PART – 1

**Four elements of embedded Linux:**

**Toolchain**: The compiler and other tools needed to create code for your target device. Everything else depends on the toolchain.

**Bootloader**: The program that initializes the board and loads the Linux kernel.

**Kernel:** This is the heart of the system, managing system resources and interfacing with hardware.

**Root Filesystem**: Contains libraries and programs that are run once the kernel has completed its initialization.

One more element can be collection of programs specific to your embedded application which make the device do what ever it is supposed to do, be it weigh groceries, display movies, control a robot, or fly a drone.

**What is Yocto Project:**

Yocto project provides open source, high quality infrastructure and tools to help developers create their own custom Linux distributions for any hardware architecture.

To understand the outcome provided by the Yocto Project, we can use the analogy of the computing machine.

Input: Set of data that describes what we want, that is our specification

(Kernel Configuration, Hardware Name, Packages/Binaries to be installed)

Output: Linux Based Embedded Product

(Linux Kernel, Root File System, Bootloader, Device Tree)

**Advantages of Yocto Project:**

1. Widely adopted across the industry

Semiconductor, operating system, software, and service vendors exist whose products and services adopt and support the Yocto project.

1. Architecture Agnostic

Supports Intel, ARM, MIPS, AMD, PPC and other architectures

Chip vendors create and supply BSPs that support their hardware

If you have custom silicon, you can create a BSP that supports that architecture

The Yocto project fully supports wide range of device emulation through the Quick Emulator (QEMU)

1. Images and Code Transfer Easily

Yocto project output can easily move between architectures without moving to new development environments.

1. Flexibility

Through customization and layering, a project group can leverage the base Linux distribution to create a distribution that works for their product needs.

1. Ideal for constrained Embedded and IoT devices

Unlike a full Linuc distribution, you can use the Yocto Project to create exactly what you need for embedded devices

You only add the feature support or packages that you absolutely need for the device

1. Uses a Layer Model

The Yocto Project layer infrastructure groups related functionality into separate bundles.

You can incrementally add these grouped functionalities to your project as needed

Allows you to easily extend the system, make customizations, and keep functionality organized.

**Poky:**

Poky is a reference distribution of Yocto Project. The word “reference” is used to mean “example” in this context. Yocto Project uses Poky to build images (kernel, system, and application software) for targeted hardware. At the technical level it is a combined of the components

• Bitbake

• OpenEmbedded Core

• meta-yocto-bsp

• Documentation

Note: Poky does not contain binary files, it is a working example of how to build your own custom Linux distribution from source.

**What is difference between Poky and Yocto?**

The exact difference between Yocto and Poky is Yocto refers to the organization (like one would refer to ‘Cannonical’, the company behind Ubuntu), and Poky refers to the actual bits downloaded ( analogous to ‘Ubuntu’)

**Metadata:**

Non Yocto: A set of data that describes and gives information about other data.

Yocto World:

* Metadata refers to the build instructions
* Commands and data used to indicate what versions of the software are used
* Where they are obtained from
* Change or additions to the software itself (patches) which are used to fix bugs or customize the software for use in a particular situation.

Poky = Bitbake + Metadata

Metadata is a collection of:

* Configuration files (.conf)
* Recipes (.bb and .bbappend)
* Classes (.bbclass)
* Includes (.inc)

**OpenEmbedded Project:**

OpenEmbedded offers a best-in-class cross-compile environment. It allows developers to create a complete Linux Distribution for embedded systems.

**What is the difference between OpenEmbedded and the Yocto Project?**

The Yocto Project and OpenEmbedded share a core collection of metadata called

**OpenEmbedded-Core**.

However, the two organizations remain separate, each with its own focus

OpenEmbedded provides a comprehensive set of metadata for a wide variety of architectures, features, and applications

Not a reference distribution

Designed to be the foundation for others

The Yocto Project focuses on providing powerful, easy-to-use, interoperable, well-tested, metadata, and board support packages (BSPs) for a core set of architectures and specific boards.

**OpenEmbedded-Core:**

The Yocto Project and OpenEmbedded have agreed to work together and share a common core set of metadata (recipes, classes and associated files): **oe-core**

**Bitbake:**

Bitbake is a core component of the Yocto Project.

It basically performs the same functionality as of make.

It’s a task scheduler that parses python and shell script mixed code

The code parsed generates and runs tasks, which are basically a set of steps ordered according to code’s dependencies.

It reads recipes and follows them by fetching packages, building them and incorporating the results into bootable images.

It keeps track of all tasks being processed in order to ensure completion, maximizing the use of processing resources to reduce build time and being predictable.

**meta-yocto-bsp:**

A Board Support Package (BSP) is a collection of information that defines how to support a particular hardware device, set of devices, or hardware platform.

The BSP includes information about the hardware features present on the device and kernel configuration information along with any additional hardware drivers required.

The BSP also lists any additional software components required in addition to a generic Linux software stack for both essential and optional platform features.

The meta-yocto-bsp layer in Poky maintains several BSPs such as the Beaglebone, EdgeRouter, and generic versions of both 32-bit and 64-bit IA machines.

Machines Supported:

Texas Instruments BeagleBone (beaglebone)

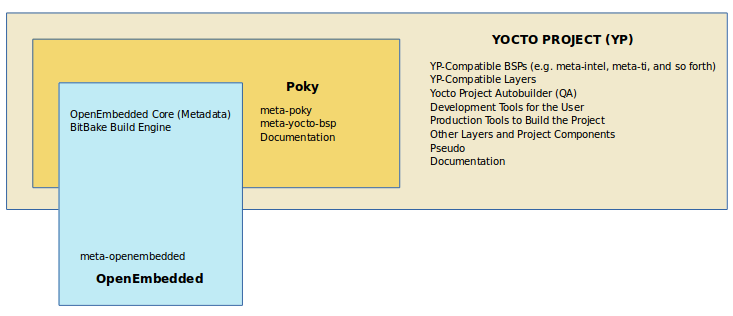
Intel x86-based PCs and devices (genericx86 and genericx86-64)

Ubiquiti Networks EdgeRouter Lite (edgerouter)

Note: To develop on different hardware, you will need to complement Poky with hardware-specific Yocto layers.

**Poky includes:**

* Some OE components (oe-core)
* Bitbake
* Demo-BSPs
* Helper scripts to setup environment
* Emulator QEMU to test the image



#### WORKFLOW OF YOCTO PROJECT

Step 1: Download the Poky Source Code

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

Step 2: Checkout to latest branch/release

$ git checkout <branch-name>

Step 3: Prepare the build environment

Poky provides you a script ‘oe-init-build-env’, which should be used to setup the build environment

Script will set up your environment to use Yocto build system, including adding Bitbake uitility to your path

$ source poky/oe-init-buid-env [ build\_directory ]

Here build\_directory is optional argument for the name of the directory where the environment is set in case it is not given, it defaults to “build”

The above script will move you in a build folder and create two files ( local.conf, bblayers.conf ) inside conf folder

Step 4: Building Linux Distribution

$ bitbake <image\_name>

$ time bitbake core-image-minimal

## core-image-minimal

This is a small image allowing a device to boot, and it is very useful for kernel and boot loader tests and development

##### COMMAND TO RUN THE GENERATED IMAGE IN QEMU

QEMU is a free and open source software package that performs hardware virtualization.

The QEMU based machines allow test and development without real hardware.

Currently emulations are supported for:

* + ARM
  + MIPS
  + MIPS64
  + PowerPC
  + x86
  + x86\_64

Poky provides a script ‘runqemu’ which will allow you to start the QEMU using yocto generated images.

The runqemu script is run as:

runqemu <machine> <zimage> <filesystem>

where:

<machine> is the machine/architecture to use (qemuarm/qemumips/qemuppc/qemu/qemux86/qemux86-64)

<zimage> is the path to a kernel (e.g zimage-qemuarm.bin)

<filesystem> is the path to an ext2 image (e.g. filesystem-qemuarm.ext2) or an nfs directory

Full usage instructions can be seen by running the command with no options specified.

# Exit QEMU

Exit QEMU by either clicking on the shutdown icon or by typing ctrl-c in the qemu transcript window from which you evoked QEMU.

##### STEPS TO GENERATE ARM IMAGE AND RUN IN QEMU

When you set up the build environment, a local configuration file named local.conf becomes available in a conf subdirectory of the Build directory.

The defaults are set to build for a qemux86-64 target

Edit ./build/conf/local.conf

Set

MACHINE = “qemuarm”

$ source poky/oe-init-build-env

$ bitbake core-image-minimal

$ runqemu core-image-minimal

### Nographic

You can launch QEMU without the graphic window by adding nographic to the command line

$ runqemu qemuarm nographic

**To add a particular package in your root file system:**

Open your local.conf file and add the name below

IMAGE\_INSTALL += “recipe-name”

E.g IMAGE\_INSTALL += “usbutils” for lsusb

Or IMAGE\_INSTALL\_append = “ usbutils” (mind the space at the beginning )

(prefer IMAGE\_INSTALL\_append over IMAGE\_INSTALL +=)

core-image-sato

This is the X11 Window-system-based image with a SATO theme and a GNOME mobile desktop environment

$ bitbake core-image-sato

**Recepies:**

Yocto: A recipe is a set of instructions that is read and processed by the bitbake

Extension of Recipe: .bb

A recipe for a software component describes:

Where you get source code

Which patches to apply

Configuration options (library dependencies)

Install

License

A software component

Examples of Recipes:

dhcp\_4.4.1.bb

gstreamer1.0\_1.16.1.bb

**Configuration Files:**

Files which hold

Global definition 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

Extension: .conf

# Types –

* Machine Configuration Options
* Distribution Configuration Options
* Compiler tuning options
* General Common Configuration Options
* User Configuration Options (local.conf)

## Classes

Class files are used to abstract common functionality and share it amongst multiple recipe (.bb) files

To use a class file, you must simply make sure the recipe inherits the class

Eg. inherit classname

Extension - .bbclass

They are usually placed in classes directory inside the meta\* directory

# Example of classes

Cmake.bbclass – Handles cmake in recipes

Kernel.bbclass – Handles building kernels. Contains code to build all kernel trees

Module.bbclass – Provides support for building out-of-tree Linux Kernel Modules

**Layers:**

A collection of related recipes.

Or

Layers are recipe containers (folders)

Typical naming convention: meta-<layername>

Poky has the following layers:

Meta, meta-poky, meta-selftest, meta-skeleton, meta-yocto-bsp

### Why Layers

Layers provide a mechanism to isolate meta data according to functionality, for instance BSPs, distribution configuration, etc

You could have a BSP layer, a GUI layer, a distro configuration, middleware, or an application

Putting your entire build into one layer limits and complicates future customization and reuse.

Example: meta-poky -- Distro metadata

meta-yocto-bsp -- BSP metadata

Layers allow to easily to add entire sets of meta data and/or replace sets with other sets.

meta-poky, is itself a layer applied on top of the OE-Core metadata layer i.e meta.

# Which layers are used by Poky build system?

BBLAYERS variable present in build/conf/bblayers.conf file lists the layers Bitbake tries to find

If bblayers.conf is not present when you start the build, the OpenEmbedded build system creates it from bblayers.conf.sample when you source the oe-init-build-env script

# Command to find out which layers are present?

$ bitbake-layers show-layers

Note: You can include any number of available layers from the Yocto Project

# Where to get other layers from?

<https://layers/openembedded.org/layerindex/branch/master/layers/>

# Yocto Project Compatible Layers

<https://www.yoctoproject.org/software-overview/layers/>

These layers are tested and are fully compatible with yocto project.

OpenEmbedded layer index contains more layers but the content is less universally validated.

**Image:**

An image is the top level recipe, it has a description, a license and inherits the core-image class

It is used alongside the machine definition

Machine describes the hardware used and its capabilities

Image is architecture agnostic and defines how the root filesystem is built, with what packages.

By default, several images are provided in Poky

Command to check the list of available image recipes:

$ ls meta\*/recipes\*/images/\*.bb

**Packages:**

Yocto: A package is a binary file with name \*.rpm, \*.deb, or \*.ipkg

A single recipe produces many packages. All packages that a recipe generates are listed in the recipe variable.

$ vi meta/recipes-multimedia/libtiff/tiff\_4.0.10.bb

PACKAGES =+ “tiffxx tiff-utils”

#### Poky source tree

bitbake - Holds all Python scripts used by the bitbake command

bitbake/bin is palced into the PATH enivironmental variable so bitbake can be found

doucumentation - All documentation sources for the Yocto Project documentation   
 Can be used to generate nice PDFs

meta - Contains the oe-core metadata

meta-poky - Holds the configuration for the Poky reference distribution

local.conf.sample, bblayers.conf.sample are present here

meta-skeleton - Contains template recipes for BSP and kernel development

meta-yocto-bsp - Maintains several BSPs such as the Beaglebone, EdgeRouter and generic versions of both 32-bit and 64-bit IA machines.

scripts - Contains scripts used to set up the environment, development tools and tools to flash the generated images on the target.

LICENSE - The license under which Poky is distributed ( a mix of GPLv2 and MIT)

Qusetion: Where should I place the content of conf/local.conf as this file is part of bulid folder?

Ans: In general, everything in your local.conf should be moved to your own distro configuration

Finally, you should only set DISTRO to your own distro in local.conf

**conf**

When you run source poky/oe-init-build-env, it will create a “build” folder in that directory

Inside this build folder, it will create “conf” folder which contains two files:

1. local.conf
2. bblayers.conf

**local.conf**

Configures almost every aspect of the build system

Contains local user settings

MACHINE: The machine the target is built for

Eg: MACHINE = “qemux86-64”

DL\_DIR: Where to place downloads

During a first build the system will download many different source code tarballs, from various upstream projects. These are all stored in DL\_DIR

The default is a downloads directory under TOPDIR which is the build directory

TMP\_DIR: Where to place the build output

This option specifies where the bulk of the building work should be done and where BitBake should place its temporary files (source extraction, compilation) and output

**Important Point:**

local.conf file is a very convenient way to override several default configurations over all the Yocto Projects’s tools.

Essentially, we can change or set any variable, for example, add additional packages to an image file

Though it is convenient, it should be considered as a temporary change as the build/conf/local.conf file is usually not tracked by any source code managementsystem

**bblayers.conf**

The bblayers.conf file tells BitBake what layers you want considered during the build.

By default, the layers listed in this file include minimally needed by the build system

However, you must manually add any custom layers you have created

E.g: BBLAYERS = “\

/home/pruthvik/poky/meta \

/home/pruthvik/poky/meta-poky \

/home/pruthvik/poky/meta-yocto-bsp \

/home/pruthvik/poky/meta-mylayer \

“

This example enables four layers, one of which is a custom user defined layer named “meta-mylayer”

#### BB\_NUMBER\_THREADS

Determines the number of tasks that Bitbake will perform in parallel

Note: These tasks are related to bitbake and nothing relate to compiling

Defaults to the number of CPUs on the system

$ bitbake -e core-image-minimal | grep ^BB\_NUMBER\_THREADS=

#### PARALLEL\_MAKE

Corresponds to the -j make option

Specifies the number of processes that GNU make can run in parallel on a compilation task

Defaults to the number of CPUs on the system

$ bitbake -e core-image-minimal | grep ^PARALLEL\_MAKE=

##### Other Directories

downloads - downloaded upstream tarballs/git repositories of the recipes used in the

build

sstatet-cache - shared state cache

tmp - Holds all the build system output

tmp/deploy/images/machine - Images are present here

cache - cache used by the bitbake’s parser

#### Ycoto/OpenEmbedded Build System Workflow

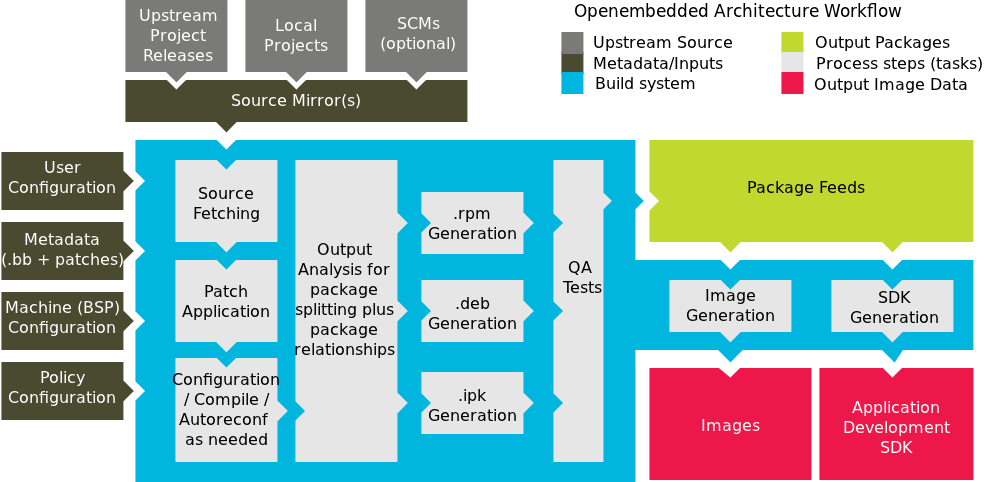
1. Developers specify architecture, policies, patches and configuration details.
2. The build system fetches and downloads the source code from the specified location

Supports downloading tarballs and source code repositories systems such as git/svn

1. Extracts the sources into a local work area
2. Patches are applied
3. Steps for configuring and compiling the software are run
4. Installs the software into a temporary staging area.

Depending on the user configuration, deb/rpm/ipk binaries are generated

1. The build system generates a binary package feed that is used to create the final root file image.
2. Finally generates the file system image and a Customized Extensible SDK (eSDK) for application development in parallel



#### Images Generated by Poky Build

The build process writes images out to the build directory inside the tmp/deploy/images/machine/ folder.

1. kernel-image:

A kernel binary file

The KERNEL\_IMAGETYPE variable determines the naming scheme for the kernel image file.

$ bitbake -e core-image-minimal | grep ^KERNEL\_IMAGETYPE=

1. root-filesystem-image:

Root filesystems for the target device (e.g \*.ext3 or \*.bz2 files).

The IMAGE\_FSTYPES variables determine the root filesystem image type

$ bitbake -e core-image-minimal | grep ^IMAGE\_FSTYPES=

1. kernel-modules;

Tarballs that contain all the modules built for the kernel

1. bootloaders:

If applicable to the target machine, bootloaders supporting the image.

symlinks

symbolic link pointing to the most recently built file for each machine

Threse links might be useful for external scripts that need to obtain latest version of each file.

### Saving disk space while building Yocto

Yocto build system can take a lot of disk space during build. But bitbake providesoptions to preserve disk space

You can tell bitbake to delete all the source code, build files after building a particular recipe by adding the following line in local.conf file

INHERIT += “rm\_work”

Disadvantage: Difficult to debug while build fails of any recipe.

For example. If you want to exclude bitbake deleting source code of a particular package, you can add it in RM\_WORK\_EXCLUDE += “recipe-name”

E.g: RM\_WORK\_EXCLUDE += “core-image-minimal”

##### What is BSP Layer?

A collection of information(metadata) that defines how to support

A particular device hardware device,

Set of devices, or

Hardware platform

Naming convention: meta-<bsp\_name>

How to find out what all hardware devices are supported?

Ans: conf/machine/\*.conf list all the hardware devices supported by the BSP layer

**meta-ti vs meta-yocto-bsp**

meta-yocto-bsp:

provides “reference” BSPs for each of the supported architectures

One for ARM (BeagleBone Black), one for MIPS, PPC and x86

It is based on the mainline kernel/bootloader

Does not support any advanced features or anything not in the upstream mainline kernel

e.g. no capes, no power management, no hardware acceleration, no 3D, no PRU, etc

The purpose of this BSP is to have some basic out-of-box experience for the select hardware platforms within Poky to evaluate the Yocto Project and OpenEmbedded framework, but not the specific hardware platforms

meta-ti:

official Texas instruments BSP that provides the latest WIP “staging” kernel and bootloader

most of the advanced features and peripheral support for the wider range of latest TI platforms

#### Adding Layers

Two ways:

1. Manual: edit bblayers.conf file and add the new layer to BBLAYERS
2. Automatic:

$ bitbake-layers add-layer <path-to-new-layer>

$ bitbake-layers add-layer ~/yocto/source/meta-ti/

PART – 2

**Operators:**

Basic Variable Settings (=)

VARIABLE = “value”

The following example sets VARIABLE to “value”

E.g. MACHINE = “raspberrypi3”

This assignment occurs immediately as the statement is parsed

It is a “hard” assignment

If you include leading or trailing spaces as part of an assignment, the spaces are retained:

VARIABLE = “ value”

VARIABLE = “value ”

Note: You can also use single quotes (‘ ‘) instead of double quotes when setting a variable’s vale

Benefit:

VARIABLE = ‘I have a “ in my value’

Q: How to check value of variable?

A: For configuration changes, use the following:

$ bitbake -e

This command displays variable values after the configuration files (i.e local.conf, bblayers.conf. bitbake.conf and so forth ) have been parsed.

For recipe changes, use the following

$ bitbake recipe -e | grep VARIABLE=”

Setting a default value (?=)

?= is used for soft assignment for a variable

Benefit:

Allows to define a variable if it is undefined when the statement is parsed,

If the variable has a value, then the soft assignment is lost

E.g:

MACHINE ?= “quemuram”

If MACHINE is already set before this statement is parsed, the above value is not assigned

If MACHINE is not set, then the above value is assigned

Note: Assignment is immediate

# What if we have multiple ?=

If multiple “?=” assignments to a single variable exist, the first of those ends up getting used

### Setting a weaker default value ??=

Weaker default value is achieved using the ??= operator

# Difference between ?= and ??=

Assignment is made at the end of the parsing process rather than immediately

When multiple ??= exist, the last one is used

It is called weak assignment, as assignment does not occur until the end of the parsing process.

Note: “=” or “?=” assignment will override the calue set with “??=”

### Variable Expansion

Variables can reference the contents of other variables using a syntax that is similar to variable expansion in Bourne shells

A = “hello”

B = “${A} world”

$ bitbake -e | grep ^A=

$ bitbake -e | grep ^B=

The “=” operator does not immediately expand variable references in the right-hand side

Instead, expansion is deferred until the variable assigned to is actually used

A = “${B} hello”

B = “${C} world”

C = “linux”

$ bitbake -e | grep ^A=

Q: What happens if C is not defined in above?

A: The string is kept as is

Immediate Variable Expansion (:=)

The “:=” operator results in a variable’s contents being expanded immediately, rather than when variable is actually used

# Appending operators

+=

A += “hello”

A += “world”

.=

A .= “hello”

A .= “world”

Difference between += and .= is space is automatically added in +=

These operators take immediate effect during parsing

# Prepending Operators

=+ and =.

Same as previous, =+ adds an additional space

Appending and Prepending (Override Style Syntax)

# In this syntax no space is inserted

A = “hello”

A\_append = “ world”

B = “full”

B\_prepend = “house”

# Reomval

You can remove values from lists using the removal override style syntax

Specifying a value for removal causes all occurrences of that value to be removed from the variable.

FOO = “123 456 789 123456 123 456 123 456”

FOO\_remove = “123”

$ bitbake -e | grep ^FOO=

# Override Style Operation Advantages

“\_append”, “\_prepend” and “\_remove” as compared to the “+=” and “=+” operators is that the override style operators provide guaranteed operations.

#### What is a layer?

A layer is a logical collection of related recipes

Types of Layers: oe-core, BSP Layer, application layer.

Layer name starts with meta- , but this is not a technical restriction

E.g meta-mycustom

# Why create a meta layer?

Despite most of the customization can be done with the local.conf configuration file, it is not possible to :

Store recipes for your own software projects

Create your own images

Consolidate patches/modifications to other people’s recipes

Add a new custom kernel

Add a new machine

Most important point: Do not edit POKY/UPSTREAM Layers, as it complicates future updates

Advantage: This allows you to easily port from one version of Poky to another

Depending on the type of layer, add the content:

If the layer is adding support for a machine, add the machine configuration in conf/machine/

If the layer is adding distro policy, add the distro configuration in conf/distro/

If the layer introduces new recipes, put the recipes you need in recipes-\* subdirectories of the layer directory.

Recipe directories inside layers:

By convention, recipes are splitted into categories

The most difficult part is deciding in which category your recipe will go

By checking what was already done in the official layers should give you a good idea of what you should do

Layer Priority:

Each layer has a priority, which is used by bitbake to decide which layer takes precedence if there are recipes files with the same name in multiple layers

A higher numeric value represents a higher priority.

Creating Layers:

There are two ways to create your own layer.

1. Manually
2. Using script

Manually:

Step 1: Create a directory for the layer. For example: ‘meta-mylayer’

Step 2: Create a conf/layer.conf

You can simply copy one meta-oe’s one and just change “openembedded-layer” to something appropriate for your layer; you may also want to set the priority as appropriate

Step 3: Update bblayers.conf file with the new layer

Creating layer using tool:

Create your own layer using the bitbake-layers create-layer command

$ bitbake-layers create-layer –help

The tool automates layer creation by setting up a subdirectory with a layer.conf configuration file, a recipes-example subdirectory that contains an example.bb recipe, a licensing file, and a README

$ bitbake-layers crate-layer ../source/meta-mylayer

Default priority of the layer is 6

$ bitbake-layers add-layer ../source/meta-mylayer

$ bitbake-layers show-layers

Layer Configuration File layer.conf:

# The configuration and classes directory is appended to BBPATH

BBPATH .= “:${LAYERDIR}”

# The recipes for the layers are appended to BBFILES

BBFILES += “${LAYERDIR}/recipes-\*/\*/\*.bb ${LAYERDIR}/recipes-\*/\*/\*.bbapend”

The BBFILE\_COLLECTIONS variable is then appended with the layer name

BBFILE\_COLLECTIONS += “skeleton”

BBFILE\_PATTERN\_skeleton = “^${LAYERDIR}/”

# The BBFILE\_PRIORITY variable then assigns a priority to the layer.

BBFILE\_PRIORITY\_skeleton = “1”

# This should only be incremented om significant changes that will cause compatibility issues with other layers

LAYERVERSION\_skeleton = “1”

LAYERVERSION\_skeleton = “core”

LAYERSERIES\_COMPAT\_skeleton = “kirkstone”

# Yocto-check-layer

The yocto-check-layer script provides you a way to asses how compatible your layer is with the Yocto Project

You should use this script if you are planning to apply for Yocto Compatible Program

$ source oe-init-build-env

$ yocto-check-layer your\_layer\_directory

#### What is an image?

Image is a top level recipe. (It inherits an image.bbclass)

Building an image creates an entire Linux distribution from source

Compiler, tools, libraries

BSP: Bootloader, Kernel

Root filesystem:

* Base OS
* Services
* Applications
* Etc

# Creating custom images

You often need to create you own Image recipe in order to add new packages or functionality

Two ways:

1. Creating an image from scratch
2. Extend an existing recipe (preferable)

**Package group:**

A package group is a set of packages that can be included on any image.

A package group can contain a set of packages.

Using the packagegroup name in IMAGE\_INSTALL variable install all the packages defined by the package group into root file system of your target image.

There are many package groups. They are present in subdirectories named “packagegeoups”

$find . -name ‘packagegroups’

They are recipe file(.bb) and start with packagegroup-

For example,

packagegroup-core-boot: Provides the minimum set of packages necessary to create a bootloader image with console.

1. Create an image from scratch:

The simplest way is to inherit the core-image bbclass, as it provides a set of image features that can be used very easily

inherit core-image

Which tells us that the definition of what actually is defined in the core-image.bbclass

Image recipes set IMAGE\_INSTALL to specify the packages to install into an image through image.bbclass

Create an image directory

$ mkdir -p recipes­-example/images

Create the image recipe

$ vi recipes-examples/images/lwl-image.bb

SUMMARY = “A small boot image for LWL learners”

LICENSE = “MIT”

Inherit core-image

# Core files for basic console boot

IMAGE\_INSTALL = “packagegroup-core-boot”

IMAGE\_ROOTFS\_SIZE ?= “8192”

#Add our needed appliations

IMAGE\_INSTALL += “usbutils”

1. Reusing an existing image

When an image mostly fits our needs and we need to do minor adjustments on it, it is very convenient to reuse its code

This makes code maintenance easier and highlights the functional differences

For example, if we want to include an application (lsusb)

Create another recipe:

$ vim recipes-examples/images/lwl-image-reuse.bb

require recipes-core/images/core-image-minimal.bb

IMAGE\_INSTALL\_append = “usbutils”

# Customizing Images Using Custom IMAGE\_FEATURES and EXTRA\_IMAGE\_FEATURES

Another method for customizing your image is to enable or disable high-level image features by using the IMAGE\_FEATURES and EXTRA\_IMAGE\_FEATURES variables

IMAGE\_FEATURES / EXTRA\_IMAGE\_FEATURES is made to enable special features for your image, such as empty password for root, debug image, special packages, x11, splash, ssh-server

What’s the difference between IMAGE\_FEATURES and EXTRA\_IMAGE\_FEATURES ?

A: Best practice is to

Use IMAGE\_FEATURES from a recipe

Use EXTRA\_IMAGE\_FEATURES from local.conf

How it works?

A: To understand how these features work, the best reference is meta/classes/core-image.bbclass

This class lists out available IMAGE\_FEATURES of which most map to package groups while some, such as debug-tweaks and read-only-rootfs, resolve as general configuration settings

In summary, the file looks at the contents of the IMAGE\_FEATURES variable and then maps or configures the feature accordingly

Based on this information, the build system automatically adds the appropriate packages or configurations to the IMAGE\_INSTALL variable

# Example of IMAGE\_FEATURES

To illustrate how you can use these variables to modify your image, consider an example that selects the SSH server.

The Yocto Project ships with two SSH servers you can use with your images: Dropbear and OpenSSH

OpenSSH is a well known standard SSH server implementation

Dropbear is a minimal SSH server appropriate for resource-constrained environments

By default, the core-image-sato image is configured to user Dropbear. The core-image-full-cmdline and core-image-lsb images both include OpenSSH.

The core-image-minimal image does not contain an SSH server.

debug-tweaks

In the default state local.conf file has EXTRA\_IMAGE\_FEATURES set to “debug-tweaks”

debug-tweaks features enable password-less login for the root user

Advantage: makes logging in for debugging or inspection easy during development

Disadvantage: anyone can easily log in during production.

So, you need to remove the ‘debug-tweaks’ feature from production image.

# Read-Only Root Filesystem

Why do we need read-only rootfs

A: Reduce wear on flash memory

Eliminate system file corruption

How to do it?

A: To create the read-only root filesystem, simplify add the “read-only-rootfs” feature to your image.

IMAGE\_FEATURES = “read-only-rootfs” in your recipe

Or

EXTRA\_IMAGE\_FEATURES += ”read-only-rootfs” in local.conf

# Boot Splash Screen

IMAGE\_FEATURES += “saplash”

Or

EXTRA\_IMAGE\_FEATURES += “saplash”

# Some other Features

tools-debug: Installs debugging tools such as strace and gdb.

tools-sdk: Installs a full SDK that runs on the device.

# IMAGE\_LINGUAS

Specifies the list of locales to install into the image during the root filesystem construction process

IMAGE\_LINGUAS = “zh-cn”

Inside quemu image

$ locale -a

# IMAGE\_FSTYPES

# The IMAGE\_FSTYPES variable determines the root filesystem image type

If more than one format is specified, one image per format will be generated

Image formats instructions are delivered in Poky: meta/classes/image\_types.bbclass

$ bitbake -e <image\_name> | grep ^IMAGE\_FSTYPES=

Types supported:

btrfs container cpio cpio.gz cpio.lz4 cpio.lzma cpio.xz

cramfs elf ext2 ext2.bz2 ext2.gz ext2.lzma ext3

ext3.gz ext4 ext4.gz f2fs hddimg iso jffs2 jffs2.sum

multiubi squashfs squashfs-lz4 squashfs-lzo squashfs-xz tar

tar.bz2 tar.gz tar.lz4 tar.xz ubi ubifs

wic wic.bz2 wic.gz wic.lzma

Creating your own image type:

If you have particular layout on your storage (for example bootloader location on an SD card), you may want to create your own image type

This is done through a class that inherits from image\_types

It has to define a function named IMAGE\_CMD\_<type>

Example: sdcard\_image-rpi.bbclass in meta-raspberrypi

# IMAGE\_NAME

The name of the output image files minus the extension

This variable is derived using the IMAGE\_BASENAME, MACHINE, and DATETIME variables

IMAGE\_NAME = “${IMAGE\_BASENAME}-${MACHINE}-${DATETIME}”

# IMAGE\_MANIFEST

The manifest file for the image

This file lists all the installed packages that make up the image

The file contains package information on a line-per-package basis as follows:

packagename packagearch version

The image class defines the manifest file as follows:

IMAGE\_MANIFEST = “${DEPLOY\_DIR\_IMAGE}/${IMAGE\_NAME}.rootfs.manifest”

#### Recipes

Recipes are fundamental concepts in the Yocto Project environment.

A Yocto/OpenEmbedded recipe is a text file with file extension .bb

Each software component built by the OpenEmbedded build system requires a recipe to define thes component

A recipe contains information about single piece of software.

# What information is present in a recipe?

Information such as :

Location from which to download the unaltered source

Any patches to be applied to that source (if needed)

Special configuration options to apply

How to compile the source files and

How to package the compiled output

Poky includes several classes that abstract the process for the most common development tools as projects based on Autotools, CMake and QMake.

Recipe File Format:

File Format: <base\_name>\_<version>.bb

For example the file dropbear\_2019.78.bb in poky/meta/recipes-core/dropbear has

base name : dropbear

version : 2019.78

Another Example:

file tiff\_4.0.10.bb in poky/meta/recipes-multimedia/libtiff/ has

base name : tiff

version : 4.0.10

The recipe is for a C library to read and write tiff image files

Note: Use lower-cased characters and do not include the reserved suffixes -native, -cross, -initial or

-dev

**Bitbake**

Yocto/OpenEmbedded’s build tool bitbake parses a recipe and generates list of tasks that it can execute to perform the build steps

$ bitbake basename

The most important tasks are:

do\_fetch Fetches the source code

do\_unpack Unpacks the source code into them to the source code

do\_patch Locate patch files and applies them to the source code

do\_configure Configures the source by enabling and disabling any build-time and configuration

options for the software being built

do\_compile Compiles the source in the compilation directory

do\_install Copies files form the compilation directory to a holding area

do\_package Analyses the content of the holding area and splits it into subsets

based on available packages and files

do\_package\_write\_rpm Creates the actual RPM packages and places them in the Package Feed Area

Generally, the only tasks that the user needs to specify in a recipe are

do\_configure,

do\_compile and

do\_install ones.

The remaining tasks are automatically defined by the YP build system

The above task list is in the correct dependency order. They are executed from top to bottom.

You can use the -c argument to execute the specific task of a recipe.

$ bitbake -c compile dropbear

To list all tasks of a particular recipe

$ bitbake <recipe name> -c listtasks

Stage 1: Fetching Code (do\_fetch)

The first thing your recipe must do is specify how to fetch the source files.

Fetching is controlled mainly through the SRC\_URI variable

Your recipe must have a SRC\_URI variable that points where the source is located.

The SRC\_URI variable in your recipe must define each unique location for your source files.

Bitbake supports fetching source code from git, svn, https, ftp, etc

URI scheme syntax: scheme://url;param1;param2

scheme can describe a local file using file:// or remote locations with https://, git://, svn://, hg://, ftp://

By default, sources are fetched in $BUILDDIR/downloads

###### Examples of SRI\_URI

busybox\_1.31.0.bb : SRC\_URI = “[https://busybox.net/downloads/busybox-${PV}.tar.bz2](https://busybox.net/downloads/busybox-$%7bPV%7d.tar.bz2)”

linux-yocto\_5.2.bb : SRC\_URI = “git://git.yoctoproject.org/linux-yocto.git”

Weston-init.bb : SRC\_URI = “<file://init>”

The do\_fetch task uses the prefix of each entry in the SRC\_URI variable value to determine how to

fetch the source code.

Note: Any patch present, needs to be specified om SRC\_URI

Stage 2: Unpacking (so\_unpack):

All local files found in SRC\_URI are copied into the recipe’s working directory, in $BUILDDIR/tmp/work/

When extracting a tarball, Bitbake expects to find the extracted files in a directory named <application>-<version> . This is controlled by the S variable.

If the tarball follows the above format, then you need not define S variable.

Eg. SRC\_URI = [https://busybox.net/downloads/busybox-${PV}.tar.bz2;name=tarball](https://busybox.net/downloads/busybox-$%7bPV%7d.tar.bz2;name=tarball)

If the directory has another name, you must explicitly define S

If you are fetching from SCM like git or SVN, or your file is local to your machine, you need to define S

If the scheme is git, S - ${WORKDIR}/git

Stage 3: Patching Code (do\_patch)

Sometimes it is necessary to patch code after it has been fetched.

Any files mentioned in SRC\_URI whose name end in .patch or .diff or compressed versions of these

suffixes (e.g diff.gz) are treated as patches

They do\_patch task automatically applies these patches.

The build system should be able to apply patches with the “-p1” option (i.e one directory level in the

Path will be stripped off).

If your patch needs to have more directory levels stripped off, specify the number of levels using the

“striplevel” option in the SRC\_URI entry for the patch

#### Licensing

Your recipe needs to have both the LICENSE and LIC\_FILES\_CSKSUM variables.

License: for standard licenses, us the names of the files in meta/files/common-licenses/

# LIC\_FILES\_CHKSUM

The OpenEmbedded build system uses this variable to make sure the license text has not changed.

If it has, the build produces and error and it affords you the chance to figure out the correct the problem.

Example that assumes the software has a COPYING file:

LIC\_FILES\_CHKSUM = “<file://COPYING;md5=xxx>”

Stage 4: Configuration (do\_configure)

Most software provides some means of setting build-time configuration options before compilation

Typically, setting these options is accomplished by running a configure script with options, or by modifying a build configuration file

Autotools: If your source files have a configure.ac file, then your software is built using Autotools.

CMake: If your source files have a CMakeLists.txt file, then your software is built using CMake

If your source files do not have a configure.ac or CMakeLists.txt file, you normally need to provide a do\_configure task in your recipe unless there is nothing to configure.

# Stage 5: Compilation (do\_compile)

do\_compilation task happens after source is fetched, unpacked, and configured.

Stage 6: Installation (do\_install)

After compilation completes, Bitbake executes the do\_install task

During do\_install, the task copies the built files along with their hierarchy to locations that would mirror their locations on the target device.

Stage 7: Packaging (do\_package)

The do\_package task splits the files produced by the recipe into logical components.

Even software that produces a single binary might still have debug symbols, documentation, and other logical components that should be split out.

The do\_package task ensures that files are split up and packaged correctly.

#### Example

Step 1: Create a file userprog.c with the following content:

#include <stdio.h>

int main()

{

printf(“Hello World\n”);

return 0;

}

Step 2: Create folder in the layer recipes-example ‘myhello’

mkdir -p recipes-example/myhello

Step 3: Create ’files’ folder inside the ‘myhello’ folder and copy userprog.c inside this folder

mkdir -p recipes-examples/myhello/files

copy the userprog.c into the above location

Step 4: Create a file called ‘myhello\_0.1.bb’ with the following content:

DESCRIPTION = “Simple helloworld application”

LICENSE = “MIT”

LIC\_FILES\_CHKSUM = “file://${COMMON\_LICENSE\_DIR}/MIT;0835ade698e0bcf850ecda2f7b4f302”

SRC\_URI = “<file://userprog.c>”

S = “${WORKDIR}”

do\_compile() {

${CC} userprog.c ${LDFLAGS} -o userprog

}

do\_install() {

install -d ${D}${binddir}

install -m 0755 userprog ${D}${bindir}

}

Step 5: bitbake myhello

do\_install keyword : not only copies files but also changes ownership and permissions and optionally removes debugging symbols from executables. It combines cp with chown, chomd and strip

# OpenEmbedded Varibles

S : Contains the unpacked source files for a given recipe

D : The destination directory (root directory of where the files are installed, before creating the image)

WORKDIR : The location where OpneEmbedded builds a recipe (i.e does the work to create the package).

PN: The name of the recipe used to build the package

PV: The version of the recipe used to build the package

PR: The revision of the recipe used to build the package

# WORKDIR

The location of the work directory in which the OpenEmbedded build system builds a recipe.

This directory is located within the TMPDIR directory structure and is specific to the recipe being built and the system for which it is being built.

The WORKDIR directory is defined as follows:

${TMPDIR}/work/${MULTIMATCH\_TARGET\_SYS}/${PN}/${EXTENDPE}${PV}-${PR}

TMPDIR: The top-level build output directory

MULTIMACH\_TARGET\_SYS: The target system identifier

PN: The recipe name

EXTENDPE: Mostly blank

PV: The recipe version

PR: The recipe revision

#### Recipe Explanation

The most relevant tasks that will be executed when calling bitbake myhello are the following:

$ bitbake -c cleanall myhello

1. do\_fetch:

in this case, since the specified SRC\_URI variable points to a local file, Bitbake will simply copy the file in the recipe WORKDIR.

This is why the S environment variable (which represents the source code location) is set to WORKDIR.

$ bitbake -c fetch myhello

$ bitbake -c unpack myhello

$ bitbake -c configure myhello

What is **sysroot**?

Contains needed headers and libraries for generating binaries that run on the target architecture

recipe-sysroot-native:

includes the build dependencies used in the host system during the build process.

It is critical to the cross-compilation process because it encompasses the compiler, linker, build script tools, and more.

recipe-sysroot:

the libraries and headers used in the target code

1. do\_compile :

when executing this task, BB will invoke the C cross-compiler for compiling the myhello.c source file.

The results of the compilation will be in the folder pointed by the B environment variable (that, in most of the cases, is the same as the S folder).

1. do\_install:

This task specifies where the helloworld binary should be installed into the rootfs.

It must be noticed that this installation will only happen within a temporary rootfs folder within the recipe WORKDIR (pointed by the variable D)

Image: this contains the files installed by the recipe (pointed to D variable).

1. do\_package:

in this phase the file installed in the directory D will be package named myhello.

This package will be used later from BitBake when eventually building a rootfs image containing the helloworld recipe package

packages: The extracted contents of packages are stored here

packages-split: The contents of packages, extracted and split, are stored here. This has a sub-directory for each package.

Add recipe to rootfs

IMAGE\_INSTALL += “myhello”

# Who defines the fetch, configure and other tasks?

When bitbake is run to build a recipe, base.bbclass file gets inherited automatically by any recipe

You can find it in classes/base.bbclass

This class contains definitions for standard basic tasks such as fetching, unpacking, configuring (empty by default), compiling (runs any Makefile persent), installing (empty by default) and packaging (empty by default).

These classes are often overridden or extended by other classes such as the autotools calss or package class.

**Logs**

# Where do I find logs?

Every build produces lots of logs output for diagnostic and error chasing

Output of bitbake gets logged to tmp/log/cooker/<machine>/

$ cat tmp/log/cooker/<machine>/<timestamp>.log | grep ‘Note:.\*task.\*Started’

# Where do I find build logs?

For each individual recipe, there is a “temp” directory under the work directory

Within the build system this directory is pointed to by the T variable, so if you need to you can find it by using bitbake -e

$ bitbake -e <recipename> | grep ^T=

Each task that runs for a recipe produces “log” and “run” files in ${WORKDIR}/temp

You can find log files for each task in the recipe’s temp directory

Log files are named log.taskname (e,g log.do\_configure, log.do\_fetch, log.do\_compile)

For convenience, symbolic links are kept updated by BitBake, pointing to the latest log files using the pattern log.<task>

We can run the scripts for every task with the pattern run.<task>.<pid>

These files contain the commands which produce the build results

# Logging Information during task execution

The logging utilities provided by BitBake are very useful to trace the code execution path.

BitBake provides logging functions for use in Python and Shell Script code, as described

Python: For use within Python functions, BitBake supports several log levels, which are bb.fatal, bb.error, bb.warn, bb.note, bb.palin, and bb.debug

Shell Script: For use in Shell Script functions, the same set of log levels exist and is accessed with a similar sytax: bbfatal, bberror, bbearn, bbnote, bbpalin and bbdebug.

There is one subtle difference between the use of the logging functions in Python and Shell Scripting.

The logging functions in Python are directly handled by BitBake, seen on the console, and stored in the execution log that can be seen inside build/tmp/log/cooker/<machine>

The logging functions in Python are directly handled by BitBake, seen on the console, and stored in the execution log that can be seen inside build/temp/log/cooker/<machine>

When the logging functions are used in Shell Script, the information is outputted on the task’s respective task log file, which is available in build/tmp/work/<arch>/<recipe name>/<software version>/temp

# Logging Levels

bb.fatal and bbfatal: These have the highest priority of logging messages as they print the message and terminate the processing. They cause the build to be interrupted.

bb.error and bberror: They are used to display an error but do not force the build to stop.

bb.warn and bbwarn: They are used to warn the users about something.

bb.note and bbnote: These add a note to the user. They are only informative.

bb.palin and bbplain: These output a message.

bb.debug and bbdebug: These add debugging information that is shown depending on the debug level used.

do\_compile() {

bbpalin “\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*”

bbplain “\* \*”

bbplain “\* Example recipe created by bitbake-layers \*”

bbplain “\* \*”

bbpalin “\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*”

}

# Debug Output

You can see the debug output from BitBake by using the -D option

The debug output gives more information about what BitBake is doing and the reason behind it

Each -D option you use increases the logging level

The most common usage is -DDD

**oe\_runmake**

The default behaviour of do\_compile task is to run the oe\_runmake function if a makefile (Makefile, makefile or GNUMakefile) if found, If no such file is found, the do\_compile tasks does nothing

The default behaviour of do\_configure task is to run oe\_runmake clean if a makefile (Makefile, makefile or GNUMakefile) is found.

oe\_runmake vs make

or\_runmake function is used to run make.

oe\_runmake

passes EXTRA\_OEMAKE settings to make

displays the make command

checks for errors generated via the call

In OE environment you should not call make directly rather use oe\_runmake when you need to run make.

oe\_runmake is one of many helper functions defined by the bb base class

Try: check the definition of oe\_runmake in base.bbclass

# EXTRA\_OEMAKE

If you need additional make options, you should store them in the EXTRA\_OEMAKE variable

# What if I have a Makefile which doesn’t have clean target?

CLEANBROKEN = “1”

If set to “1” within a recipe, CLEANBROKEN specifies that the make clean command does not work for the software being built.

Consequently, the OpenEmbedded build system will not try to run make clean during the do\_configure task, which is the default behaviour.

**Recipe for Git Remote Repository**

# Write a recipe for git remote repository

Yocto supports the ability to pull code from online git repositories as part of the build process.

Step 1: Set SRC\_URI

SRC\_URI = “git://<URL>;protocol=https”

Step 2: Set S environment variable

S = ${WORKDIR}/git

Step 3: Set SRCREV environment variable

# What is the use of SRCREV?

When fetching a repository, bitbake uses SRCREV variable to determine the specific revision from which to build

There are two options to this variable:

1. AUTOREV:

SRCREV = “${AUTOREV}”

When SRCREV is set to the value of this variable, it specifies to use the latest source revision in the repository

The build system accesses the network in an attempt to determine the latest version of software in the repository

1. A specific revision (SHA1 has when fetching from git)

If you want to build a fixed revision and you want to avoid performing a query on the remote repository every time BitBake parses your recipe

SRCREV = “d5651as6551651651a565d1516d6sads65asd66asd”

# Specifying a different branch

You can specify the branch using the following from:

SRC\_URI = “git://server.name/repository;branch=branchname”

If you do not specify a branch, BitBake looks in the default “master” branch

BitBake will now validate the SRCREV value against the branch.

# Fetch form a local git source

git fetcher supports protocol parameter

protocol:

The protocol used to fetch the files.

The default is “git” when a hostname is set. If a hostname is not set, the Git protocol is “file”

SRC\_URI = “git:///home/user/git/myTest/;protocol=file”

# What about private repositories?

Private repositories have the added complexity of requiring authentication before you can download (a.k.a clone) them.

Luckily, Yocto supports the ssh protocol.

To make Yocto use a GitHub Repository:

SRC\_URI = “git://git@github.com/group\_name/repo\_name.git;protocol=ssh”

To use SSH, you must generate an SSH keypair on your computer and add the public key to your GitHub account.

# How to pass a tag to a git recipe

tag parameter of git fetcher

“tag”: Specifies a tag to use for the checkout. To correctly resolve tags, BitBake must access the network

SRCREV is not needed in this case

# Patching the source for a recipe

Advantage of Yocto is that everything is built from source.

It’s fairly easy to make changes to anything that gets built.

As part of building a recipe, OE creates a tmp/work/<architecture>/<recipe>/<version> directory, known as the “work directory”.

This is where all of the work done to build a recipe takes place.

A subdirectory contains the source of recipe named <recipename>-<version> or “git” (depending on how the fetched source is provided)

The temptation is to simply make changes here and then recompile, but there are several reasons why that’s not a good idea:

1. You can easily lose your changes if you’re not careful e.g running bitbake -c clean will wipe the directory out
2. You have to force the compilation as the build system doesn’t know that you’ve made any changes

Bitbake -c compile <recipe> -f

# Patches:

Patches can be easily create using Git

If you download the third-party source code as a Git repository, this is definitely the easiest solution

After downloading the repository, make the required changes to the code, and add these changes to the repository as a new commit.

You can tell Git to make a patch file.

If all the changes are contained within a single additional commit, you can use the following command:

$ git show HEAD > my-patch.patch

These generated patches should be bundled with recipe files.

Patches should always be in a sub-directory of where the recipe lives.

Yocto will automatically apply these patches when it needs to build your recipe.

PART – 3

# **Splitting Files**

The do\_package task splits the files produced by the recipe during do\_install into locgical components.

Even software that produces a single binary might still have

Debug symbols

Documentation and

other logical components that should be split ou

This separation exists because not all of those installed files are useful in every image

For example, you would probably do not need any of the documentation installed in a production image

The do\_package task ensures that files are split up and packaged correctly

The PACKAGES and FILES variables controls splitting

PACKAGES list all of the packages to be produced

FILES specifies which files to include in each package by using an override to specify the package

PACKAGES:

PACKAGES - The list of packages the recipe creates

The default value: ${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-locale ${PACKAGE\_BEFORE\_PN} ${PN}

FILES - The list of files and directories that are placed in a package

To use the FILES variable, provide a package name override that identifies the resulting package

Eg. FILES\_${PN} specifies the files to go into the main package

Then, provide a space-separated list of files or paths that identify the files you want included as part

of the resulting image.

Eg. FILES\_${PN} += “${bindir}/mydir1 ${bindir}/mydir2/myfile”

Good Practice : use ${sysconfdir} rather than /etc or ${bindir} rather than /usr/bin

List of the variables can be found in **meta/conf/bitbake.conf**

What are the default values of various FILES\_\* variables?

Consequently, you might find you do not even need to set these variables in your recipes unless the software the recipe is building installs files into non-standard locations.

The PACKAGES and FILES\_\* variables in the meta/conf/bitbake.conf configuration file define how files installed by the do\_install task are packaged

Important Point :

For each installed file, the first package whose FILES value matches the file is the package into which to file goes

If a file matches the FILES variable for more than one package in PACKAGES, it will be assigned to the earliest (leftmost) package.

FILES\_${PN}-dbg += “${bindir}/userprog”

FILES\_${PN} += “${bindir}/userprog”

# Files/ directories were installed but not shipped in any package [installed-vs-shipped]

Files have been installed within the so\_install task but not have been included in any package by the way of FILES variable.

FILES do not appear in any package cannot be present in an image later on in the build process

You need to do one of the following:

Add the files to FILES for the package you want them to appear in (e.g. FILES\_${PN} for the main package).

Delete the files at the end of the do\_install task if the files are not needed in any package.

#### Create a Static Library

A static library is basically a set of object files that were copied into a single file with the suffix .a

The basic tool used to create static libraries is a program called **ar** (archiver).

This program can be used to

Create static libraries (also known as archive files)

Modify object files in the static library

List the names of object files in the library, etc.

In order to create a static library, we have to perform two steps:

1. Create the object files form the source files of the project
2. Create the static library (the archive file) from the object files

# Create object files

$ gcc -c arith.c

$ gcc -c print.c

# Create Static Library

$ ar rcs liblwl.a arith.o print.o

c – create the archive if it doesn’t exist

r – replace the older object files in the library, with the new object files

s – write an object-file index into the archive

this index will later be used by the compiler to speed up symbol-lookup inside the library

# Linking static library into application

$ gcc userprog.c -o userprog -llwl -L

The -L flag indicates (a non standard) directory where the libraries can be found, else you can copy this in standard location (/usr/lib)

The -l flag indicates the name of the library

Note the -larith will be converted to libarith.a by the compiler

# Write a Static Library recipe

File Location:

.a -> ${libdir}

.h -> ${includedir}

# ALLOW\_EMPTY

By default, BitBake does not produce empty packages

This default behaviour can cause issues when there is an RDEPENDS or some other hard runtime on the existence of the package

ALLOW\_PACKAGE variable specifies whether to produce an output package even if is empty.

Ex:

ALLOW\_EMPTY\_${PN} = “1”

ALLOW\_EMPTY\_${PN}-dev = “1”

ALLOW\_EMPTY\_${PN}-staticdev = “1”

**How to create the dynamic library:**

Step 1: Create object files using the below command

$ gcc -fPIC -c \*.c

The -fPIC flag stands for Postion Independent Code, a characteristic required by shared libraries

Step 2: Create the library

$ gcc -shared -Wl, -soname, liblwl.so.1 -o liblwl.so.1.0 \*.o

The shared key tells the compiler to produce a shared object which can then be linked with other objects to from an executable.

-Wl flag passes options to linker with following format -Wl, options

In case of our example it sets the name of library, as it will be passed to the linker.

# Use the dynamic library

$ cd app

$ gcc main.c -o main -I.. -lairth -L..

# Shared Libraries Names

Dynamic libraries follow certain naming conventions on running systems so that multiple versions can co-exist

Linux shared library can have three names. Which are:

Linker name (e.g: libexample.so)

Sonname (e.g: libexample.so.1.2)

Real Name (e.g: libexample.so.1.2.3)

# Linker Name

Linker name is the name that is requested by the linker when another code is linked with your library (with -lexample linker option).

Linker name typically starts with

The prefix *lib*

Name of the library

The phrase .so

soname

Every shared library has a special name called the ``soname’’.

The soname has

The prefix *lib*

The name of the library

The phrase .so

Followed by a period

And a version number that is incremented whenever the interface changes

Eg. liblwl.so.1

# Real Name

Real Name is the actual name of the shared library file.

Real Name = soname + minor version number

Eg. liblwl.so.1.0

It can also be liblwl.so.1.0.1.3

# During share library installation

soname is a symbolic link to the real name

linker name is a symbolic link to the soname

In this way both soname and linker names ultimately point to real name of the library (i.e actual library file).

# Command to read the soname

$ readelf -d libwl.so

# Write a Dynamic Library Recipe

File location :

.so -> ${libdir}

.h -> ${includedir}

# Why is liblwl.so is not present in the image?

The un-versioned symbolic link is only used at development time.

Packaging Un-versioned libraries

When shared libraries are built, they should be versioned but sometimes this is not done

Some library vendors do not release source code for their software but do release pre-bulit binaries

You will get error when you add un-versioned libraries

Because versioned libraries are far so common than un-versioned libraries, the default packaging rules assume versioned libraries

# PACKAGING Un-versioned Libraries

It follows that packaging an un-versioned library requires a bit of work in the recipe.

By default, “liblwl.so” gets packaged into ${PN}-dev, which triggers a QA warning that a non-symlink library is in a -dev package

To solve this problem, you need to package the un-versioned library into ${PN} where it belongs

Following are the abridged defaults FILES variables in bitbake.conf

SOLIBS = “.so.\*”

SOLIBSDEV = “.so”

FILES\_${PN} = “… ${libdir}/lib\*${SOLIBSDEV} …”

FILES\_SOLIBSDEV ?= “… ${libdir}/lib\*${SOLIBSDEV} …”

FILES\_${PN}-dev = “… ${FILES\_SOLIBSDEV} …”

SOLIBS defines a pattern that matches real shared object libraries.

SOLIBSDEV matches the development from (unversioned symlink).

These two variables are then used in FILES\_${PN} and FILES\_${PN}-dev

Which puts the real libraries into ${PN} and the unversioned symbolic link into PN-dev.

To package unversioned libraries, you need to modify the variables in the recipe as follows:

SOLIBS = “.so”

FILES\_SOLIBDEV = “”

The modification cause .so to be real library and unsets FILES\_SOLIBDEV so that no libraries get packed into PN-dev.

#### Dependencies

Most software packages have a short list of other packages that they require, which are called dependencies

e.g make menuconfig need ncurses library.

These dependencies fall into two main categories

build-time dependencies required when the software is built

runtime dependencies: which are required to be installed on the target in order for the software to execute

Examples:

Build Time Dependencies:

Your software uses a particular library. For example pthread/openssl.

Application cannot build without pthread/openssl library

Run Time Dependencies:

If your software internally calls a particular command for example (lspci)

If your software uses run time loading of dynamic library (dlopen)

Application/Software can build, but need them during execution.

##### Yocto Variables

There are two variables provided by Yocto to allow specifications of dependencies

DEPENDS : Specifies build-time dependencies, via a list of bitbake recipes to build prior to building

the recipe

RDEPENDS : Specifies run-time dependencies, via a list of packages to install prior to installing the

current package

# DEPENDS

Within a recipe, you can specify build time dependencies using DEPENDS variable

It is important that you specify all build-time dependencies explicitly.

When a recipe ‘A’ is DEPENDS on recipe ‘B’. In this case, Bitbake first builds recipe ‘B’ and then recipe ‘A’

Expamle: Adding recipe1 to recipe2 as build dependency

DEPENDS = “recipe1”

do\_populate\_sysroot

Copies a subset of files installed by the do\_install task into the appropriate sysroot.

Check the log of do\_populate\_sysroot in recipe2

sysroot-destdir/ : Contains a subset of files installed within do\_install that have been put into the shared sysroot.

# Sharing Files Between Recipes

Recipes often need to use files provided by other recipes on the build host .

For example, an application linking to a common library needs access to the library itself and its associated headers.

The way this access is accomplished is through the sysroot.

One sysroot exists per machine:

A sysroot exists for the target machine.

A sysroot exists for the build host.

Subset of files installed into standard locations during the do\_install task within the ${D} directory automatically goes into the sysroot.

do\_prepare\_recipe\_sysroot

this task sets up the two sysroots in ${WORKDIR} (i.e recipe-sysroot and recipe-sysroot-native)

\*\*\*\*

Try :

Convert the myhello recipe to use ‘static’ and ‘dynamic’ library

Check recipes-sysroot after you do ‘bitbake -c configure myhello’

\*\*\*\*

RDEPENDS

Within a recipe, run time dependencies can be specified using RDEPENDS variable

If your recipe says that T RDEPENDS on P, that tells bitbake that it must deploy P to the target system

if it deploys T, because T can’t be used without P.

Difference between DEPENDS and RDEPENDS

DEPENDS List of the recipe build time dependencies

RDEPENDS List of the package runtime dependencies. Must be package specific (e.g with\_${PN})

RDEPENDS\_${PN} = “package\_name”

If T RDEPENDS on P then T’s do\_build task is made to depend on P’s do\_package\_write task

# Dependency on a specific version

Sometimes a recipe have dependencies on specific versions of another recipe

Bitbake allows to reflect this by using:

DEPENDS = “recipe-b (>= 1.2)”

RDEPENDS\_${PN} = “recipe-b {>= 1.2}”

The following are supported: =, >, <, >= and <=

# Recipe Dependency Information

Dependency information can help you understand why a recipe is built.

To generate dependency information for a recipe, run the following command:

$ bitbake -g recipename

This command writes the following files in the current directory:

pn-buildlist

A list of recipes/targets involved in building recipename.

“Involved” here means that at least one task from the recipe needs to run when building recipename from scratch

task-depends.dot

A graph showing dependencies between tasks

The graphs are in DOT format and can be converted to images (e.g using the dot tool from Graphviz)

# DOT Format

The DOT format is a text description language for graphics that is understood by the GraphViz open source package and all utilities that use it

$ dot -Tpdf task-depends.dot -o outfile.pdf

$ envince outfile.pdf

The most useful way to display dependency data is to ask Bitbake to display it graphically with the dependency explorer, as follows:

Bitbake -g -u taskexp recipename

This command displays a GUI window from which you can view build-time and runtime dependencies for the recipes involved in building recipenamed

noexec

When set to “1”, marks the task as being empty, with no execution required.

Eg: do\_configure[noexec] = “1”

do\_compile[noecec] = “1”

This can be used to disable tasks which are defined elsewhere for example in a class that are not needed in a particular recipe.

#### CMAKE

# What is CMAKE?

Cmake is a cross-platform free and open source software tool for managing the build process of software using a compiler-independent method.

# CMAKE\_INSTALL\_PREFIX

If “make install” is invoked or INSTALL is built, this directory is prepended onto all directories.

This variable defaults to /usr/local/ on UNIX

To install your binaries in /usr/bin

$ cmake -DCMAKE\_INSTALL\_PREFIX=/usr ..

$ make

$ make install

##### Recipe for CMake package

Applications that use cmake require

A recipe that has a source archive listed in SRC\_URI

Also inherit the cmake class

Generally the CMake build system knows how to install the software so a over do\_install is not necessary

# EXTRA\_OECMAKE

Variable can be used to pass any needed configure options that are specific to the recipe

#### DevShell – Development Shell

The Yocto build system handles all the steps needed to build software from scratch by following the Yocto recipes.

When editing packages or debugging build failures, a development shell can be a useful tool.

# What is devshell?

The devshell is a terminal shell that runs in the same context as the Bitbake task engine

In the new terminal, all the environment variables needed for the build are still defined, so we can use commands such as configure and make

The commands execute just as if the build system was executing them.

Command:

$ bitbake -c devshell <recipename>

$ bitbake -c devshell myhello

# What happens when you run the command?

Strats a shell whose environment is set up for development, debugging or both.

All tasks up to and terminal and including do\_patch are run for the specified target.

Then, a new terminal is opened and you are placed in ${S}, the source directory.

When you are finished using devshell, exit the shell or close the terminal window.

Devshell

Devshell provides you with an interactive shell with all the appropriate variables set for cross-compiling

$ echo $CC

$ echo $PATH

$ echo $CXX

$ echo $LDFLAGS

$ echo $CFLAGS

Now you can make modifications and compile and generate an executable

$ ${CC} -DUSE\_SYSCALL user.prog ${LDFLAGS} -o userprog

To manually run a specific task using devshell, run the corresponding run.\* script in the ${WORKDIR}/temp directory (e.g run.do\_configure.pid)

If a task’s script does not exist, you can create the task by first running it outside of the devshell:

$ bitbake -c task

Execution of a task’s run .\* script and Bitbake’s execution of a task are identical.

In other words, running the script re-runs the task just as it would be running using the bitbake -c command.

Any run.\* file that does not have a .pid extension is a symbolic link (symlink) to the most recent version of that file.

It is important to bear in mid that all changes made inside devshell are not persistent between builds; thus, we must be careful to record any changes that is important, prior to leaving it

As we have the source at out disposal, we can use it to generate extra patches.

One very practical way of doing that is using Git and git format-patch to create the patch to be included in the recipe afterwards.

If the recipe doesn’t use git, we need to create repository

$ git init

$ git add userprog.c

$ git commit -m “First commit”

Make modifications

$ git status

$ git add userprog.c

$ git commit -m “Second Commit”

$ git format-patch -1 -o ~/yocto/source/meta-mylayer/recipes-examples/myhello/files/

Update SRC\_URI now

#### Understanding file searching paths

When a file (a patch or a generic file) is included in SRC\_URI, Bitbake searches for FILESPATH and FILEXTENSION variables.

The default value for the FILESPATH variable is defined in the **base.bbclass** class found in **meta/classes**

FILESPATH = “${@base\_set\_filespath([“${FILE\_DIRNAME}/${BP}”, “${FILE\_DIRNAME}/${BPN}”, “${FILE\_DIRNAME}/files”], d)}”

The value is a colon-separated list of directories that are searched left-to-right in order.

The default setting is to look in the following locations:  
 <recipe>-<version>

<recipe>

<files>

$ bitbake -e myhello | grep ^FILESPATH=

Note: Do not hand-edit the FILESPATH variable

# FILESOVERRIDES

Custom paths and files can be added using FILESEXTRAPATHS and FILESOVERRIDES.

The FILESPATH variable is automatically extended using the overrides from the FILESOVERRIDES variable

FILESOVERRIDES = “${MACHINESOVERRIDES}:${DISTROOVERRIDES}”

$ bitbake -e myhello | grep ^ FILESOVERRIDES=

You can take advantage of this searching behaviour in useful ways.

Try: Change the MACHINE in local.conf and check FILESOVERRIDES value

For example, you want to have different defconfigs for different Machines

files/defconfig

files/MACHINEA/defconfig

files/MACHINEB/defconfig

SRC\_URI = “[file://defconfig](file://deconfig)”

When MACHINE = MACHINEA build system uses file from files/MACHINEA.

When MACHINE = MACHINEB build system uses file from files/MACHINEB.

For other machines, it will use files/defconfig

# FILESEXTRAPATHS

Extends the search path the OpenEmbedded build system uses when looking for files and patches as it processes recipes and append files.

FILESEXTRAPATHS\_prepend := “${THISDIR}:”

THISDIR The directory in which the file BitBake is currently parsing is located.

Best practices dictate that you accomplish this by using FILESEXTRAPATHS from within a .bbappend file and that you prepend paths as follows:  
FILESEXTRAPATHS\_prepend := “${THISDIR}/{PN}:”

#### Splash Screen

We have enabled splash screen by

IMAGE\_FEATURES += “splash”

We get a splash screen with yocto logo

psplash

By default, this screen is provided by psplash

psplash is a userspace graphical boot screen for mainly embedded Linux devices supporting a 16bpp or 32bpp framebuffer

Recipe: psplash found under “poky/meta-poky/recipes-core”

It uses an encoded image psplash-poky-img.h which it compiles /usr/bin/psplash to binary

# Changing custom splash screen

psplash expects an image to be in header file format, so you need to convert your image into a header file format by using script called “make-image-header.sh”

source – <http://git.yoctoproject.org/cgit/cgit.cgi/tree/make-image-header.sh>

Command:

./make-image-header.sh <path\_to\_image\_file> <NAME>

$ ./make-image-header.sh logo.png POKY

Rename it to psplash-poky-img.h

We can give a different NAME but it requires modification to source psplash.c

$ mv logo-img.h psplash-poky-img.h

Replace your generated image header in poky/meta-poky/recipes-core/psplash/files

Extending a Recipe

It is a good practice not to modify recipes available in Poky

But it is sometimes useful to modify an existing recipe

# Copy Approach:

Copy recipe in your layer and make modifications in the recipe

Extend Approach:

The Bitbake build engine allows to modify a recipe by extending it

# What is Bitbake Append?

A recipe that appends Metadata to another recipe is called a Bitbake append file.

File Extension: .bbappend

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

.bbappend file resides in your layer

.bb recipe file to which you are appending Metadata resides in a different layer.

# Benefits of bbappend

If you were copying recipes, you would have to manually merge changes as they occur.

Being able to append information to existing recipe not only avoids duplication, but also automatically applies changes from a different layer into your layer

# Points to Consider while working with .bbappend

1. Append files must have the same root name as the recipe they extended

Eg. example\_0.1.bbappend applies to example\_0.1.bb

This means the original recipe and append file names are version number-specific.

If the corresponding recipe is renamed to update to newer version, you must also rename and possibly update the corresponding .bbappend as well

During the build process, Bitbake displays an error on starting if it detects a .bbapend file that does not have a corresponding recipe with a matching name

2. If adding new files, you must prepend the FILESEXTRAPATHS variable with the path to file’s directory.

# Prioritizing Your Layer

Each layer is assigned a priority value.

Priority values control which layer takes precedence if there are recipe files with same name in multiple layers.

For these cases, the recipe files from the layer with a higher priority number takes precedence.

Priority values also affect the order in which multiple .bbappend files for the same recipe are applied.

To specify the layer’s priority manually, use the BBFILE\_PRIORITY variable and append the layer’s root name:

BBFILE\_PRIORITY\_mylayer = “1”

Note: the layer priority does not currently affect the precedence order of .conf or .bbclass files.

\*\*\* Use .bbappend for splash screen

Q: How to validate whether my bbappend is successfully applied to recipe?

$ bitbake-layers show-appends

Lists .bbappend files and the recipe files to which they apply.