-mylayer$ cd recipes-example/
ahmed@Sabry:~/TheYoctoProject/poky/meta-mylayer/recipes-example$ mkdir images
ahmed@Sabry:~/TheYoctoProject/poky/meta-mylayer/recipes-example$ ls ../../meta/recipes-core/images/
build-appliance-image core-image-minimal-dev.bb
build-appliance-image_15.0.0.bb core-image-minimal-initramfs.bb
core-image-base.bb core-image-minimal-mtdutils.bb
core-image-minimal.bb

ahmed@Sabry:~/TheYoctoProject/poky/meta-mylayer/recipes-example$ cp ../../meta/recipes-core/images/core-image-
```

```
ahmed@Sabry: ~/TheYoctoProject/poky/meta-mylayer/recipes-example/images
File Edit View Search Terminal Help
require core-image-minimal.bb

DESCRIPTION = "A small image containing the mc application"

IMAGE_INSTALL += "mc"
```

Listing 13: building our custom image

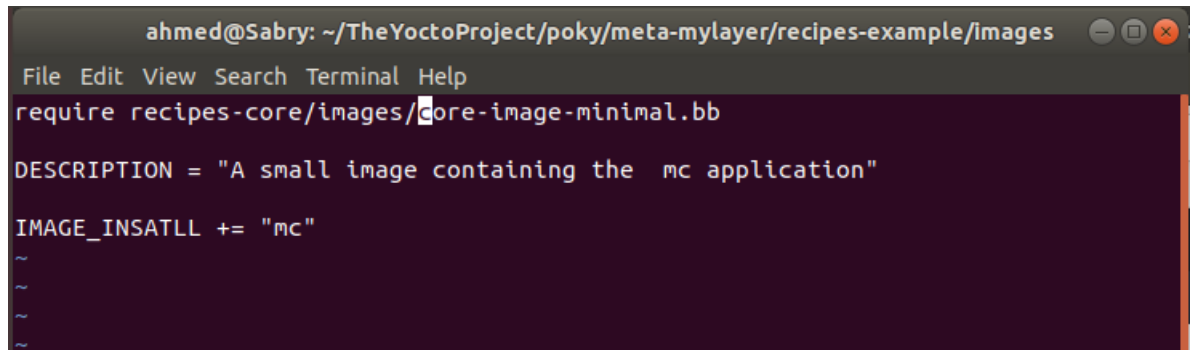
```
# Then finally
ahmed@Sabry:~/TheYoctoProject/poky/meta-mylayer/recipes-example/images$ bitbake example-image

# But there is an error!!!
WARNING: Host distribution "Ubuntu-18.04" has not been validated with this version of the build system; you may possibly
Loading cache: 100% |#####| Time: 0:00:00
Loaded 1321 entries from dependency cache.
ERROR: ParseError at /home/ahmed/TheYoctoProject/poky/meta-mylayer/recipes-example/images/example-image.bb:

Summary: There was 1 WARNING message shown.
```

Summary: There was 1 ERROR message shown, returning a non-zero **exit** code

in order to overcome this error, we need to refer to the absolute path of the **core-image-minimal.bb**.



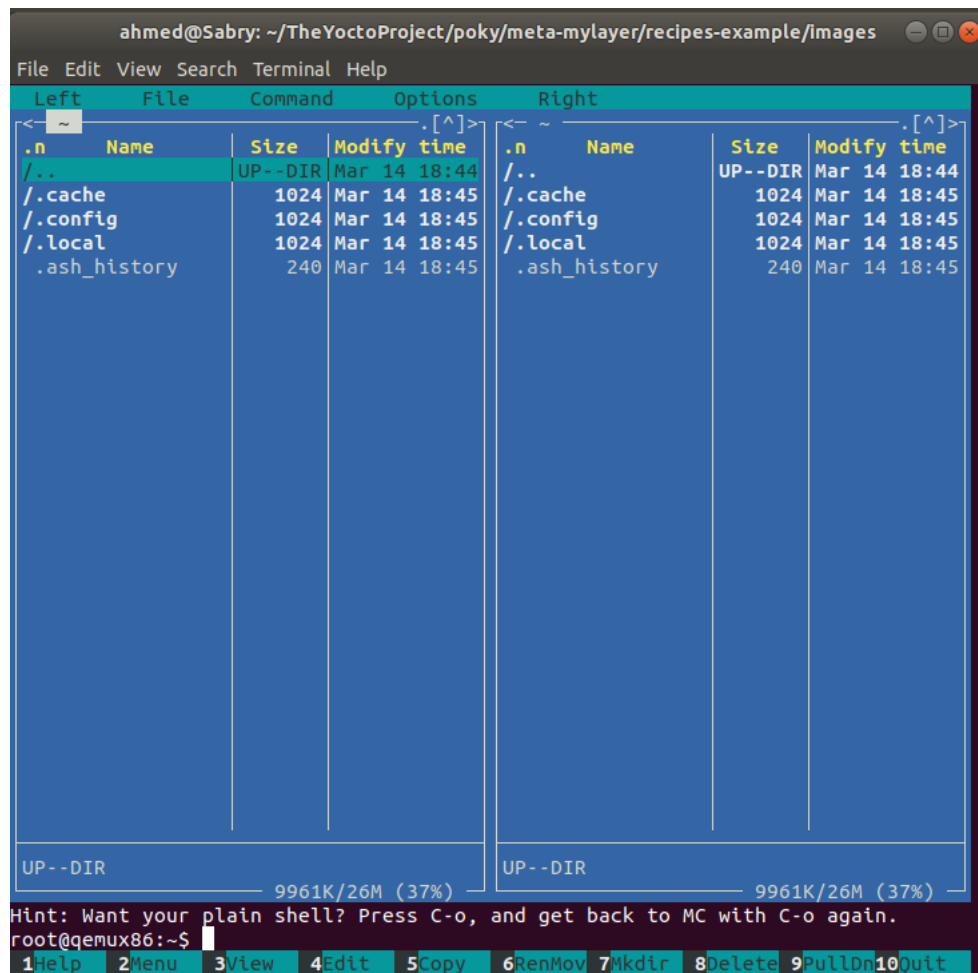
```
ahmed@Sabry: ~/TheYoctoProject/poky/meta-mylayer/recipes-example/images
File Edit View Search Terminal Help
require recipes-core/images/core-image-minimal.bb

DESCRIPTION = "A small image containing the mc application"

IMAGE_INSTALL += "mc"
~
~
~
~
```

then here we go, a minimal image with "mc" command based on our new custom layer.





## 2.2 Writing recipes

### 2.2.1 Writing recipes manually

Now it's time to create a new recipe to build our "Hello Yocto!" program and add it to the bin folder of our custom image.

in the folder **meta-my-layer/recipes-example** create an **example** folder and inside it you need to create **hello-0.1.bb** recipe and **hello.c** file.

Listing 14: hello.c file

```
#include <stdio.h>

int main() {
    printf(" Hello, Yocto!\n");
    return 0;
}
```

```
}
```

and the recipe file as follow:

Listing 15: hello-0.1 recipe

```
DESCRIPTION = "Simple hello world application"
LICENSE = "MIT"

SRC_URI = "file://hello.c"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

S = "${WORKDIR}"

do_compile() {
    ${CC} hello.c ${LDFLAGS} -o helloyocto
}

do_install() {
    install -d ${D}${bindir}
    install -m 0755 helloyocto ${D}${bindir}
}
```

Listing 16: building using bitbake

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ bitbake hello-0.1

ahmed@Sabry:~/TheYoctoProject/poky/build$ cd tmp/deploy/rpm/

# You will find the different files of the hello bin
ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/rpm$ find -name hello*
./i586/hello-0.1-dev-1.0-r0.i586.rpm
./i586/hello-0.1-dbg-1.0-r0.i586.rpm
./i586/hello-0.1-1.0-r0.i586.rpm
```

To add the output of this recipe to our custom image, you need to add the your recipe to your conf/local.conf file in your Yocto build directory:

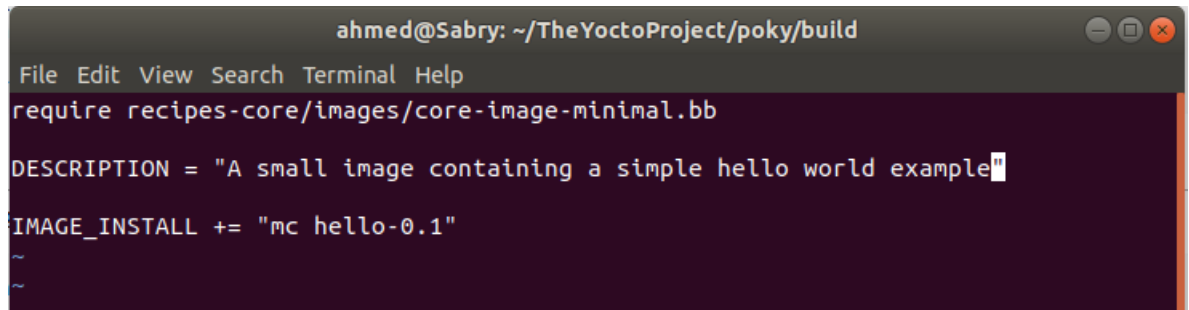
**IMAGE\_INSTALL\_append = " hello-0.1"**

or you can append it to the images file of your layer in **meta-mylayer/recipes-example/images/example-image.bb**



**Note: Recipes programming commands**

A normal bitbake recipes commands are a mix of python and bash commands!

A terminal window titled 'ahmed@Sabry: ~/TheYoctoProject/poky/build'. The window contains the following text: 'require recipes-core/images/core-image-minimal.bb', 'DESCRIPTION = "A small image containing a simple hello world example"', and 'IMAGE\_INSTALL += "mc hello-0.1"'. There are also some tilde characters '~' at the bottom.

```
ahmed@Sabry: ~/TheYoctoProject/poky/build
File Edit View Search Terminal Help
require recipes-core/images/core-image-minimal.bb

DESCRIPTION = "A small image containing a simple hello world example"

IMAGE_INSTALL += "mc hello-0.1"
~
~
```

after booting-up your custom image, the output is here !

A terminal window showing the login process for 'root' on 'qemux86'. The output is: 'qemux86 login: root', 'root@qemux86:~# helloyocto', 'Hello, Yocto!', and 'root@qemux86:~#'.

```
qemux86 login: root
root@qemux86:~# helloyocto
Hello, Yocto!
root@qemux86:~#
```

### 2.2.2 Using Devtool to automatically create recipes

**devtool** is an automated recipe creator, build and deploy tool. simply you can add your new packages or modules to your OS image as a new recipes and setup your environment automatically. and the following are devtool simple commands to deal with the recipes:

- devtool add: Assists in adding new software to be built.
- devtool modify: Sets up an environment to enable you to modify the source of an existing component.
- devtool upgrade: Updates an existing recipe so that you can build it for an updated set of source files.

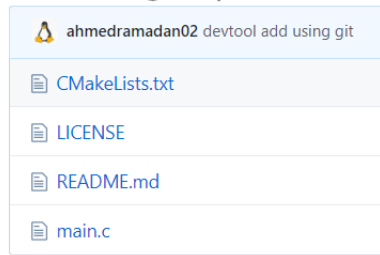
#### Types of projects currently supported by devtool:

1. autotools (autoconf, and autotools)
2. cmake
3. qmake
4. plain makefile
5. Binary Packages
6. out-of-tree kernel module
7. Node.js Module.
8. Python modules that use setuptools or distutils.

Let's have an example to automatically create our recipes using devtool:

Prerequisites: Github repo containing our code and our build environment, here we are using **cmake**, devtool will read the **CMakeFile.txt** and create our recipe accordingly, and you need to notice that devtool can work with different build environments as mentioned in [Using Devtool to automatically create recipes](#).

git repo



CMakeLists.txt

```
1 cmake_minimum_required(VERSION 2.6)
2 project(say_hello)
3 add_executable(say_hello main.c)
4 install(TARGETS say_hello DESTINATION bin)
```

LICENSE file

```
1 MIT License
2
3 Copyright (c) 2019
4
5 Permission is hereby granted, free of charge, to any person obtaining
6 copies of this software and associated documentation files (the "Software"),
7 to use, copy, modify, merge, publish, distribute, sublicense, and/or
8 to make any improvements to the Software without restriction, including
9 without limitation to the rights to use, copy, modify, merge, publish,
10 distribute, sublicense, and/or to make any improvements to the Software
```

1. Use devtool add to create your recipe from the github repo.

Listing 17: Devtool add

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ devtool add sayhello https://github.com/ahmedramadan02/yocto-test
NOTE: Using default source tree path /home/ahmed/TheYoctoProject/poky/build/workspace/sources/sayhello
NOTE: Recipe /home/ahmed/TheYoctoProject/poky/build/workspace/recipes/sayhello/sayhello_git.bb has been automatically
generated.

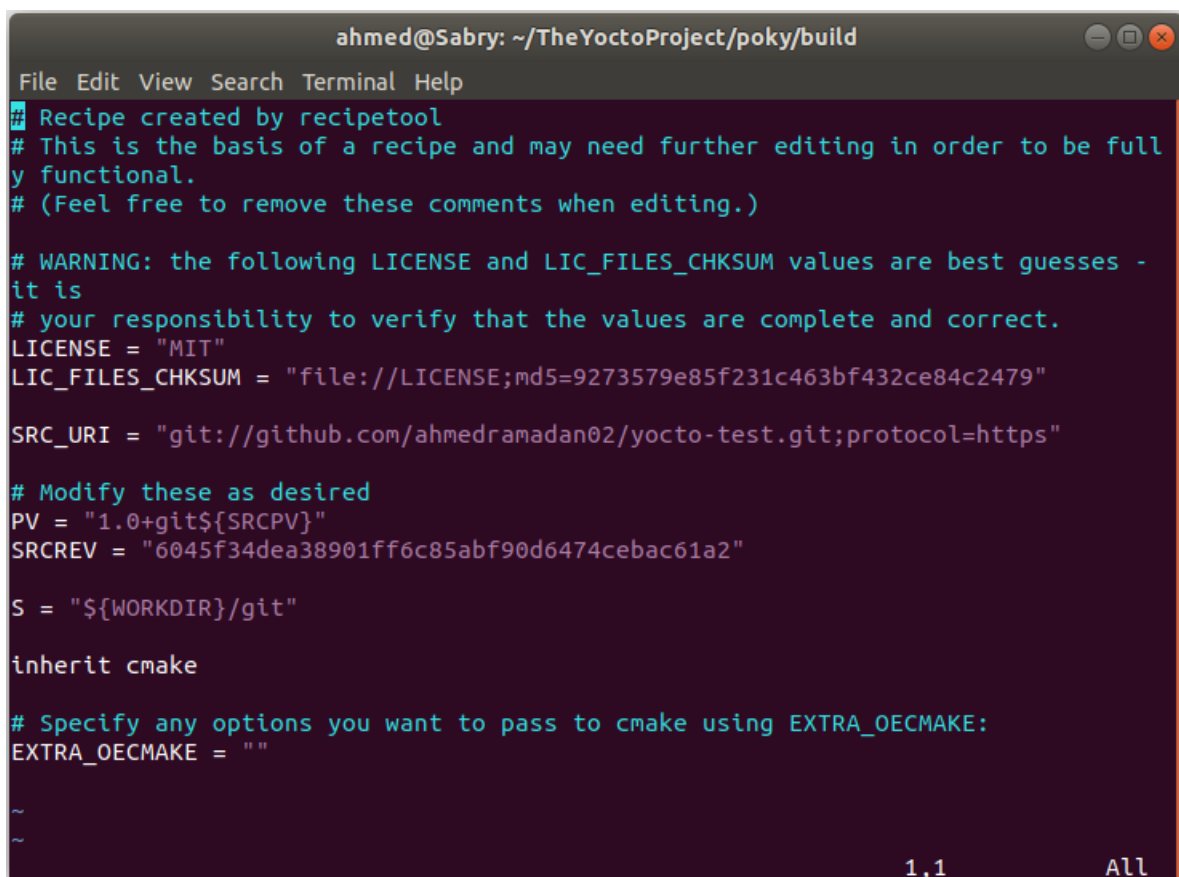
# And you might notice that the following is created
ahmed@Sabry:~/TheYoctoProject/poky/build$ cd workspace/
ahmed@Sabry:~/TheYoctoProject/poky/build/workspace$ ls
conf  README  recipes  sources
ahmed@Sabry:~/TheYoctoProject/poky/build/workspace$ ls recipes/sayhello/
sayhello_git.bb
ahmed@Sabry:~/TheYoctoProject/poky/build/workspace$ ls sources/sayhello/
CMakeLists.txt  LICENSE  main.c  README.md
```

The recipe is created and you might noticed that by using the devtool, all the stuff is going to be in **"/build/workspace/recipes"** folder. also

devtool will add this **workspace** folder to the **bblayers.conf** automatically.

```
BBLAYERS ?= " \
/home/ahmed/TheYoctoProject/poky/meta \
/home/ahmed/TheYoctoProject/poky/meta-poky \
/home/ahmed/TheYoctoProject/poky/meta-yocto-bsp \
/home/ahmed/TheYoctoProject/poky/meta-mylayer \
/home/ahmed/TheYoctoProject/poky/build/workspace \
"
```

and the output recipes is created as follow, by inheriting the **cmake** and adding our MIT license:



```
ahmed@Sabry: ~/TheYoctoProject/poky/build
File Edit View Search Terminal Help
# Recipe created by recipetool
# This is the basis of a recipe and may need further editing in order to be fully functional.
# (Feel free to remove these comments when editing.)

# WARNING: the following LICENSE and LIC_FILES_CHKSUM values are best guesses - it is
# your responsibility to verify that the values are complete and correct.
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://LICENSE;md5=9273579e85f231c463bf432ce84c2479"

SRC_URI = "git://github.com/ahmedramadan02/yocto-test.git;protocol=https"

# Modify these as desired
PV = "1.0+git${SRCPV}"
SRCREV = "6045f34dea38901ff6c85abf90d6474cebac61a2"

S = "${WORKDIR}/git"

inherit cmake

# Specify any options you want to pass to cmake using EXTRA_OECMAKE:
EXTRA_OECMAKE = ""

~
~
1,1 All
```

2. use the devtool edit command to make some edits before bitbaking our recipe.

Listing 18: Devtool edit

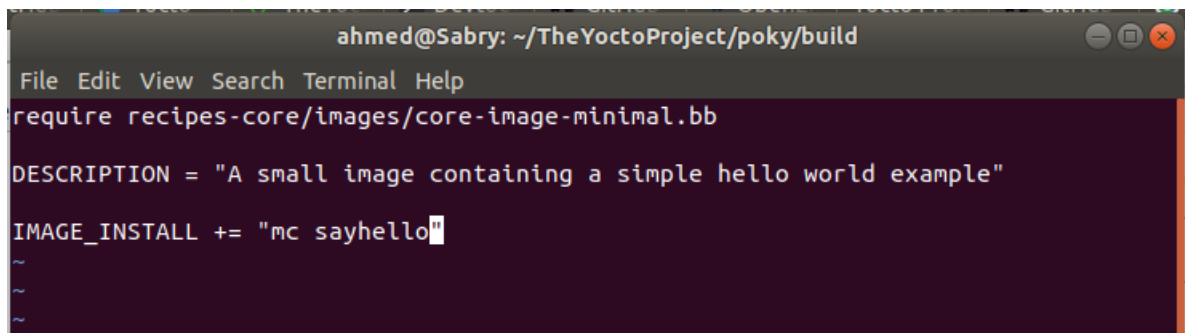
```
$ devtool edit--recipe sayhello
```

3. then we can use devtool to build our recipe

Listing 19: Devtool build a certain recipe

```
$ devtool build sayhello
```

You need add the recipe first to your **local.conf** file, and then build the image.



Listing 20: Devtool build the image

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ devtool build--image example--image
```

4. deploy your recipe to your image (optional if you added the recipe already to **local.conf** file)  
after bitbaking "sayhello", you can directly transfer it to our image by using **bitbake deploy** command. and in order to do so, your image should be running as **devtool deploy** command uses the same way as **scp** command.

Listing 21: Devtool deploy-image

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ devtool deploy--target -s -c sayhello root@192.168.7.2
Parsing recipes..done.
Warning: Permanently added '192.168.7.2' (RSA) to the list of known hosts.
devtool_deploy.list 100% 25 3.7KB/s 00:00
devtool_deploy.sh 100% 1014 144.4KB/s 00:00
Warning: Permanently added '192.168.7.2' (RSA) to the list of known hosts.
./
```

```
./usr/  
./usr/bin/  
./usr/bin/say_hello
```

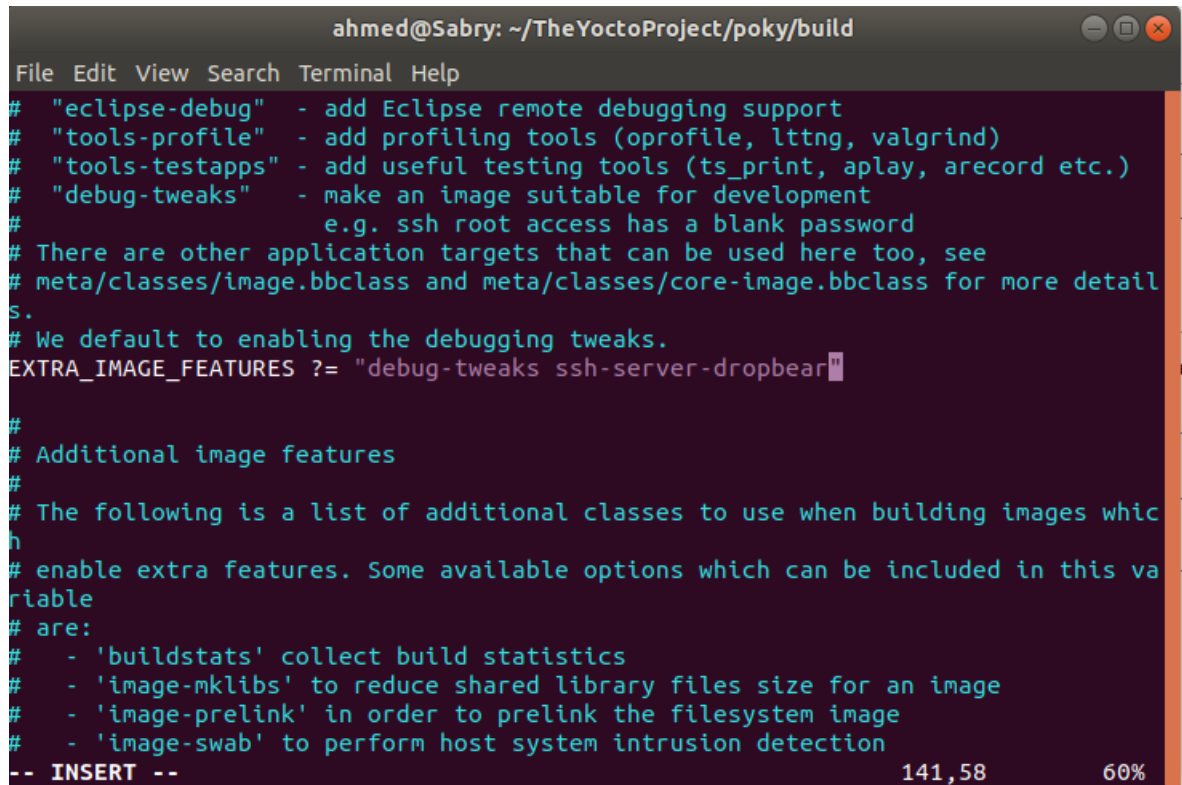
NOTE: Successfully deployed /home/ahmed/TheYoctoProject/poky/build/tmp/work/armv5e-poky-linux-gnueabi/

Here we go, the following is the deployed binary, before and after using the **devtool deploy** command:

The image shows two terminal windows. The left window, titled 'ahmed@Sabry: ~/TheYoctoProject/poky/build', shows the output of the 'ifconfig' command for the 'eth0' interface, displaying IP address 192.168.7.2 and other network details. It also shows the configuration for the 'lo' (loopback) interface. The right window, titled 'ahmed@Sabry: ~/TheYoctoProject/poky/build', shows the execution of the 'devtool deploy-target -s -c' command. It displays progress bars for 'devtool\_deploy.list' and 'devtool\_deploy.sh', both at 100%. It also shows a warning about adding the IP address '192.168.7.2' to the list of known hosts. Finally, it shows the execution of the 'say\_hello' command, which outputs 'Hello, Ahmed'.

```
ahmed@Sabry: ~/TheYoctoProject/poky/build  
File Edit View Search Terminal Help  
root@gemuarml:~# ifconfig  
eth0      Link encap:Ethernet  HWaddr 52:54:00:12:34:02  
          inet addr:192.168.7.2  Bcast:192.168.7.255  Mask:255.255.255.0  
          inet6 addr: fe80::5054:ff:fe12:3402%71/64 Scope:Link  
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1  
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:1000  
          RX bytes:110 (110.0 B)  TX bytes:578 (578.0 B)  
  
lo        Link encap:Local Loopback  
          inet addr:127.0.0.1  Mask:255.0.0.0  
          inet6 addr: ::1%71/128 Scope:Host  
          UP LOOPBACK RUNNING  MTU:65536  Metric:1  
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:1  
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)  
  
root@gemuarml:~# say_hello Ahmed  
-sh: say_hello: not found  
root@gemuarml:~# say_hello Ahmed  
Hello, Ahmed  
root@gemuarml:~#  
  
ahmed@Sabry: ~/TheYoctoProject/poky/build  
File Edit View Search Terminal Help  
ahmed@Sabry:~/TheYoctoProject/poky/build$ devtool deploy-target -s -c  
o root@192.168.7.2  
Parsing recipes..done.  
Warning: Permanently added '192.168.7.2' (RSA) to the list of known hosts  
devtool_deploy.list 100% 25 3.7KB/s 00  
devtool_deploy.sh 100% 1014 144.4KB/s 00  
Warning: Permanently added '192.168.7.2' (RSA) to the list of known hosts  
./usr/  
./usr/bin/  
./usr/bin/say_hello  
NOTE: Successfully deployed /home/ahmed/TheYoctoProject/poky/build/tmp/work/armv5e-poky-linux-gnueabi/sayhello/1.0+git999-r0/image  
ahmed@Sabry:~/TheYoctoProject/poky/build$
```

Note: In order to use the **devtool deploy** feature, you need to edit your **build/conf/local.conf** file to add the ssh server feature (e.g. openssh, or ssh-server-dropbear) to your image and bitbake it again.



```
ahmed@Sabry: ~/TheYoctoProject/poky/build
File Edit View Search Terminal Help
# "eclipse-debug" - add Eclipse remote debugging support
# "tools-profile" - add profiling tools (oprofile, lttng, valgrind)
# "tools-testapps" - add useful testing tools (ts_print, aplay, arecord etc.)
# "debug-tweaks" - make an image suitable for development
#                   e.g. ssh root access has a blank password
# There are other application targets that can be used here too, see
# meta/classes/image.bbclass and meta/classes/core-image.bbclass for more detail
# S.
# We default to enabling the debugging tweaks.
EXTRA_IMAGE_FEATURES ?= "debug-tweaks ssh-server-dropbear"
#
# Additional image features
#
# The following is a list of additional classes to use when building images which
# enable extra features. Some available options which can be included in this variable
# are:
# - 'buildstats' collect build statistics
# - 'image-mklibs' to reduce shared library files size for an image
# - 'image-prelink' in order to prelink the filesystem image
# - 'image-swab' to perform host system intrusion detection
-- INSERT -- 141,58 60%
```

## 2.3 Dependencies

There are couple of run-time and compile time dependencies that you can specify in a recipe. for the compile time, you let bitbake realize that and build it with your binary at the compile time. and the run-time dependencies is used at the run time to make your software able to run.

You can add dependencies using the depend variable as the below example:

## 2.4 Package Splitting

Beside the package we build, we can use bitbake to install other packages which can be installed upon request.

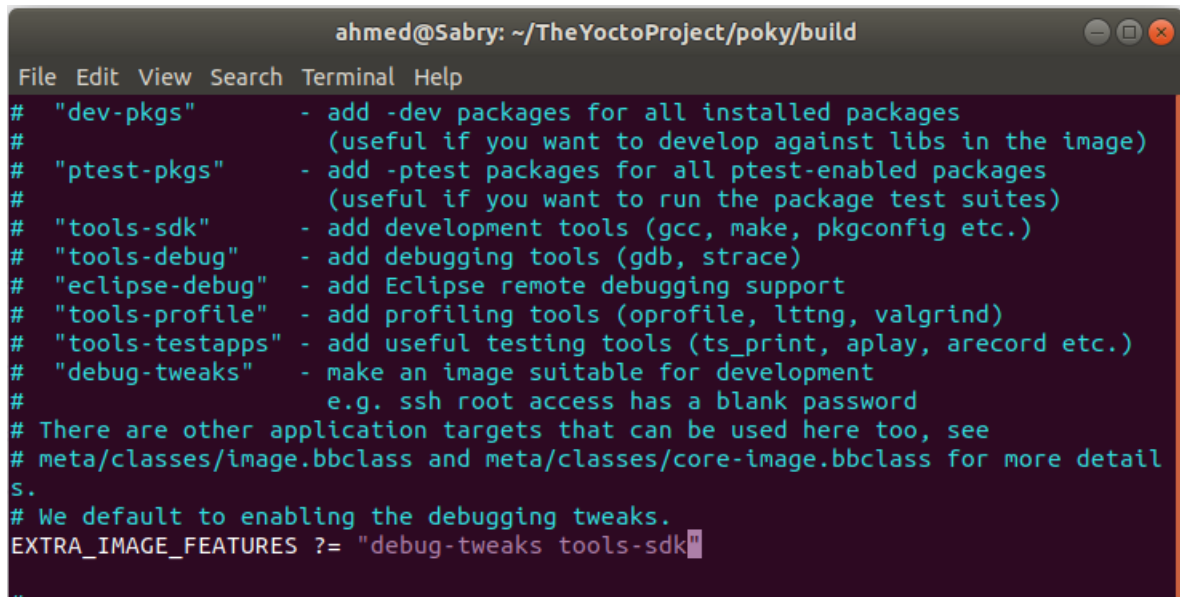
## 2.5 Preparing the SDKs for your application development

In order to develop an applications for your target, you might use one of the following way to achieve that:

1. Native Development In native development we compile directly on the target machine by a native tool-chain, which means if we compiling for x86, the we use compiler natively and compile on our x86 target. as a

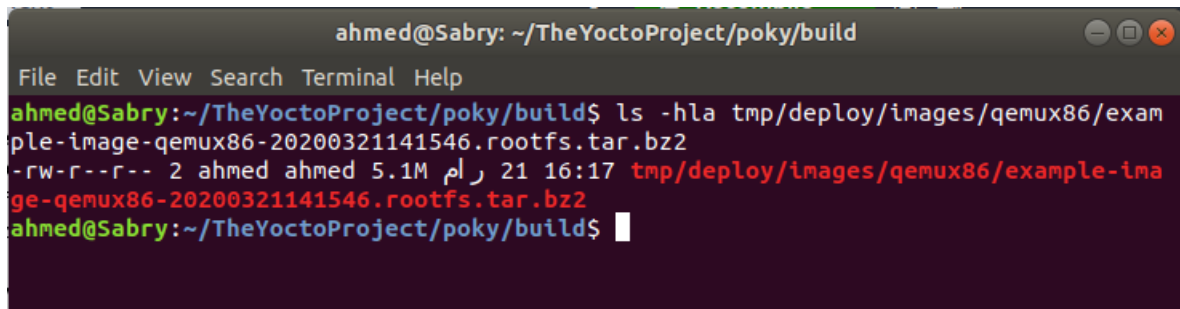


summary the target is the host machine for the tool chain. In order to include the tool chain to our image, we need to edit **local.conf** file to add a new **"tools-sdk"** as a new feature.



```
ahmed@Sabry: ~/TheYoctoProject/poky/build
File Edit View Search Terminal Help
# "dev-pkgs" - add -dev packages for all installed packages
# (useful if you want to develop against libs in the image)
# "ptest-pkgs" - add -ptest packages for all ptest-enabled packages
# (useful if you want to run the package test suites)
# "tools-sdk" - add development tools (gcc, make, pkgconfig etc.)
# "tools-debug" - add debugging tools (gdb, strace)
# "eclipse-debug" - add Eclipse remote debugging support
# "tools-profile" - add profiling tools (oprofile, lttng, valgrind)
# "tools-testapps" - add useful testing tools (ts_print, aplay, arecord etc.)
# "debug-tweaks" - make an image suitable for development
# e.g. ssh root access has a blank password
# There are other application targets that can be used here too, see
# meta/classes/image.bbclass and meta/classes/core-image.bbclass for more detail
# We default to enabling the debugging tweaks.
EXTRA_IMAGE_FEATURES ?= "debug-tweaks tools-sdk"
```

after we bitbake our image, here we have our core-image-minimal with only "5.1 MB".



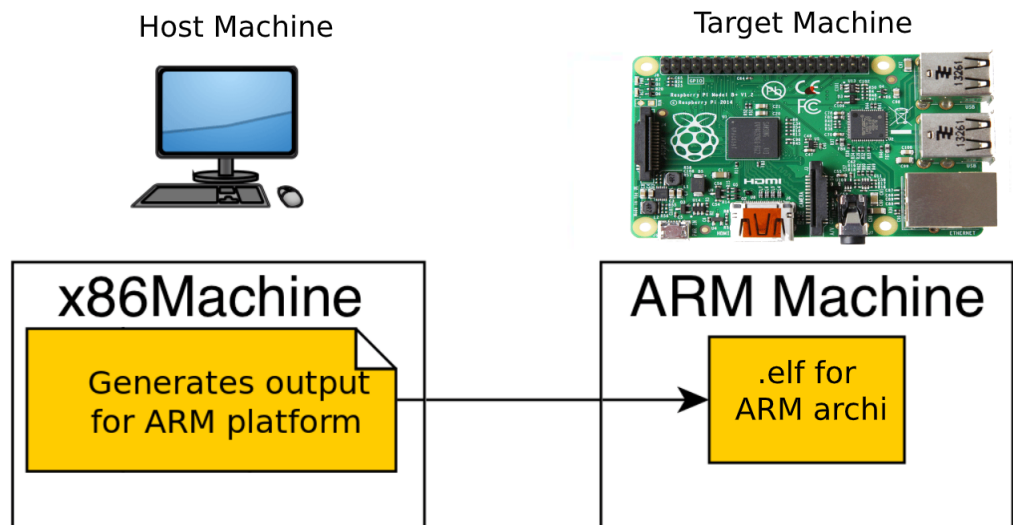
```
ahmed@Sabry: ~/TheYoctoProject/poky/build
File Edit View Search Terminal Help
ahmed@Sabry:~/TheYoctoProject/poky/build$ ls -hla tmp/deploy/images/qemu86/example-image-qemu86-20200321141546.rootfs.tar.bz2
-rw-r--r-- 2 ahmed ahmed 5.1M 21 16:17 tmp/deploy/images/qemu86/example-image-qemu86-20200321141546.rootfs.tar.bz2
ahmed@Sabry:~/TheYoctoProject/poky/build$
```

and now after including a basic tools, we have our image with "70 MB" size!, which 14 times the original size.

```
ahmed@Sabry: ~/TheYoctoProject/poky/build/tmp/deploy/images/qemux86
File Edit View Search Terminal Help
ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/images/qemux86$ ls -hla exam
ple-image-qemux86-20200321152313.rootfs.tar.bz2
-rw-r--r-- 2 ahmed ahmed 70M 21 18:00 example-image-qemux86-20200321152313.r
ootfs.tar.bz2
ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/images/qemux86$
```

Unfortunately, as you see the size of the our image is very big as we included the complete tool chain into it, this sounds okay if we will use our image on an efficient PC, but in the world of embedded systems this is not accepted at all, as the footprint of the image should be less, and the processing power needed for compilation might not be that efficient. so we need to introduce the other way which using a cross-compiler.

2. Cross-Development using a cross-compiler on a host system.  
simply you develop and build on powerful "host" machine for your target machine, then send the output to be executed on the "target" machine.



first let's change our machine to ARM in order to generate an SDK for it. we can do this by editing **local.conf** file to comment "**MACHINE ??= qemux86**" and uncomment "**MACHINE ?= qemuarm**".

```
ahmed@Sabry: ~/TheYoctoProject/poky/build
File Edit View Search Terminal Help
# of emulated machines available which can boot and run in the QEMU emulator:
#
#MACHINE ?= "qemuarm"
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemumips"
#MACHINE ?= "qemumips64"
#MACHINE ?= "qemuppc"
#MACHINE ?= "qemux86"
#MACHINE ?= "qemux86-64"
#
# There are also the following hardware board target machines included for
# demonstration purposes:
#
#MACHINE ?= "beaglebone"
#MACHINE ?= "genericx86"
#MACHINE ?= "genericx86-64"
#MACHINE ?= "mpc8315e-rdb"
#MACHINE ?= "edgerouter"
#
# This sets the default machine to be qemux86 if no other machine is selected:
MACHINE ??= "qemux86"
#
34,1 7%
```

then let's get our ARM based cross-sdk:

- (a) Use populate\_sdk command to fetch your SDK  
You can simply use the following command in order to have an SDK  
for your target

Listing 22: building using bitbake

```
# bitbake <recipe-image> -c populate_sdk
ahmed@Sabry:~/TheYoctoProject/poky/build$ bitbake example-image -c populate_sdk
```

and it's simply calls the do\_populate task in order to fetch the SDK  
for your target. and the output is an archived SDK in the following  
path "build/tmp/ deploy/sdk":

Listing 23: STD SDK location

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ ls tmp/deploy/sdk/
poky-glibc-x86_64-example-image-armv5e-toolchain-2.2.4.host.manifest
poky-glibc-x86_64-example-image-armv5e-toolchain-2.2.4.sh
poky-glibc-x86_64-example-image-armv5e-toolchain-2.2.4.target.manifest

ahmed@Sabry:~/TheYoctoProject/poky/build$ ls -hla tmp/deploy/sdk/poky-glibc-x86_64-example-image-
```

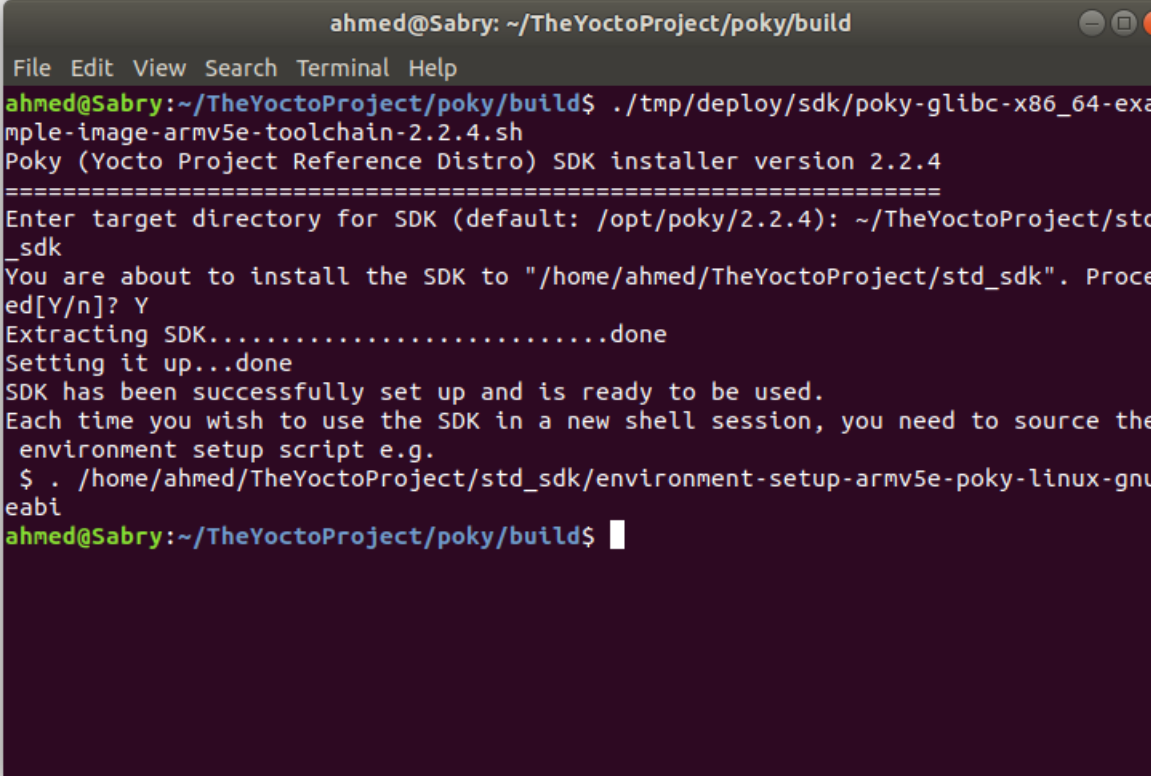
```
—rwxr—xr—x 2 ahmed ahmed 115M 25 18:57 tmp/deploy/sdk/poky—glibc—x86_64—example—image—armv5e-  
ahmed@Sabry:~/TheYoctoProject/poky/build$
```

(b) Extract your SDK

I will extract my STD SDK in " /TheYoctoProject/std\_sdk", and I will do that by calling the compressed SDK ".sh" file in "/build/tmp-deploy/sdk"

Listing 24: Extract the SDK

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ ./tmp/deploy/sdk/poky—glibc—x86_64—example—image—armv5e-
```

A terminal window titled "ahmed@Sabry: ~/TheYoctoProject/poky/build" with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the execution of the script "poky-glibc-x86\_64-example-image-armv5e-toolchain-2.2.4.sh". The output includes: "Poky (Yocto Project Reference Distro) SDK installer version 2.2.4", a separator line, a prompt for the target directory (default: /opt/poky/2.2.4) which is answered with "~/TheYoctoProject/std\_sdk", a confirmation prompt "You are about to install the SDK to "/home/ahmed/TheYoctoProject/std\_sdk". Proceed[Y/n]? Y", and then "Extracting SDK.....done" and "Setting it up...done". A message states "SDK has been successfully set up and is ready to be used." and provides instructions to source the environment setup script, showing the command "\$ . /home/ahmed/TheYoctoProject/std\_sdk/environment-setup-armv5e-poky-linux-gnueabi". The prompt returns to "ahmed@Sabry:~/TheYoctoProject/poky/build\$".

```
ahmed@Sabry: ~/TheYoctoProject/poky/build  
File Edit View Search Terminal Help  
ahmed@Sabry:~/TheYoctoProject/poky/build$ ./tmp/deploy/sdk/poky-glibc-x86_64-example-image-armv5e-toolchain-2.2.4.sh  
Poky (Yocto Project Reference Distro) SDK installer version 2.2.4  
=====  
Enter target directory for SDK (default: /opt/poky/2.2.4): ~/TheYoctoProject/std_sdk  
You are about to install the SDK to "/home/ahmed/TheYoctoProject/std_sdk". Proceed[Y/n]? Y  
Extracting SDK.....done  
Setting it up...done  
SDK has been successfully set up and is ready to be used.  
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.  
$ . /home/ahmed/TheYoctoProject/std_sdk/environment-setup-armv5e-poky-linux-gnueabi  
ahmed@Sabry:~/TheYoctoProject/poky/build$
```

(c) Compile for your target

In order to compile for your target, you need to execute the output script "environment-setup-armv5e-poky-linux-gnueabi" in the extracted SDK. this script will setup the environmental variables for the cross-compilation for you.

Listing 25: Setup the environmental variables

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ source /home/ahmed/TheYoctoProject/std_sdk/environment-
```

then simply, use **cmake** to compile for your target:

Listing 26: Compile using the cross compiler

```
ahmed@Sabry:~/TheYoctoProject/sources/simple_cmake$ cmake .
-- The C compiler identification is GNU 6.4.0
-- The CXX compiler identification is GNU 6.4.0
-- Check for working C compiler: /home/ahmed/TheYoctoProject/std_sdk/sysroots/x86_64-pokysdk-linux/u
-- Check for working C compiler: /home/ahmed/TheYoctoProject/std_sdk/sysroots/x86_64-pokysdk-linux/u
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info -- done
-- Detecting C compile features
-- Detecting C compile features -- done
-- Check for working CXX compiler: /home/ahmed/TheYoctoProject/std_sdk/sysroots/x86_64-pokysdk-linux/u
-- Check for working CXX compiler: /home/ahmed/TheYoctoProject/std_sdk/sysroots/x86_64-pokysdk-linux/u
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info -- done
-- Detecting CXX compile features
-- Detecting CXX compile features -- done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ahmed/TheYoctoProject/sources/simple_cmake
ahmed@Sabry:~/TheYoctoProject/sources/simple_cmake$ make
Scanning dependencies of target say_hello
[ 50%] Building C object CMakeFiles/say_hello.dir/main.c.o
[100%] Linking C executable say_hello
[100%] Built target say_hello
ahmed@Sabry:~/TheYoctoProject/sources/simple_cmake$ file say_hello
say_hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-
ahmed@Sabry:~/TheYoctoProject/sources/simple_cmake$
```

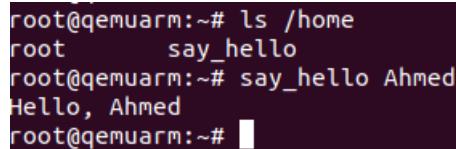
(d) Try on the target machine

Finally, we need to execute the output on our target, We will use the **scp** command in order to transfer the binary file to the target.

Listing 27: scp command to transfer our binary to the image

```
ahmed@Sabry:~/TheYoctoProject/sources/simple_cmake$ scp say_hello root@192.168.7.2:/home
```

and finally, our application is working as expected



```
root@qemuarm:~# ls /home
root      say_hello
root@qemuarm:~# say_hello Ahmed
Hello, Ahmed
root@qemuarm:~#
```

Notes:

- i. In order to use scp command, you need to configure your image to include an ssh server, then you need to edit the **local.conf** file to include openssh for example. here I am using ""

```
ahmed@Sabry: ~/TheYoctoProject/poky/build
File Edit View Search Terminal Help
# "eclipse-debug" - add Eclipse remote debugging support
# "tools-profile" - add profiling tools (oprofile, ltng, valgrind)
# "tools-testapps" - add useful testing tools (ts_print, aplay, arecord etc.)
# "debug-tweaks" - make an image suitable for development
#                   e.g. ssh root access has a blank password
# There are other application targets that can be used here too, see
# meta/classes/image.bbclass and meta/classes/core-image.bbclass for more details.
# We default to enabling the debugging tweaks.
EXTRA_IMAGE_FEATURES ?= "debug-tweaks ssh-server-dropbear"
#
# Additional image features
#
# The following is a list of additional classes to use when building images with
# enable extra features. Some available options which can be included in this
# variable
# are:
# - 'buildstats' collect build statistics
# - 'image-mklibs' to reduce shared library files size for an image
# - 'image-prelink' in order to prelink the filesystem image
# - 'image-swab' to perform host system intrusion detection
-- INSERT -- 141,58
```

- ii. You might use devtool deploy command to directly build and deploy to your target as we covered previously.

### 2.5.1 Types of SDKs

With *Yocto*, there are different types of SDKs:

- (a) Standard SDK Which contains just the needed tool chains to build your application. as we already covered how to fetch it and used it in the previous section.
- (b) Extensible SDK It extends the STD SDK to add more feature mainly you can say that it include a yocto image to create, edit recipes.
  - Contains the Toolchain
  - Contains everything to re-create images, also it have the devtool command.

You can fetch the EXT SDK by executing the following command:

Listing 28: populating the EXT SDK

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ bitbake example--image --c populate_sdk_ext
```

and you might notice the difference in size between the STD SDK and the EXT SDK!.

Listing 29: STD SDK size V.s. EXT SDK Size

```
ahmed@Sabry:~/TheYoctoProject/poky/build$ cd tmp/deploy/sdk/
ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/sdk$ ls
poky-glibc-x86_64-example-image-armv5e-toolchain-2.2.4.host.manifest
poky-glibc-x86_64-example-image-armv5e-toolchain-2.2.4.sh
poky-glibc-x86_64-example-image-armv5e-toolchain-2.2.4.target.manifest
poky-glibc-x86_64-example-image-armv5e-toolchain-ext-2.2.4.host.manifest
poky-glibc-x86_64-example-image-armv5e-toolchain-ext-2.2.4.sh
poky-glibc-x86_64-example-image-armv5e-toolchain-ext-2.2.4.target.manifest
x86_64-buildtools-nativesdk-standalone-2.2.4.host.manifest
x86_64-buildtools-nativesdk-standalone-2.2.4.sh
x86_64-buildtools-nativesdk-standalone-2.2.4.target.manifest

ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/sdk$ ls -hla poky-glibc-x86_64-example-image-
-rwxr-xr-x 2 ahmed ahmed 115M  25 18:57 poky-glibc-x86_64-example-image-armv5e-toolchain-2.2.4.

ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/sdk$ ls -hla poky-glibc-x86_64-example-image-
-rwxr-xr-x 2 ahmed ahmed 969M  25 21:39 poky-glibc-x86_64-example-image-armv5e-toolchain-ext-2.2.4.
ahmed@Sabry:~/TheYoctoProject/poky/build/tmp/deploy/sdk$
```

### 3 Packaging

## 4 building an image for Raspberry Pi Board

In the following section we will make a simple steps to have a ready to run image for raspberry pi. also we will be using it's cross-toolchain in order to build our pi applications, then to see the different ways to deploy the resultant image to run on our pi.

And as you will see that we kind of repeating the previous sections to have this image.

1. Downloading the raspberry pi meta layer.

Listing 30: clone rpi meta-layer

```
ahmed@Sabry:~/TheYoctoProject/poky$ git clone git://git.yoctoproject.org/meta-raspberrypi -b morty

#Now we have the meta-raspberry pi layer
ahmed@Sabry:~/TheYoctoProject/poky$ ls
bitbake meta-mylayer meta-yocto README.hardware
build meta-poky meta-yocto-bsp scripts
documentation meta-raspberrypi oe-init-build-env
LICENSE meta-selftest oe-init-build-env memres
meta meta-skeleton README
```

```
# and we have the configuration of our pi machines
ahmed@Sabry:~/TheYoctoProject/poky$ ls meta-raspberrypi/conf/machine/*.conf
meta-raspberrypi/conf/machine/raspberrypi0.conf
meta-raspberrypi/conf/machine/raspberrypi0-wifi.conf
meta-raspberrypi/conf/machine/raspberrypi2.conf
meta-raspberrypi/conf/machine/raspberrypi3-64.conf
meta-raspberrypi/conf/machine/raspberrypi3.conf
meta-raspberrypi/conf/machine/raspberrypi4-64.conf
meta-raspberrypi/conf/machine/raspberrypi4.conf
meta-raspberrypi/conf/machine/raspberrypi-cm3.conf
meta-raspberrypi/conf/machine/raspberrypi-cm.conf
meta-raspberrypi/conf/machine/raspberrypi.conf
```

Note: Here I am using *Poky* with **morty** version, so make sure to branch the raspberry pi for the **morty** version (morty branch of rpi meta-layer) as well to avoid compatibility issue. otherwise I got some issues.

## 2. configuring the environment

- Edit the bblayer.conf file to include the pi layer

Listing 31: add pi meta-layer to bblayer.conf

```
ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ vim conf/bblayers.conf
```





```

../meta-raspberrypi/conf/machine/raspberrypi4-64.conf
../meta-raspberrypi/conf/machine/raspberrypi4.conf
../meta-raspberrypi/conf/machine/raspberrypi-cm3.conf
../meta-raspberrypi/conf/machine/raspberrypi-cm.conf
../meta-raspberrypi/conf/machine/raspberrypi.conf
ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$

```

And as we are working on raspberry 1 model B, then we will change the **MACHINE="raspberrypi"** variable in **local.conf**.

Listing 34: choose our new machine from local.conf

```

ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ vim conf/local.conf

```

```

#
#MACHINE ?= "beaglebone"
#MACHINE ?= "genericx86"
#MACHINE ?= "genericx86-64"
#MACHINE ?= "mpc8315e-rdb"
#MACHINE ?= "edgerouter"
#
# This sets the default machine to be qemu86 if no other machine is selected:
#MACHINE ??= "qemu86"
MACHINE ?= "raspberrypi"
#

```



**Note: Note: All machines with the BCM SoC number: ?**

```

raspberrypi (BCM2835)
raspberrypi0 (BCM2835)
raspberrypi0-wifi (BCM2835)
raspberrypi2 (BCM2836 or BCM2837 v1.2+)
raspberrypi3 (BCM2837)
raspberrypi4 (BCM2838)
raspberrypi-cm (BCM2835)
raspberrypi-cm3 (BCM2837)

```

### 3. Bitbake the image

Listing 35: bitbake the raspberry pi

```

ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ bitbake rpi-basic-image

ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ ls tmp/deploy/images/raspberrypi/*sdimg
tmp/deploy/images/raspberrypi/rpi-basic-image-raspberrypi-20200326205401.rootfs.rpi-sdimg
tmp/deploy/images/raspberrypi/rpi-basic-image-raspberrypi.rpi-sdimg

```

And the output image will be in the ”**rpi-build/tmp/images/raspberrypi**” folder.

```
ahmed@Sabry: ~/TheYoctoProject/poky/rpi-build
File Edit View Search Terminal Help
ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ ls tmp/deploy/images/raspberrypi/*
sding
tmp/deploy/images/raspberrypi/rpi-basic-image-raspberrypi-20200326205401.rootfs.
rpi-sding
tmp/deploy/images/raspberrypi/rpi-basic-image-raspberrypi.rpi-sding
ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$
```

4. Deploy to our SD card First look on which sdx the sd card has been mounted in order to use the **dd** command with it:

Listing 36: Check SD card mount point

```
ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ dmesg | tail
[84842.451867] sd 6:0:0:0: [sdc] Write Protect is off
[84842.451875] sd 6:0:0:0: [sdc] Mode Sense: 0b 00 00 08
[84842.452979] sd 6:0:0:0: [sdc] No Caching mode page found
[84842.452991] sd 6:0:0:0: [sdc] Assuming drive cache: write through
[84842.458785] sdc: sdc1 sdc2
[84842.462429] sd 6:0:0:0: [sdc] Attached SCSI removable disk
[84843.066728] EXT4-fs (sdc2): mounting ext3 file system using the ext4 subsystem
[84843.974810] EXT4-fs (sdc2): recovery complete
[84843.976497] EXT4-fs (sdc2): mounted filesystem with ordered data mode. Opts: (null)
[84844.002352] FAT-fs (sdc1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.

# You can try also lsblk, or fdisk -l commands
```

FYI: you can use **lsblk** command also:

Listing 37: Check SD card mount point

```
ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ lsblk
sda 8:0 0 465.8G 0 disk
|--sda1 8:1 0 450M 0 part
|--sda2 8:2 0 100M 0 part /boot/efi
|--sda3 8:3 0 16M 0 part
|--sda4 8:4 0 145.1G 0 part
|--sda5 8:5 0 803M 0 part
|--sda6 8:6 0 249G 0 part
|--sda7 8:7 0 70.3G 0 part /home
```

```
sdb 8:16 0 22.4G 0 disk
|--sdb1 8:17 0 476M 0 part /boot
|--sdb2 8:18 0 18.6G 0 part /
|--sdb3 8:19 0 3.3G 0 part [SWAP]

sdc 8:32 1 7.2G 0 disk
|--sdc1 8:33 1 1G 0 part /media/ahmed/BOOT
|--sdc2 8:34 1 6.2G 0 part /media/ahmed/ROOTFS
```

Here we use the **dd** command in order to partition our SD card for our image.

Listing 38: building using bitbake

```
ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ sudo dd if=tmp/deploy/images/raspberrypi/rpi-basic-image-ra
```

then you can check your **dd** command output as follow:

Listing 39: building using bitbake

```
ahmed@Sabry:~/TheYoctoProject/poky/rpi-build$ lsblk
sdc 8:32 1 7.2G 0 disk
sdc1 8:33 1 40M 0 part /media/ahmed/raspberrypi
sdc2 8:34 1 76M 0 part /media/ahmed/dc6e7cac-b6d9-4658-8c50-c64d188006
```

5. Try the image and interface with it through **minicom** Now after we have finished the building, and we transferred the output image to our SD card, let's try to boot it up, I am using here raspberry pi 1 Model B, and I will interface with it through serial interface using FTDI to USB driver, and a minicom SW for serial communication.

You need to notice that you might use other ways to work with your image, like directly connect it using HDMI with a screen and keyboard, or to communicate with it using Ethernet (or your home router) + Putty on windows platform.

finally the communication with our pi will be based on the following steps:

- Prepare your environment to work with our serial interface

Just plug your FTDI usb to your computer and discover which device file it is associated with.

Listing 40: Check the FTDI device file

```
ahmed@Sabry:~$ sudo apt-get install minicom

ahmed@Sabry:~$ dmesg | tail
[86312.323400] usb 3-1: Product: FT232R USB UART
[86312.323403] usb 3-1: Manufacturer: FTDI
[86312.323406] usb 3-1: SerialNumber: A700e9Wx
```

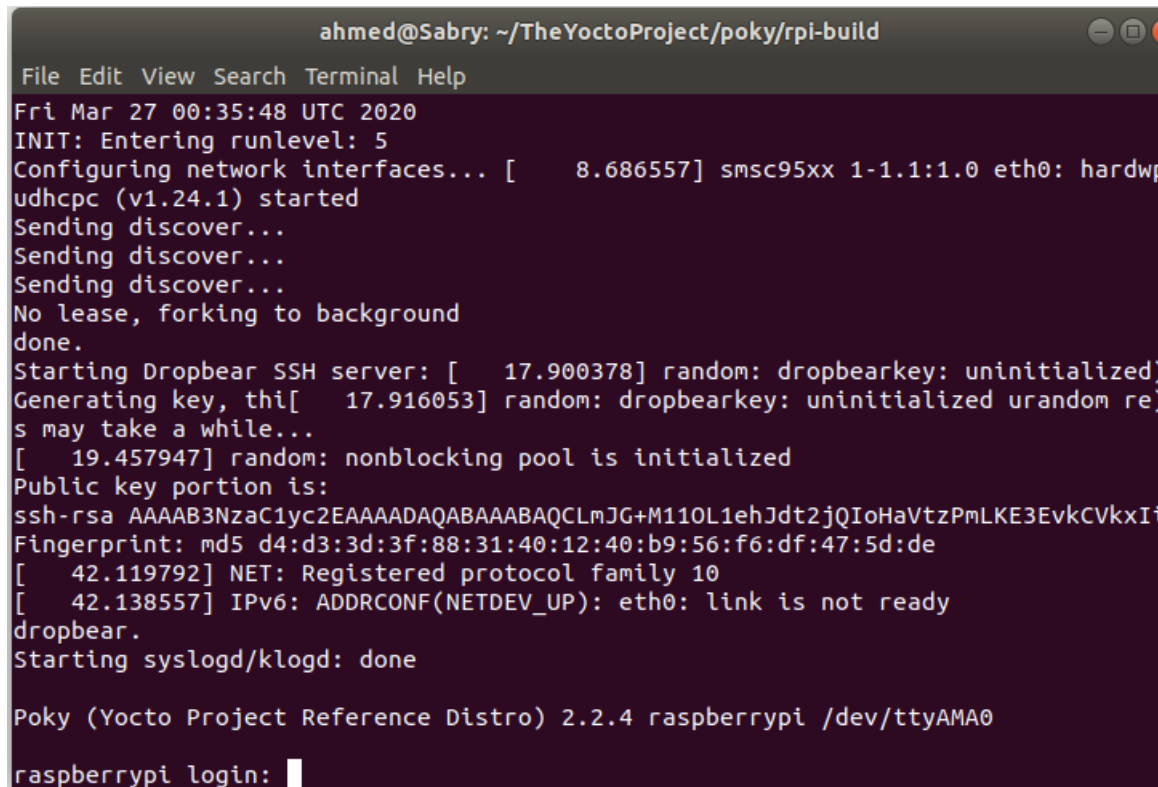
```
[86312.351457] usbcore: registered new interface driver usbserial_generic
[86312.351520] usbserial: USB Serial support registered for generic
[86312.359599] usbcore: registered new interface driver ftdi_sio
[86312.359618] usbserial: USB Serial support registered for FTDI USB Serial Device
[86312.359753] ftdi_sio 3-1:1.0: FTDI USB Serial Device converter detected
[86312.359865] usb 3-1: Detected FT232RL
[86312.360556] usb 3-1: FTDI USB Serial Device converter now attached to ttyUSB0
```

- Use minicom to access the device  
Now after we are aware on which device file we will connect,

Listing 41: building using bitbake

```
ahmed@Sabry:~$ sudo minicom -b 115200 -D /dev/ttyUSB0
```

and here we are, our Raspberry pi is up and running with our new image:



```
ahmed@Sabry: ~/TheYoctoProject/poky/rpi-build
File Edit View Search Terminal Help
Fri Mar 27 00:35:48 UTC 2020
INIT: Entering runlevel: 5
Configuring network interfaces... [ 8.686557] smsc95xx 1-1.1:1.0 eth0: hardware
udhcpd (v1.24.1) started
Sending discover...
Sending discover...
Sending discover...
No lease, forking to background
done.
Starting Dropbear SSH server: [ 17.900378] random: dropbearkey: uninitialized
Generating key, thi[ 17.916053] random: dropbearkey: uninitialized urandom re
s may take a while...
[ 19.457947] random: nonblocking pool is initialized
Public key portion is:
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCLmJG+M110L1ehJdt2jQIoHaVtzPmLKE3EvkCVkxI
Fingerprint: md5 d4:d3:3d:3f:88:31:40:12:40:b9:56:f6:df:47:5d:de
[ 42.119792] NET: Registered protocol family 10
[ 42.138557] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
dropbear.
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 2.2.4 raspberrypi /dev/ttyAMA0
raspberrypi login: █
```

## 5 Appendix

### 5.1 Debugging the build messages