INTRODUCTION TO YOCTO

GOAL

We all have used Linux-powered systems at some point in our lives. In fact, most of us use such systems on a daily basis and some of us use them almost all the time!

The Linux kernel is used in a lot of gadgets around us – our smartphones, tablet computers, personal computers, medical instruments, car infotainment systems, GPS navigation systems (replaced mostly by smartphones today), set-top boxes, Wi-Fi routers and so on. A lot of items that we don't use directly but are used in systems that we may interact daily also use Linux kernels like industrial gateway systems, telecommunication equipment, a lot of data servers across the globe, etc. But all these systems don't use the same processors! Some of these are built using extremely powerful multi-core processors, some use a simpler processor at their heart.

The sheer volume of systems around us that are Linux-powered tells us that there must be something that makes it *easy* for the manufacturers and designers of these systems to use it, isn't it? Think of it – if it wasn't easy to use it, someone else somewhere in the world would have designed an alternative system. That's just how the tech world works!

One of the key steps in being *easily* able to deploy Linux-based systems is to be able to make system images that use Linux kernel – *easily*. Considering that such systems are complex and support a wide variety of processors, making it *easy* sounds like an enormous task which is achieved by the **Yocto Project!**

WHAT THE YOCTO PROJECT IS NOT…

Before we see what the Yocto Project *is*, let us know what it **is not**.

The Yocto Project is **not an SDK** that you can use for your hardware machine – it can be used to build one!

The Yocto Project is **not a system binary image** that you can deploy on your hardware machine – it can be used to build one!

The Yocto Project is **not a linux distribution** that you can use to install to your hardware machine – it can be used to build an extremely tailored one for your resource-constrained hardware machine.

WHAT THE YOCTO PROJECT IS…

The Yocto Project is an open-source collaboration project that helps developers create custom Linux-based systems for embedded products, regardless of the hardware architecture. The project provides a *flexible set of tools and a space* where embedded developers *worldwide* can share technologies, software stacks, configurations and best practices which can be used to create *tailored* Linux images for embedded devices.

The Yocto Project combines, maintains and validates three key development elements.

1. A set of integrated tools to make working with embedded Linux successful, including tools for automated building and testing, processes for board support and license compliance, and component information for custom Linux-based embedded operating systems

2. A reference embedded distribution (called Poky)

3. The OpenEmbedded build system, co-maintained with the OpenEmbedded Project

WHAT IS POKY?

Poky (pronounced Pock-ee) is a reference embedded distribution and a reference test configuration created to:

1. *Provide a base* level functional distro which can be used to illustrate how to customize a distribution
2. *To test* the Yocto Project components, Poky is used to validate Yocto Project
3. As a *vehicle for users to download Yocto* Project. Poky is **not** *a product level distro*, but a good starting point for customization. Poky is an integration layer on top of oe-core.

WHAT IS OE-CORE?

oe-core or OpenEmbedded-Core is meta-data comprised of foundation *recipes*, classes and associated files that are meant to be common among many different OpenEmbedded-derived systems, including the Yocto Project. It is a *curated subset* of an original repository developed by the OpenEmbedded community which has been pared down into a smaller, core set of *continuously validated* recipes resulting in a *tightly controlled* and a *quality-assured* core set of recipes.

Simply put, *oe-core is a quality assured core foundational recipes* that Poky uses to be able to generate a *good and working* base Linux image.

We just said that oe-core is a set of recipes.

WHAT IS A RECIPE?

Recipe is the most common form of *metadata*. A recipe will contain a list of settings and tasks (instructions) for building packages which are then used to build the binary image. A recipe describes where you *get source code and which patches to apply*. Recipes describe dependencies for libraries or for other recipes, as well as configuration and compilation options.

They are stored in *layers*. In fact, the layered nature of Poky is what makes it extremely scalable, versatile and easy to adapt to a variety of systems. For example, you will have all recipes pertaining to networking in one layer, all recipes related to your application in another, a dedicated layer for your graphics subsystems, and so on!

Similarly, there are files called configuration files. These are files which hold global definitions of variables, user defined variables and hardware configuration information. They tell the build system what to build and put into the image to support a particular platform.

Configuration Files and Recipes are often referred to as the *metadata* in the Poky build system. Other than these, commands and data that are used to build the image using the recipes and configuration files also constitute metadata.

We have oe-core which contains the validated metadata for a good image. But what do you do with these recipes? Enter BitBake…

WHAT IS BITBAKE?

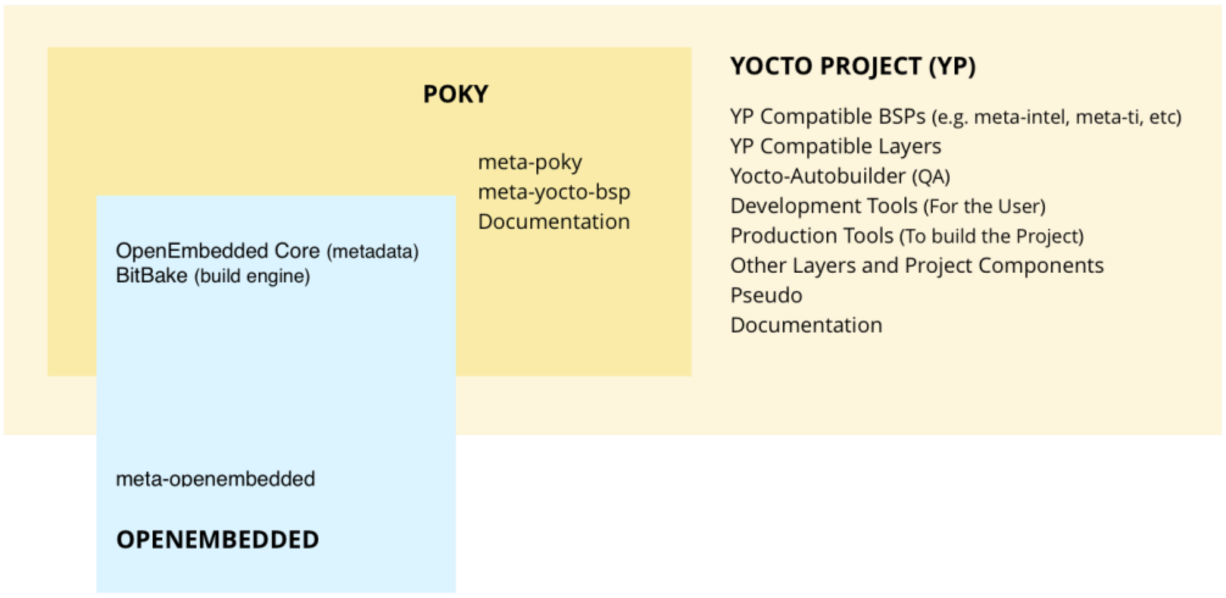
BitBake is a scheduler and execution engine which *parses* instructions (recipes) and configuration data. It then *creates a dependency tree* to order the compilation, *schedules* the compilation of the included code, and finally, *executes the building* of the specified, custom Linux image (distribution). BitBake is a make-like build tool. BitBake recipes specify how a particular package is built. They include all the package dependencies, source code locations, configuration, compilation, build, install, and remove instructions.

During the build process dependencies are tracked and native or cross-compilation of the package is performed. As a first step in a cross-build setup, the framework will attempt to create a cross-compiler toolchain suited for the target platform.

Remember we talked about layers above? The BitBake parser is the one that ensures that a layer at the top can override settings in the lower layer thus avoiding any conflicts as the parser moves down the layers!

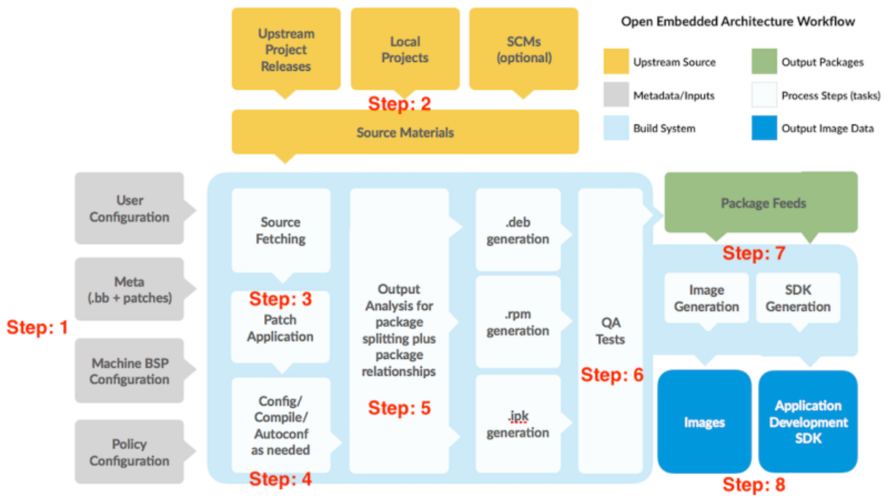
The BitBake engine uses the metadata to create a build output called *packages*. These packages come together to give what is the ultimate goal – the final image.

Below is a good **high-level representation** of the Yocto Project components:


POKY
meta-poky
meta-yocto-bsp
Documentation
YOCTO PROJECT (YP)
YP Compatible BSPs (e.g. meta-intel, meta-ti, etc)
YP Compatible Layers
Yocto-Autobuilder (QA)
Development Tools (For the User)
Production Tools (To build the Project)
Other Layers and Project Components
Pseudo
Documentation
OpenEmbedded Core (metadata)
BitBake (build engine)
meta-openembedded
OPENEMBEDDED

YOCTO – DEVELOPMENT WORKFLOW

There are a *lot* of components that play a significant role in the creation of the image. But what does the overall workflow look like? Let’s find out.



- **Step 1:** It all starts with getting the source code? No! First, the developer has to decide the various high-level configurations like what machine the image is to be built for, any special configuration data needed for the build, types of images to be built, etc.

- **Step 2:** Once the configuration is ready, next comes the source code. The source code can be in the form of tarballs, fetched from project releases or from Git or SVN or even be locally added to the workspace.

- **Step 3:** Often, there may be a need to apply patches to the source code to tailor the software component to the desired machine or this specific image. These patches are applied next.

- **Step 4:** What do you do with source code? You build! That indeed is the next step. Most common build tools like autotools, cmake, etc. are supported. BitBake takes care of doing the necessary configuration and compilation.

- **Step 5:** The outputs of this build process are then placed into a temporary staging area where the packaging is done like .deb, .rpm, .ipk, etc. This packaging info is one of the configurations specified at the very beginning before we obtained the source code.

- **Step 6:** Next comes the QA process. Although it is incorrect to say that this is the first stage where the QA happens – it is more correct to say that QA activities happen throughout the process – especially during the BitBake engine's operation.

- **Step 7:** Once the binary packages are ready, all that is necessary is to create a package feed that is suitable for the image that is requested.

- **Step 8:** This package feed culminates with the creation of the final image! Easy-peasy! By the way, a Linux image is not all that the build process can create. The build process can also optionally be used to generate an SDK that can be used to develop and build applications for the machine that is running the very image you generate and load on that machine. This is extremely handy and takes away tons of complications that may result from not using a tailored SDK – especially in a diverse team of developers!

This was a brief introduction to the world of Yocto. In the next section, we will see how to prepare a host computer to use the Yocto Project.

SETTING UP UBUNTU HOST

GOAL

In the last section, we introduced the Yocto Project and discussed topics like open-embedded, poky, BitBake, recipes and so on.

In this section, we will discuss the process of setting up an Ubuntu host to perform your Yocto builds. According to the documentation of the Yocto Project, Ubuntu is a supported Linux distribution on the host side, although not all versions are equally supported. It is recommended to use LTS versions as your host build PC. You can see the list of supported Linux distributions here. In this section, we talk about setting up your **PC running Ubuntu 20.04 LTS** as the operating system.

ABSOLUTE MINIMUM REQUIREMENTS

FREE MEMORY REQUIREMENT

It is recommended to have at least 50 GB free memory on your hard disk. The larger free memory, the more future proof your builds would be!

SOFTWARE TOOLS REQUIREMENT

The most essential software tools for performing Yocto builds on your PC, you need the below. These are common across all kinds of hosts including Ubuntu 20.04.

1. **Git:** As we saw in the last section, git is used to fetch the source code from various software repositories.
2. **Tar:** Most source code is downloaded as a tarball to minimise data usage and make the fetch faster.
3. **Python3:** Although Ubuntu should already have Python3 installed, it does not hurt to check that the installation is good and the version is >= Python 3.6.0
4. **GCC:** The toolchain used to perform the *enormous* amount of builds as a part of the package generation and finally image generation process is the GNU Compiler Collection (GCC) tool.

To install these software tools, execute the below commands. For the second command, you *may* need to press the Enter/Return key when prompted by the package manager to fetch the packages and install them.

```
sudo apt update

sudo apt install git git-lfs tar python3 python3-pip python-is-python3 gcc
```

Finally, install the repo utility:

```
mkdir ~/bin

curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > ~/bin/repo

chmod a+x ~/bin/repo

export PATH=$PATH:~/bin
```

UBUNTU-SPECIFIC PACKAGE REQUIREMENTS

ESSENTIAL PACKAGES REQUIREMENT

Other than the minimum requirements listed above, you also need certain host packages to perform yocto builds successfully. These packages are required in the various steps (1 to 8) as listed in the last section.

To install these packages, execute the below command and press the Enter/Return key when prompted by the package manager to fetch the packages and install them.

```
sudo apt install gawk wget git diffstat unzip texinfo gcc build-essential chrpath socat
cpio python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping python3-git
python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit mesa-common-dev
zstd liblz4-tool
```

You may have noticed that the python3 and python3-pip are present in this command as well – this is not a problem as the package manager will simply skip them if they are already installed!

DOCUMENTATION PACKAGES REQUIREMENT (OPTIONAL)

Yocto releases also contain the source of the documentation. You can generate this documentation yourself and also optionally, add your own documentation into the mix using – you guessed it – your *own* meta layer! Some packages are needed to be able to build the Yocto documentation.

To install these packages, execute the below command and press the Enter/Return key when prompted by the package manager to fetch the packages and install them.

```
sudo apt install make python3-pip

sudo pip3 install sphinx sphinx_rtd_theme pyyaml
```

Depending on your internet speed, it can take anywhere between 5 to 15 minutes to perform the above steps. Once done, you have successfully set up your Ubuntu 20.04 PC for your first Yocto build!

In the next section, we will download the Yocto sources and try to do our first basic Yocto build and run the resulting image generated at the end of this build.

BUILD & RUN YOUR FIRST EVER IMAGE ON QEMU

GOAL

In the previous sections, we have introduced you to Yocto and showed how to set up your Ubuntu PC for doing Yocto builds.

In this section, we get right into the action and build our first ever Yocto image. We highly recommend doing this build to everyone – irrespective of the actual machine you want to build for as the resulting image is pretty handy to quickly check for features that you originally want in the image!

CREATE A WORKING DIRECTORY

To follow this Lab, you will need: The very first step is to create a working directory for yocto. It is not mandatory but considered a good practice to have a dedicated working directory rather than do everything in a random place on your PC.

Let us assume you are in your home directory. Execute the below commands to create a working directory.

```
mkdir yocto

cd yocto
```

DOWNLOAD POKY AND SELECT THE DESIRED RELEASE

We are almost there! The next step is to download the Poky source code and then checkout the desired release that you need. We will use the dunfell release in this example.

```
git clone git://git.yoctoproject.org/poky

cd poky

git checkout -b dunfell origin/dunfell
```

CREATE YOUR ACTIVE BUILD DIRECTORY

A build directory is where all your Yocto builds reside. What this allows you to do is to compartmentalise the builds cleanly but at the same time re-use as many resources as possible. In the build directory resides the machine configuration, specific variables that your image/machine needs, additional recipes you need added to your image, etc.

At the same time, the reusable components like common recipes, source code, tarballs, build state cache, etc. all reside in the poky folder. This allows you to simply create a new build directory for your other machines or projects but still reuse existing components already downloaded, created and/or built.

To create a build directory, you simply need to source a script called oe-init-build-env that will create the build environment (paths, shortcuts, variables, etc.).2

```
cd ..

source poky/oe-init-build-env <name of your build directory>
```

If you don't specify a name, a build directory named build is automatically created. The next time you want to use the same directory, simply call the same command again. Also, once this command has executed, you will already be inside the newly created build directory.

START YOUR FIRST BUILD

The very first build is generally the longest as it downloads hundreds of tarballs from various upstream sources mentioned in the recipes. So, it is a good idea to break it down into two parts. First step is to just fetch all the source code, tarballs, etc. Second step is to do the actual build using the downloaded resources along with the configuration metadata and recipes.

Speaking of recipes, there are a number of core- image recipes already available for you to use. These core recipes enable you to create a working Linux image without doing absolutely any customisation for your platform. A successful creation of such an image is proof that your build setup works and you can then move on to doing more sophisticated builds for your own machines.

A small subset of these core images would already be on your screen if your last command executed correctly i.e., you are already inside your build directory. Some popular core images are:

- core-image-base – A console-only image that fully supports the target hardware.
- core-image-minimal – A small image allowing the device to just boot.
- core-image-sato – An image with Sato support, a mobile environment and visual style that works with mobile devices.
- core-image-clutter – An image with support for the OpenGL-based toolkit Clutter
- core-image-full-cmdline – A console-only image with more full-featured Linux system functionality installed.
- …

The default machine selected for Yocto build is named qemux86-64. This selection can be viewed in the file conf/local.conf. QEMU is a very handy tool to test your Yocto images as it allows you to see if all the feature that you needed in the image are there without really downloading the image to your target machine. As a bonus, it is also a pretty cool Linux learning tool as the QEMU runs the image using virtualisation and has access to the same resources as your host machine (say Ubuntu).

In this section, we use core-image-base and keep qemux86-64 as the machine.

To fetch the resources needed for this build, execute the below command.

```
bitbake core-image-base --runall=fetch
```

Depending on your internet connection speed and the CPU capability itself, it may take anywhere from few tens of minutes to an hour or more.

Once the resources have been fetched, it is time to start your build. To do so, execute the below command.

```
bitbake core-image-base
```

Depending on your CPU capability, it may take anywhere from few tens of minutes to maybe even a few hours (took some hours for us!).

RUNNING THE IMAGE

Once the build is complete, the output image can be found in `tmp/deploy/images/qemux86-64` folder. Poky provides a handy tool called `runqemu` that abstracts out a lot of gory steps needed to run the image. It all comes down to executing a simple command to boot your very first Yocto image for the very first time!

To do so, execute the below command:

```
runqemu qemux86-64 nographic
```

The `nographic` argument tells `runqemu` that we are not interested in launching a GUI. This is handy as the graphics card access is sometimes finicky which can lead to crashes.

That's it – you should now see your `qemux86-64` image boot up. The first boot may take a long time…maybe up to a minute or slightly more. Patience is the key!

You have just built your first ever Yocto image. In the next lab, we will build our first image for our Raspberry Pi 3 Model B+ single-board computer system.

BUILDING A BASIC IMAGE FOR RASPBERRY PI

GOAL

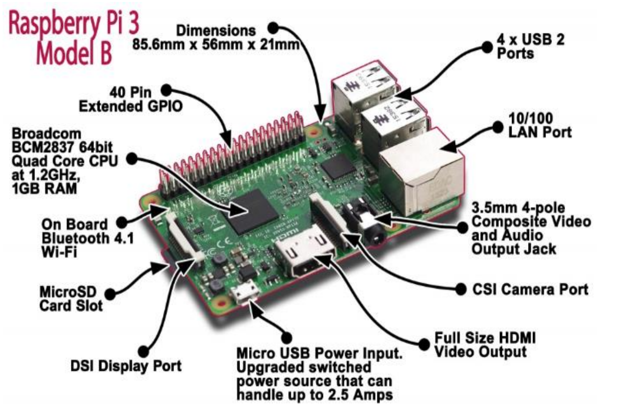
In the previous lab, we talked about Yocto basics and setup of a host followed by a basic image build. In this lab, we will see how to take our past work done forward by building a test image for the Raspberry Pi. If you are not aware of the Raspberry Pi, check out this YouTube playlist that gives a crash course into the Raspberry Pi ecosystem and basic usage.

This section assumes that a build directory has already been created. If not, please head to the previous lab and perform the steps detailed there but stop after creating your build directory. Let's start building an image for the Raspberry Pi 3.

THE RASPBERRY PI 3

The Raspberry Pi 3 is the successor to the Raspberry Pi 2. It builds upon the Pi 2 by upgrading the Arm cores to Cortex-A53 and adding an onboard single-band 2.4GHz-only wireless chipset.

The Raspberry Pi 3 measures the same 85.60mm x 53.98mm x 17mm, with a little overlap for the SD card and connectors that project over the edges. The SoC is a Broadcom BCM2837. This contains a quad-core Cortex-A53 running at 1.2GHz and a VideoCore 4 GPU.

Raspberry Pi 3 Model B
Dimensions 85.6mm x 56mm x 21mm
4 x USB 2 Ports
40 Pin Extended GPIO
10/100 LAN Port
Broadcom BCM2837 64bit Quad Core CPU at 1.2GHz, 1GB RAM
3.5mm 4-pole Composite Video and Audio Output Jack
On Board Bluetooth 4.1 Wi-Fi
CSI Camera Port
MicroSD Card Slot
Full Size HDMI Video Output
DSI Display Port
Micro USB Power Input. Upgraded switched power source that can handle up to 2.5 Amps

SET UP THE BUILD ENVIRONMENT

Let us assume that the name of your build directory is build-rpi. Set it up by executing the following command in your yocto directory.

```
source poky/oe-init-build-env build-rpi
```

NOTE: Replace build-rpi with any name you like for your build directory.

BITBAKE LAYER CONFIGURATION FOR RASPBERRY PI

When we built the image for QEMU, we did not have to do any layer configuration as the poky distribution itself ships with the necessary recipes for the particular QEMU machine. For instance, our machine in the last lab was qemux86-64.

From the readme.md document of the meta-raspberrypi layer (the layer we need for building images for Raspberry PI line of embedded systems), we need at least meta-openembedded layer and probably some more depending on the features needed in the image. The recipes within meta-openembedded enable builds for embedded architectures like ARM, MIPS, etc.

For our build, we will also add other layers necessary for networking, python, multimedia, etc. We will first download these and then add to our build configuration.

DOWNLOADING NECESSARY LAYERS

Let us now download the layers. It is considered a good practice to have the meta- layers in the same directory as yocto so that multiple build directories can use the same. For our system, the folder structure looks something like this:

- yocto
    - poky
        - …
    - << new layers that we will download now >>

We will first cd into the yocto directory.

```
cd ..
pwd
    <your yocto directory>/yocto
```

Let us now download the Open-Embedded source code and check out the release we want. As we saw in our last lab, we are using the dunfell release.

```
git clone git://git.openembedded.org/meta-openembedded -b dunfell
```

Let us now download the meta-raspberrypi layer source code. This layer contains all the recipes that are necessary to build basic images for the Raspberry Pi series of boards. The layering concept we discussed in the previous lab is seen in action here! The meta-raspberrypi layer will contain the appropriate recipes for the kernel, drivers, configuration, etc. that will override those in the corresponding poky and meta-openembedded recipes. For example: the Linux kernel configuration.

```
git clone git://git.yoctoproject.org/meta-raspberrypi -b dunfell
```

We have downloaded the necessary layers. Our folder structure now looks something like this:

- yocto
    - poky
        - …
    - meta-raspberrypi

    - …
- meta-openembedded
    - meta-oe
    - meta-networking
    - meta-python
    - meta-multimedia
    - …

ADDING LAYER TO BUILD USING BITBAKE COMMANDS

You can manually modify the bitbake layer configuration by editing the conf/bblayers.conf file in your active build directory. However, such an approach is prone to syntax errors which may lead to failed builds due to parser failure.

The best approach is to use the handy bitbake-layers command. To add the necessary layers to the build, execute the below from your build-rpi directory.

```
bitbake-layers add-layer ../meta-raspberrypi
bitbake-layers add-layer ../meta-openembedded/meta-oe
bitbake-layers add-layer ../meta-openembedded/meta-python
bitbake-layers add-layer ../meta-openembedded/meta-networking
bitbake-layers add-layer ../meta-openembedded/meta-multimedia
```

You can now check that your layer configuration is updated by executing the below.

```
bitbake-layers show-layers
```

CHOOSE THE MACHINE AND THE IMAGE

Image recipes are .bb files that contain complete details of how the image for that particular machine is to be built. Machine names can be found in the respective meta- layers.

CHOOSE THE MACHINE

The machine names supported by the meta-raspberrypi layer can be seen in the meta-raspberrypi/conf/machine directory. Each .conf file is the machine configuration – the name of the machine is the name of the file (without the .conf file extension). You can list the machines supported in meta-raspberrypi by executing the below command from the yocto directory.

```
ls meta-raspberrypi/conf/machine
```

As we want to build an image for Raspberry Pi 3, we will use the machine name as raspberrypi3.

CHOOSE THE IMAGE

To see the images that you can build, see the contents of the meta-raspberrypi/recipes-core/images folder. To do so, execute the below command from the yocto directory.

```
ls meta-raspberrypi/recipes-core/images
```

You should see rpi-basic-image.bb, rpi-hwup-image.bb and rpi-test-image.bb. The images names are the names of these files without the .bb file extension.

In this section, we will build the rpi-test-image.

SET THE MACHINE VARIABLE

By default, the build target machine is chosen to be qemux86-64. This needs to be changed to raspberrypi3. We will do this in the conf/local.conf file located in our build directory. We will first navigate to the active build directory and then make the necessary modification.

```
cd build-rpi

nano conf/local.conf
```

NOTE: Replace build-rpi with the name of your build directory if using a different name.

NOTE: You can use any other editor if nano is not available or your preference.

Navigate to the presently uncommented line which sets the machine variable, comment it out by putting a # in front of it and the set raspberrypi3 as the machine. Below is how the final result should look like.

```
# MACHINE ??= "qemux86-64"

MACHINE ??= "raspberrypi3"
```

PRO-TIP: To test this Raspberry Pi 3 image, it is useful to have the serial console enabled. To do so, add the below line towards the end of the conf/local.conf file.

```
ENABLE_UART = "1"
```

The serial console on the Raspberry Pi 3 boards is available on pin numbers **8 (GPIO14 – TXD)** and **10 (GPIO15 – RXD)** of the 40-pin IO connector.

We are now ready to do our build. We will follow the same approach we followed last time i.e., first only fetch the resources needed for the build and then perform the build itself.

FETCH AND BUILD!

You can now execute the below command for only fetching the necessary resources without starting the build.

```
bitbake rpi-test-image --runall=fetch
```

Depending on your internet connection and the CPU capability, it can take anywhere between few minutes to few tens of minutes to complete this process.

Once done, we can start the build by executing the below command.

```
bitbake rpi-test-image
```

Depending on your internet connection and the CPU capability, it can take anywhere between a few tens of minutes to maybe a few hours to do this build. Patience is the key!

You'll find the completed image in your build-rpi directory under tmp/deploy/images/raspberrypi3/rpi-test-image-raspberrypi3.wic.bz2.

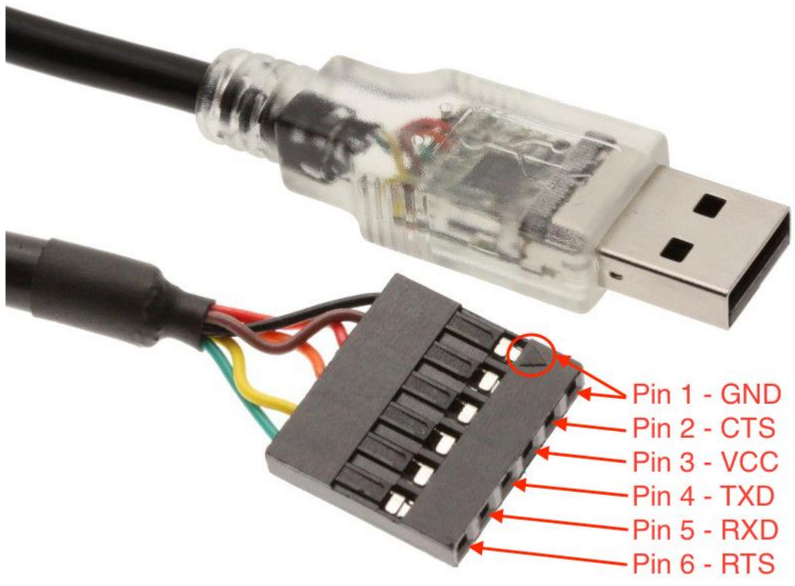
This file is compressed. To decompress it, use the bzip2 command or a tool like 7zip.

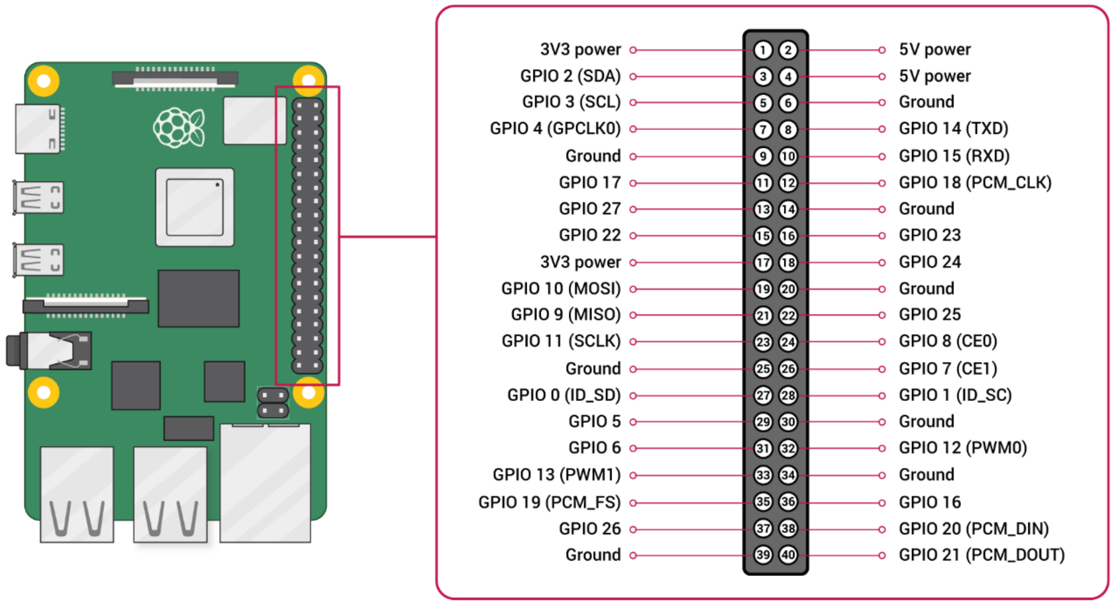
```
bzip2 -d -f tmp/deploy/images/raspberrypi3/rpi-test-image-raspberrypi3.wic.bz2
```

That's it – if all went well, you have built a test image (the extracted file called rpi-test-image-raspberrypi3.wic) for Raspberry Pi 3 using Yocto!

BOARD SETUP

Once the image (i.e., rpi-test-image-raspberrypi3.wic) is flashed, insert the SD card into your Pi and connect the ground, Tx and Rx pins of the USB to Serial cable (shown below) to the ground (Pin6), Rx (Pin 10) and Tx (Pin 8) pins of your Raspberry Pi 3 (also shown below).





Connect the USB end of the USB to Serial cable to your computer and open the serial port using a serial terminal software such as GNU Screen with baud rate set to 115200. Power up the system and you should see the custom welcome message you set in the previous step, enter the username as “root” and no password is required. Congratulations, you just logged into your own custom-built Linux.

SUMMARY

Yocto is a powerful tool and there is a lot more to learn. If you need an additional piece of software in your Image, the first place to start is on the OpenEmbedded Layer Index. There you can search for meta layers containing a recipe for the software. Download the layer and add it to your bblayers.conf, if not already there. Then, add the name of the recipe in your local.conf file as shown below as an example for the recipe called nano existing within the meta-oe layer.

```
IMAGE_INSTALL_append += "nano"
```

This will add the command line text editor “nano” to your image.

Rebuild your Image, flash it, and you’re ready to go.

CREATING & ADDING A NEW LAYER TO YOUR IMAGE

GOAL

In the previous sections about Yocto, we talked about a variety of things like what Yocto is, how you download it, build base images and so on.

In this section, we cover a rather basic but very important topic that is often the starting point for you to customize your image. We talk about creating and adding a new layer to your image.

LAYERS – A REFRESHER

In the introduction, we talked about the layered nature of the Yocto build system and how it enables maximum re-use of software as well as allows you to easily migrate from one hardware platform to the other. The idea of layers is that you can easily bundle your *secret sauce* (read "your own source code") in your own layer as well as over-ride some configurations for the same recipe sitting in a layer below yours. For example – you could modify the packages that get installed in a base image that already exists in meta-oe by adding your software to it or customize the Linux kernel configuration for a particular image and so on.

We will now learn how to create our own layer.

CREATING A NEW LAYER

The easiest and the most recommended method to create a new layer is using the bitbake-layers script. This script is already available for you once you have sourced the oe-build-init-env script inside yocto/.

The bitbake-layers script is pretty versatile and does a lot more than just creating the layer. To see what it can do, we will first init our build environment and then look at the help documentation available to us. These commands have to be run while inside yocto/ folder.

```
source poky/oe-init-build-env build-rpi

bitbake-layers --help
```

You should see output as below.

```
NOTE: Starting bitbake server...
usage: bitbake-layers [-d] [-q] [-F] [--color COLOR] [-h] <subcommand> ...

BitBake layers utility

optional arguments:
  -d, --debug           Enable debug output
  -q, --quiet           Print only errors
  -F, --force           Force add without recipe parse verification
  --color COLOR         Colorize output (where COLOR is auto, always, never)
  -h, --help            show this help message and exit

subcommands:
  <subcommand>
    add-layer           Add one or more layers to bblayers.conf.
    remove-layer        Remove one or more layers from bblayers.conf.
    flatten             flatten layer configuration into a separate output directory.
    show-layers         show current configured layers.
    show-overlayed      list overlayed recipes (where the same recipe exists in another
layer)
    show-recipes        list available recipes, showing the layer they are provided by
    show-appends        list bbappend files and recipe files they apply to
    show-cross-depends  Show dependencies between recipes that cross layer boundaries.
    layerindex-fetch    Fetches a layer from a layer index along with its dependent
layers, and adds them to conf/bblayers.conf.
    layerindex-show-depends
                        Find layer dependencies from layer index.
    create-layer        Create a basic layer

Use bitbake-layers <subcommand> --help to get help on a specific command
```

As you can see, this script can help us not just to create a layer but also to add a layer to our configuration, remove a layer from the build configuration, show the currently added layers and also a few more useful operations like showing us the recipes that are a part of the build or from a particular layer and so on.

Let us check what layers are currently a part of our configuration.

```
bitbake-layers show-layers
```

This should print out an output something like below.

NOTE: Your output may be different depending on the builds you have done in the past. This is just an indicative output.

```
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
meta                  /home/skasap/yocto/poky/meta              5
meta-poky             /home/skasap/yocto/poky/meta-poky         5
meta-yocto-bsp        /home/skasap/yocto/poky/meta-yocto-bsp    5

```

STEP 1 – CHOOSE THE LOCATION FOR YOUR LAYER

You already know that our folder path looks something like below. However, if yours is different, just make the necessary changes in the below commands so that your paths are consistent.

- yocto/
    - meta-openembedded
    - meta-raspberrypi
    - poky/
        - bitbake
        - build
        - meta
        - ….

It is generally to good idea to have your own layer placed inside the yocto/ folder such that is it along other non-poky layers that you have added. Also, it is considered good practice to have the layer name start with meta- although it is not mandatory.

We will now create a layer named meta-ke inside yocto/ folder.

STEP 2 – CREATE THE LAYER

Suppose we are currently in the build-rpi/ directory. We will execute the below commands.

```
bitbake-layers create-layer ../meta-ke
```

The folder structure should now look like this.

- yocto/
    - meta-openembedded
    - meta-raspberrypi
    - meta-ke
    - poky/
        - bitbake
        - build
        - meta
        - ….

ADDING THE LAYER TO BUILD CONFIGURATION

Now that we have successfully created our new layer, let us now add it to our configuration. In the future, we shall add our own recipes to this layer and create our own customized images. More on this later.

We shall use the bitbake-layers script again for adding this layer to our build configuration.

```
bitbake-layers add-layer ../meta-ke
```

To confirm that this addition was successful, we can run the bitbake-layers script with the show-layers argument. We should see our layer added to the output.

```
layer                 path                                      priority
==========================================================================
meta                  /home/skasap/yocto/poky/meta              5
meta-poky             /home/skasap/yocto/poky/meta-poky         5
meta-yocto-bsp        /home/skasap/yocto/poky/meta-yocto-bsp    5
meta-ke               /home/skasap/yocto/meta-ke                6
```

TRY BUILDING THE RECIPE INSIDE NEW LAYER

When bitbake-layers created the layer for us, it also created an example recipe for us named example_0.1.bb – this is located at meta-ke/recipes-example/example/.

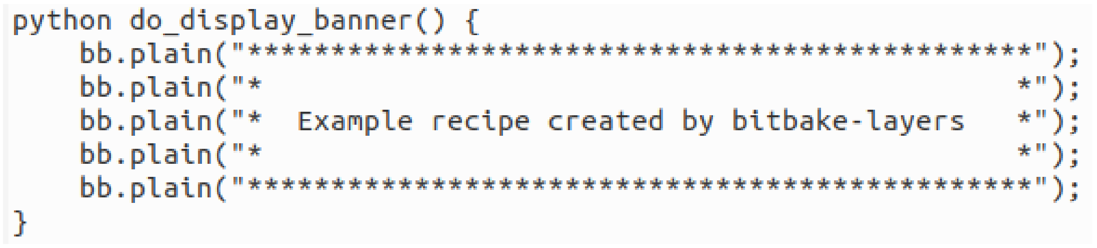
Let us now try to bake this recipe. If all goes well, we should see build stage prints. A successful bake tells us that the layer operations went well and we can now use this layer for all our image builds in the future!

```
bitbake example
```

You should see output like below during the build process. If you had already done an image build in the past, this should be relatively quick.

```
NOTE: Executing Tasks
***********************************************
*                                             *
*  Example recipe created by bitbake-layers   *
*                                             *
***********************************************
```

This emanates from the do_display_banner() function inside example.bb.



```
python do_display_banner() {
    bb.plain("***********************************************");
    bb.plain("*                                             *");
    bb.plain("*  Example recipe created by bitbake-layers   *");
    bb.plain("*                                             *");
    bb.plain("***********************************************");
}
```

HOW DID BITBAKE KNOW WHAT TO DO?

How does bitbake know what example is?

Why did you not need to specify which layer bitbake should look for to locate the recipe named example ?

What else does creating the new layer do for us?

Let us understand this in the next section!

UNDERSTANDING AND CREATING YOUR FIRST CUSTOM RECIPE

GOAL

In the last section, we talked about how creating a custom Yocto layer is the first step towards creating a custom image for the machine of your interest. This section will build on the work done in the previous one (rather loosely!).

In this section, we talk about creating your first ever recipe! However, before we do that, we will discuss some of the basics of Yocto recipes – how to name them, how they are found by `bitbake`, how to write one yourself for a very basic use-case and finally end with learning what people do with recipes!

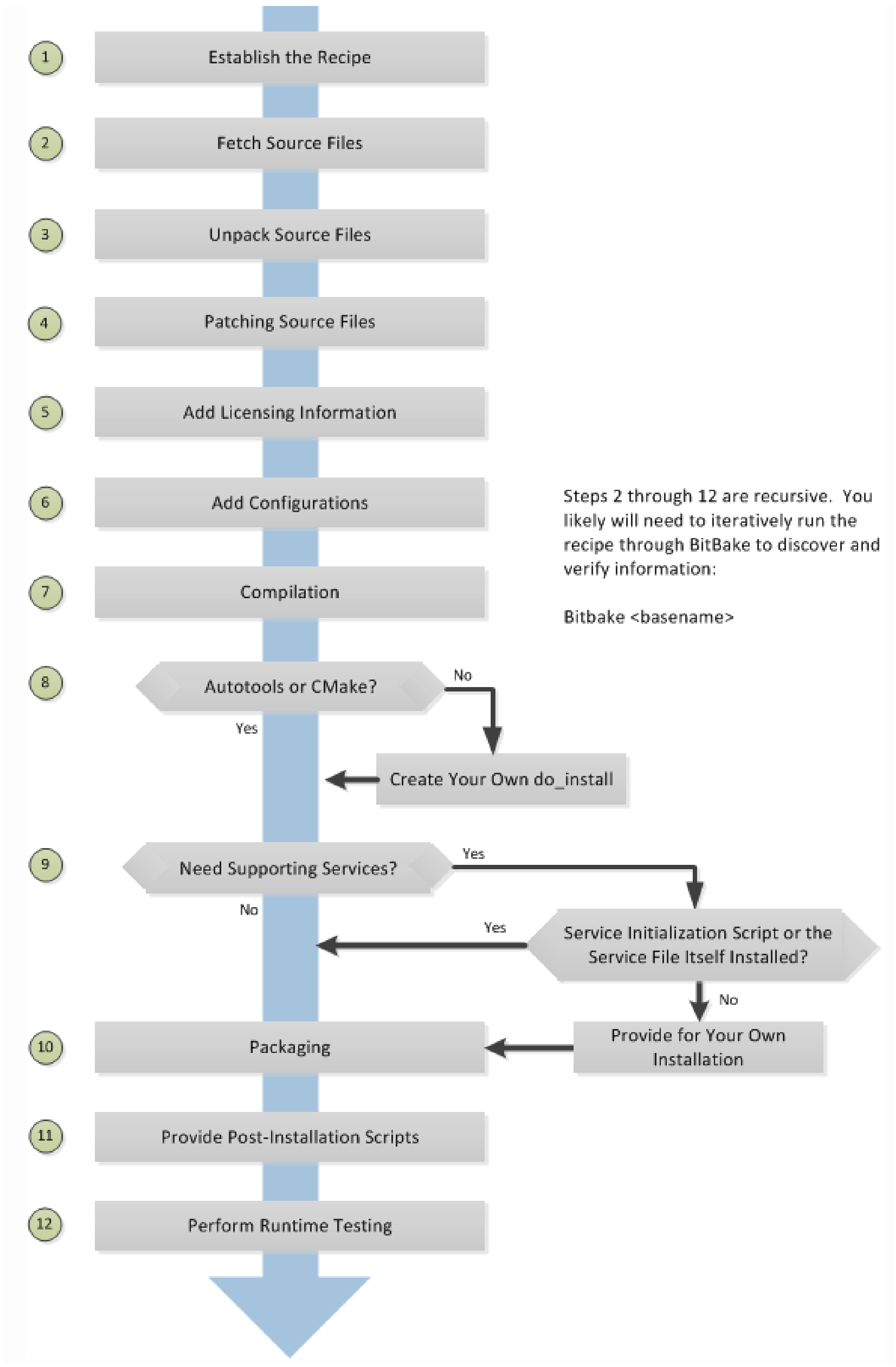
RECIPES – AN INTRODUCTION

In the previous lab, we saw how Yocto is an extremely scalable build system which is used ubiquitously to create Linux distributions for a variety of machines around us. It employs a tool called BitBake which is basically a scheduler-cum-execution engine that parses metadata passed to it to and performs actions as specified by the metadata. It can be said that `bitbake` is the workhorse of the entire build process.

A bitbake recipe or simply a recipe is a form of metadata that contains information about how bitbake can obtain the source code, configure the source code, patch it, build it, install it, and basically everything that bitbake should be doing with the individual software component for whom the recipe is written.

Basically, a bitbake recipe is analogous to a recipe for a food item in the real life – the ingredients, the process, the presentation, the serving, and so on till you are ready to gobble it up!

The official Yocto documentation specifies the process of creating a bitbake recipe as shown in the below image. It is important to understand that not all recipes are created using the exact same process below. Some steps can be skipped if they are not needed. For example: some recipes may not require any patches to be applied or may not need to be compiled at all!


1
Establish the Recipe
2
Fetch Source Files
3
Unpack Source Files
4
Patching Source Files
5
Add Licensing Information
6
Add Configurations
7
Compilation
8
Autotools or CMake?
No
Yes
Create Your Own do_install
9
Need Supporting Services?
Yes
No
Yes
Service Initialization Script or the Service File Itself Installed?
No
10
Packaging
Provide for Your Own Installation
11
Provide Post-Installation Scripts
12
Perform Runtime Testing

Steps 2 through 12 are recursive. You likely will need to iteratively run the recipe through BitBake to discover and verify information:

Bitbake <basename>

HOW TO NAME A BITBAKE RECIPE

The nomenclature of a bitbake recipe is very simple. It looks something like this.

```
basename_version.bb
```

As the name suggests, the basename is the fundamental name of the recipe. You are not allowed to use reserved suffixes or prefixes like cross, native, lib, etc. in this name. Of course, if your basename itself makes use of these words, it should be fine. The idea is to not tie down the recipe to an architecture or a type as this can be done by Yocto.

The version can be a dot-separated string. The convention is to use a major.minor or a major.minor.patch notation. For example, the below are acceptable.

```
somerecipe_1.2.bb

anotherrecipe_1.2.1.bb
```

WHERE TO PLACE A BITBAKE RECIPE

BitBake recipes are always located within a Yocto layer. For the recipe to be visible to bitbake, that particular Yocto layer should be added to the current build configuration as explained in the previous section.

Suppose you have added the layer containing your recipe to the build configuration – how does bitbake find it?

To answer this, let us look at the conf/layer.conf file in the layer we created in the previous section.

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

.....
.....
```

The most relevant configurations are BBPATH and BBFILES.

In the auto-generated layer.conf,

- The BBPATH variable (similar to PATH variables in our system) is used by BitBake to locate .bbclass and configuration files.
- The BBFILES variable is a space-separated list of recipe files BitBake uses to build software. Most importantly, it is possible to use wild cards like * in the file above that allows you to specify a generic path. The syntax is similar to the glob functionality in Python.

Based on the above info, it is clear to us that as long as your recipe is located in a folder inside the layer where the folder name is something like recipes- , your recipe will be found by bitbake.

A SKELETAL BITBAKE RECIPE

The below can be used as the starting point when starting your own custom recipe. This skeleton is directly taken from Yocto's manual. It is not necessary to have all of these mandatorily – this is just a recommendation.

```
DESCRIPTION = "This is a short description of the recipe"
HOMEPAGE = "You can specify your home page "
LICENSE = "Type of license. Eg: GPLv2, MIT, etc."
SECTION = "Useful if package management is required - can omit"
DEPENDS = "Specify the recipes on which your recipe depends"
LIC_FILES_CHKSUM = "md5sum or sha256sum of the license"

SRC_URI = "Where to get the source code from"
```

Save this skeleton recipe for future reference!

WRITING A RECIPE FOR A SIMPLE HELLO WORLD PRINTER APPLICATION

When we created our custom layer in the last section, the layer directory looked something like this.

- meta-ke
    - conf/
    - COPYING.MIT
    - README
    - recipes-example/

Let us create a new folder named recipes-ke to hold our custom recipe that we want to create. This is in accordance with the requirements of the BBFILES setting in conf/layer.conf. The folder structure should now look like this.

- meta-ke
    - conf/
    - COPYING.MIT
    - README
    - recipes-example/
    - recipes-ke/

Let us quickly write a hello world printer C code and name this file hello-world-local.c.

```
#include <stdio.h>

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

If a recipe uses a locally available source tree, then it is supposed to be located in a folder named files which is located in the same folder as the recipe, i.e., *.bb file. To keep things clean, let us create a folder inside recipes-ke called hwlocal and add the files folder to it along with the source code. Also, let us create a recipe named hwlocal_0.1.bb inside hwlocal.

Now, the meta-ke folder looks like this.

- meta-ke
    - conf/
    - COPYING.MIT
    - README
    - recipes-example/
    - recipes-ke/
        - hwlocal/
            - files/
                - hello-world-local.c
            - hwlocal_0.1.bb

Using the skeletal recipe as a base, let us write hwlocal_0.1.bb and then try to understand what we did and why we did it.

```
DESCRIPTION = "This is a simple Hello World recipe - uses a local source file"
HOMEPAGE = "https://www.coventry.ac.uk"
LICENSE = "MIT"
LIC_FILES_CHKSUM                                                                     =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://hello-world-local.c"

S = "${WORKDIR}"

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

do_install() {
    install -d ${D}${bindir}
    install -m 0755 hello-world-local ${D}${bindir}
}
```

Let us now try to bake this new recipe. If all goes well, we should see build stage prints; a successful bake tells us that the layer operations went well. We can now use this layer for all our image builds in the future!

```
bitbake hwlocal
```

Let us understand now what is happening here piece-by-piece.

DESCRIPTION

This is a simple description of what is happening in this recipe.

HOMEPAGE

Generally, this points to the web address you want others to refer to know more about the recipe, the author or who owns it.

LICENSE

This is a very important information and critical for a build to succeed. It tells bitbake about the kind of license associated with the source code that will be used in the recipe. Poky ships with a large number of licenses that are most commonly used in the industry. You can view these license details at this path in your poky installation: poky/meta/files/common-licenses.

LIC_FILES_CHKSUM

This information is equally critical for a build to succeed. It informs bitbake about where the license file can be found and what its checksum is. If the checksum calculated by bitbake does not match the checksum mentioned here, the build will fail – it happens during the QA stage.

We just saw a number of license types. You can choose to copy one of these licenses into your source tree and then refer to it or you can directly refer to a license present in your poky installation using the bitbake environment variable called COMMON_LICENSE_DIR.

We have chosen the latter approach in our recipe.

SRC_URI

This tells bitbake where to find the source code. To keep things simple, we have pointed bitbake to a locally available file called hello-world-local.c. By default, BitBake will look at the files folder for this file. Notice how we have referred to file path as file:// to indicate that it is a local resource.

Let us now move on to some critical configurations.

S

This is the location in the build directory where unpacked recipe source code resides. When dealing with tarballs, there is generally no need to specify this explicitly.

But since we are dealing with a local source tree, it is imperative to tell bitbake where this source code will reside while the BitBake operations like patching, compilation, configuration, etc. are happening. We specify this as the environment variable called WORKDIR which is the active working directory for BitBake. More about WORKDIR and other Yocto variables here.

DO_COMPILE()

Here, we tell bitbake what command should be executed to compile the source tree. Ours is a relatively simple one-line C command but this can often become complex as the source tree grows in size.

Note how the compiler name and flags are passed as CC and LDFLAGS – these are environment variables populated correctly by bitbake.

DO_INSTALL()

Here, we tell bitbake where we want to install the resulting binary output that we got after compilation. Notice how the output of compilation is a binary named hello-world-local.

We here tell bitbake to install the binary into the /bin folder of the target machine's image that will be generated by bitbake. The environment variable named bindir expands as /bin and the environment variable named D points to the path of the target machine's image.

To know all the environment variables for your build configuration, you can just run the below command once the oe-init-build-env script is sourced.

```
bitbake -e
```

Study the output of the above command and see for yourself the values of WORKDIR, S, D, COMMON_LICENSE_DIR, AVAILABLE_LICENSES, MACHINE, etc.

CREATING IMAGE WITH THE CUSTOM RECIPE

Before you can re-build an image including the packages from your recipe, you need to add the name of your recipe in your local.conf file as shown below.

```
IMAGE_INSTALL_append += "hwlocal"
```

Then, you can re-build your image which will include hello-world-local executable file in its /usr/bin directory.

```
bitbake rpi-test-image
```

At this take, decompress the build image, flash it to the sdcard, make the necessary serial cable connections, and then type hello-world-local at the serial console to see your output message when the boot process is completed!

SUMMARY

This lab has covered the basics of creating a custom recipe and helped you write your first ever bitbake recipe. But often, the real world has much more complex requirements – the source tree is either located in a remote git repository or a tarball. Also, the source tree may specify a way to build it (e.g., CMake, autotools, etc.).

Fortunately, bitbake supports such use-cases and what's more – it also allows you to easily switch versions of your recipe with minimal changes. However, these are advanced concepts beyond the scope of this module.