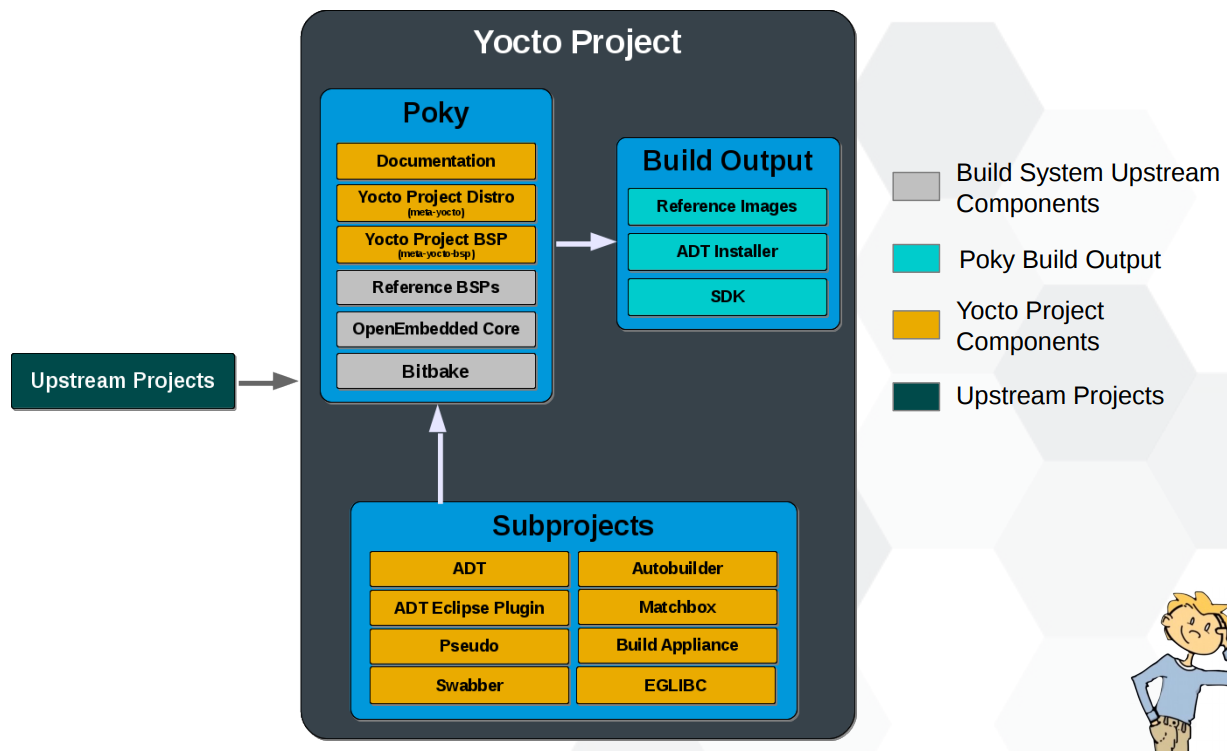
# Introduction

**Refs**:

* Official guideline: <https://docs.yoctoproject.org/>
* Slide: <https://elinux.org/images/a/a8/Getting_Started_with_Embedded_Linux-_Using_the_Yocto_Project_to_Build_your_Own_Custom_Embedded_Linux_Distribution.pdf>

## What Is Yocto?

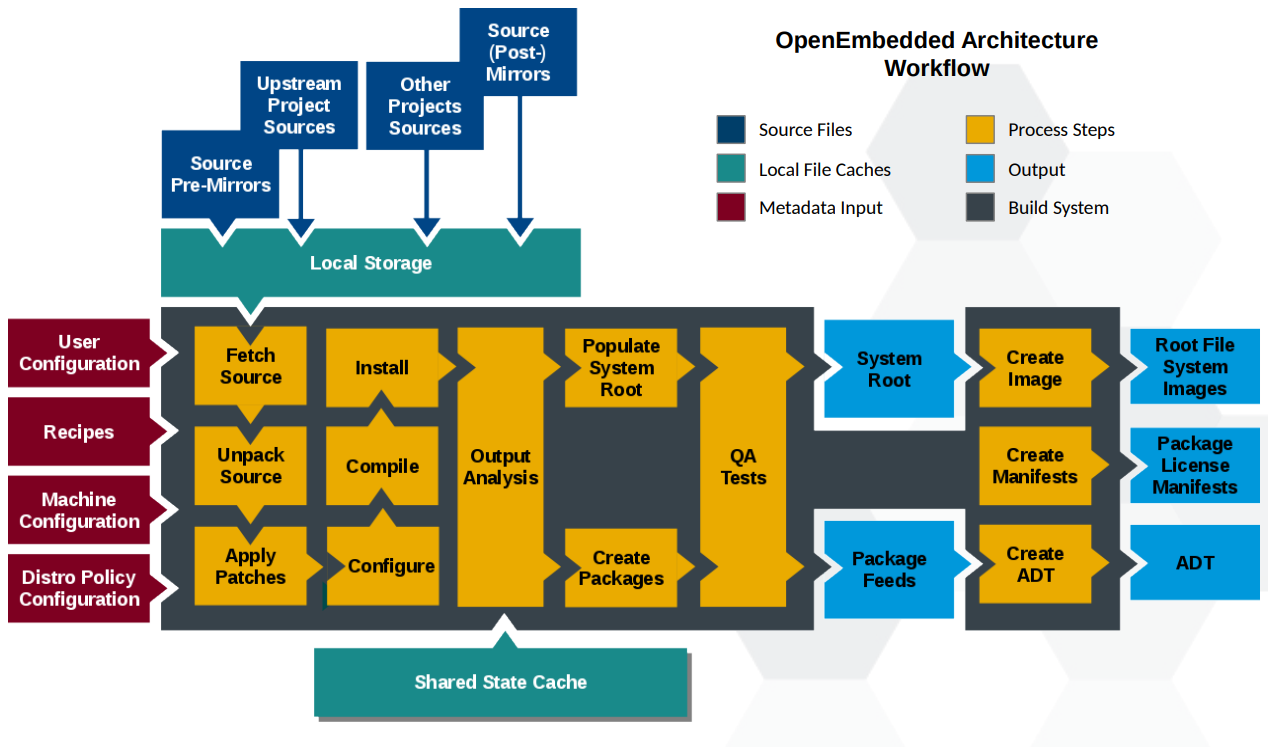
* Open-source project with a strong community
* A collection of embedded projects and tooling:
* Place for Industry to publish BSPs
* Application Development Tools including Eclipse plug-ins and emulators
* Key project is the reference distribution build environment (Poky) with:
  + Complete build system for Linux (OpenEmbedded and BitBake)



## Benefits

* Less time spent on things which don’t make money (build system, core Linux components)
* More time spent on things which do make money (app development, product development, …)
* Supports all major embedded architectures:
* x86, x86-64, ARM, PPC, MIPS
* Coming soon, MIPS64 and ARM Arch 64

## Workflow



**Explanation**:

Fetch source:

* Recipes call out the location of all sources, patches and files. These may exist on the internal or be local. (SRC\_URI in \*.bb files).
* BitBake can get the sources from git, svn, bzr, http, https, fpt, repo, ssh, etc.
* Versions of packages can be fixed or updated automatically (SRCREV\_pn-PN = "${AUTOREV}" in local.conf).

Unpack source:

* Once sources are obtained, they are extracted.
* The unpacker can cope with tarballs, zip, rar, xz, gz, bz2, etc.

Apply patches:

* Patches are applied in the order they appear in SRC\_URI.
* quilt is used to apply patches

Configure / Compile / Install:

* Recipe specifies configuration and compilation rules:
  + Various standard build rules are available, such as autotools and gettext.
  + Standard ways to specify custom environment flags.
  + Install step runs under ‘pseudo’, allows special files, permissions and owners/groups to be set.

Output analysis:

* Categorize generated software (debug, dev, docs, locales).
* Split runtime and debug information.

Create packages:

* Support popular formats: RPM, Debian, and ipk.
* Set preferred format using PACKAGE\_CLASSES in local.conf.
* Package files can be manually defined to override automatic settings.

QA tests:

* Perform sanity checks.
* Package after created and QA tested is put into Package Feed.

Create images:

* What to install on the image is based on the minimum defined set of required components in an image recipe. This minimum set is then expanded based on dependencies to produce a package solution.
* Image may be generated in a variety of formats (tar.bz2, ext2, ext3, jffs, etc.).
* Image after created are put into Root File System Image.

Create manifests:

* Manifest after created are put into Package License Manifest.

Create ADT:

* A specific SDK recipe may be created. This allows someone to build an SDK with specific interfaces in it. (i.e. meta-toolchain-gmae).
* SDK contains native applications, cross toolchain and installation scripts.
* May be used by the Eclipse Application Developer Tool to enable App Developers.
* May contain a QEMU target emulation to assist app developers.

# Concepts

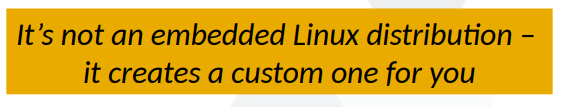
## Poky

Poky is a reference embedded distribution build environment. It contains:

* A **build system** (BitBake and OpenEmbedded Core) for Linux
* A **set of** **metadata** for building your own distro

It provides the following:

* A base-level functional distro used to **illustrate how to customize a distribution**.
* A means by which to **test the Yocto Project components** (i.e. Poky is used to validate the Yocto Project).
* A vehicle through which you can **download the Yocto Project**.



For a quick guide on how to build Poky, check <https://docs.yoctoproject.org/brief-yoctoprojectqs/index.html>

## Build Host

It's the **system used to build images**.

## BitBake

It's an build engine used by the build system to build images. It handles the **parsing and execution of metadata**, generating a list of tasks from it, and then executing those tasks.

For more details, check [this session](#_BitBake_1).

## OpenEmbedded Build System

It's the **build system specific to the Yocto Project**. It's based on another project known as Poky. We can call it "the build system" for short. If other build systems, such as a host or target build system are referenced, we should state the full name.

## OpenEmbedded-Core (OE-Core)

OE-Core is a shared base metadatathat is meant to be common among OpenEmbedded-derived systems.

## Metadata

It includes**:**

* **Configuration files**
* **Recipes**
* Data aboutbuild instructions
* Data used to control what things get built and the effects of the build
* Commands and data used to indicate what versions of software are used, from where they are obtained, and changes or additions to the software itself (patches or auxiliary files)

Metadata is contained in the files that the OpenEmbedded Build System parses when building an image.

For more details, check [this session](#_Metadata).

## Layer

It's a **collection of related metadata**. By convention, layer names start with meta-.

For more details, check [this session](#_Layer_1).

## Package

It's the **compiled binaries** produced from the recipe’s sources. You "bake" something by running it through BitBake.

It's worth noting that the term "package" can have other meanings.

## Directory

### Source Directory

It refers to the directory by creating a local copy of the poky Git repo (git://git.yoctoproject.org/poky) or expanding a released poky tarball.

**Note**

**The OpenEmbedded build system does not support file or directory names that contain spaces**.

The source directory contains:

* BitBake
* Documentation
* Metadata
* Other files that all support the Yocto Project.

### Build Directory

It refers to the **area used by the build system** for builds.

The area is **created when you source the setup environment script** that is found in the source Directory (i.e. [oe-init-build-env](https://docs.yoctoproject.org/ref-manual/structure.html#oe-init-build-env)). The TOPDIR variable points to the build directory.

You have some ways to create the build directory. The examples assume your source directory is named poky:

* Create the build directory inside the source directory, and name it as build (default):

$ cd poky

$ source oe-init-build-env

* Create the build directory inside your home directory, and name it as test-builds:

$ source poky/oe-init-build-env test-builds

### Sysroot

When cross-compiling, the target file system may be differently laid out and contain different things compared to the host system. The concept of a sysroot is **directory which looks like the target filesystem and can be used to cross-compile against**.

In the context of cross-compiling toolchains, a sysroot typically contains:

* C library
* compiled binaries for the C library
* kernel headers

In the context of the OpenEmbedded build System and of the Yocto Project, each recipe has two sysroots:

* A target sysroot contains all the **target** libraries and headers needed to build the recipe.
* A native sysroot contains all the **host** files and executables needed to build the recipe.

The SYSROOT\_\* variables controll how sysroots are created and stored.

### Package Feeds

Directories containing output packages (RPM, DEB or IPK), which are subsequently used in the construction of an image or SDK, produced by the build system.

<https://docs.yoctoproject.org/overview-manual/concepts.html#package-feeds>

## Image

It's an artifact of the BitBake build process given a collection of recipes and related Metadata. Images are the **binary output that run on specific hardware or QEMU** and are used for specific use-cases.

For a list of the supported image types that the Yocto Project provides, see the [Images](https://docs.yoctoproject.org/ref-manual/images.html#images) chapter.

## Cross-Development Toolchain

It's a **collection of software development tools and utilities** that run on one architecture and allow you to develop software for a different targeted architecture.

It contains cross-compilers, linkers, and debuggers that are specific to the target architecture.

The Yocto Project supports two cross-development toolchains:

* A toolchain only used by and within BitBake when building an image for a target architecture.
* A relocatable toolchain used outside of BitBake by developers when developing applications that will run on a targeted device.

**Refs**:

Creation of these toolchains is simple and automated. For information on toolchain concepts as they apply to the Yocto Project, see the [Cross-Development Toolchain Generation](https://docs.yoctoproject.org/overview-manual/concepts.html#cross-development-toolchain-generation) section in the Yocto Project Overview and Concepts Manual.

You can also find more information on using the relocatable toolchain in the [Yocto Project Application Development and the Extensible Software Development Kit (eSDK)](https://docs.yoctoproject.org/sdk-manual/index.html) manual.

## Application Development Toolkit (ADT)

Development environment for user-space applications to run on OS stacks built by Poky .

## Eclipse IDE Plugin

Integration of ADT into the Eclipse IDE.

## Matchbox

X Windows-based open source graphical UI for embedded devices.

## Autobuilder

Automation for Yocto Project build tests and QA .

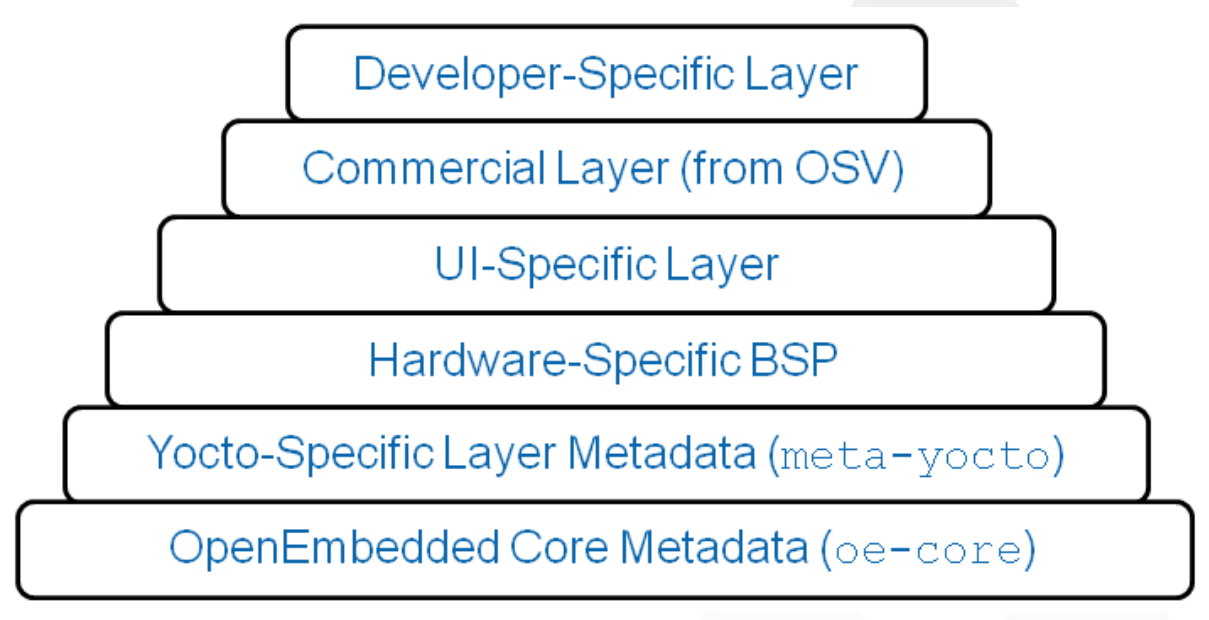
## Swabber

Host leakage detection tool.

# Layer

## What Is Layer?

The build system is composed of layers:



## Notes

### Do not put everything into one layer

Use different layers to **logically separate information in your build**. In other words, they should be grouped by functionality:

* Custom toolchains (compilers, debuggers, profiling tools)
* Distribution specifications (i.e. meta-yocto)
* BSP/Machine settings (i.e. meta-yocto-bsp)
* Functional areas (selinux, networking, etc)
* Project specific changes
* Application (e.g. meta-filesystems, meta-python)

### Never modify the POKY layer

When you update to the next release, you’ll lose all of your work. ALL OF IT.

### Use existing BSP layers when possible

Intel, TI, NXP and others have information on what BSP layers to use with their silicon. These layers have names such as meta-intel, meta-ti, etc.

**Refs**:

For introductory information on layers, see [The Yocto Project Layer Model](https://docs.yoctoproject.org/overview-manual/yp-intro.html#the-yocto-project-layer-model) section in the Yocto Project Overview and Concepts Manual.

For more detailed information on layers, see the [Understanding and Creating Layers](https://docs.yoctoproject.org/dev-manual/layers.html#understanding-and-creating-layers) section in the Yocto Project Development Tasks Manual.

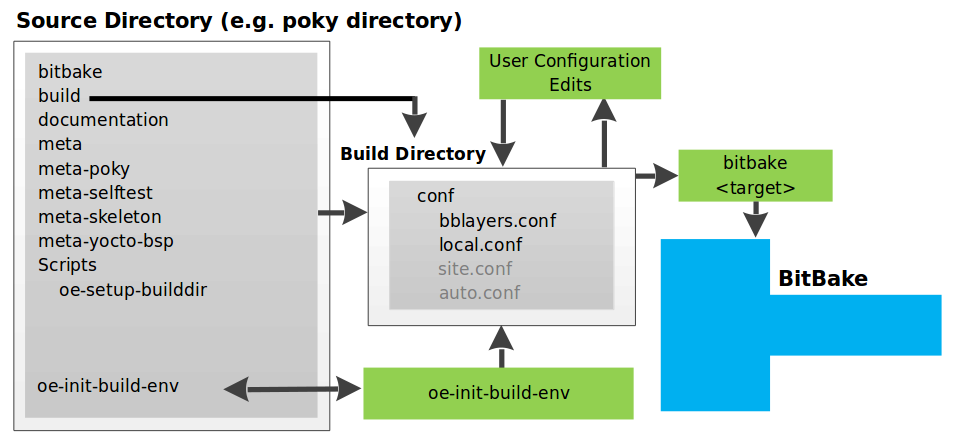
For a discussion specifically on BSP Layers, see the [BSP Layers](https://docs.yoctoproject.org/bsp-guide/bsp.html#bsp-layers) section in the Yocto Project Board Support Packages (BSP) Developer’s Guide.

<https://docs.yoctoproject.org/overview-manual/concepts.html#metadata-machine-configuration-and-policy-configuration>

# Metadata

Where to get metadata sample? <https://git.yoctoproject.org/>

## Configuration



### Files

Configuration files have .conf extension. They tell the build system **what to build** and **what to put into the image** to support a particular platform.

Following is the list of essential configuration files:

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **File** | **Description** | **Notes** |
| Layer | build/conf/bblayers.conf | Specify what layers to be considered during the build |  |
| \*/conf/layer.conf  (one per layer) |  | * Add Yocto Project Compatible layers to the BBLAYERS   Default: meta (oe-core), meta-yocto, meta-yocto-bsp |
| BitBake | meta/conf/bitbake.conf | defaults |  |
| User | build/conf/local.conf | Override and define what you are building.  Located in the Build Directory. | * BB\_NUMBER\_THREADS * PARALLEL\_MAKE * MACHINE settings * DISTRO settings * INCOMPATIBLE\_LICENSE = "GPLv3" * EXTRA\_IMAGE\_FEATURES |
| Distribution policy | meta-yocto/conf/  distro/poky.conf | Defines system-wide policies that affect the way individual recipes are built:   * Set alternative preferred versions of recipes * Enable/disable LIBC functionality (i.e. i18n) * Enable/disable features (i.e. pam, selinux) * Configure specific package rules * Adjust image deployment settings | * Enabled via the DISTRO setting * Four predefined settings * poky-bleeding: Enable a bleeding edge packages * poky: Core distribution definition, defines the base * poky-lsb: enable items required for LSB support * poky-tiny: construct a smaller then normal system |
| Machine | meta-yocto-bsp/conf/  machine/xxx.conf  E.g. beagleboard.conf (Texas Instruments ARM Cortex-A8 development board) | Define variables for specific hardware and are only used when building for that target:   * Board specific kernel configuration * Formfactor configurations * Processor/SOC Tuning files   Located throughout the Source Directory. | * Hardware machines and emulated machines (QEMU) * Machine configuration refers to kernel sources and may influence some userspace software |
|  | conf/site.conf | Configure multiple build directories | This file is not created by default, so you need to create it yourself. |
|  | conf/auto.conf |  | This file is typically created by an autobuilder.  OpenEmbedded Build System reads the configuration files in a specific order: site.conf, auto.conf, and local.conf. |

### Variables

<https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-ref-variables.html#variables-glossary>

|  |  |
| --- | --- |
| **Option** | **Description** |
| **build/conf/local.conf** | |
| BB\_NUMBER\_THREADS | Execute in parallel.  Tips: To know how many processor in your build host run:  $ grep processor /proc/cpuinfo  Or:  $ ncpus # Require "$ sudo apt install mdm" |
|  |  |
| build/conf/bblayers.conf | |
| BBLAYERS | Add layers |
|  |  |
| \*/conf/layer.conf | |
| BBPATH | Search for configuration and class files under the conf and classes directories |
| BBFILES | Locate recipe files (.bb) and recipe append files (.bbappend) |
| LAYERDIR |  |
|  |  |
| meta/conf/bitbake.conf | |
| BBPATH |  |
|  |  |
|  |  |
|  |  |
|  |  |
| SRC\_URI | Specify path to source files.  Each recipe must have a SRC\_URI. |

## Recipes

Recipe files have .bb extension. They contain a **set of instructions for building packages**. They describe:

* Where to get source code and how to fetch it
* Which patches to apply to source code, where to find them and how to apply them
* How to configure and compile source code
* Dependencies for libraries or for other recipes
* Descriptive information about the package (author, homepage, license, and so on)
* Etc.

## Classes

Class files have .bbclass extension. They provide commonly used patterns which are defined once but then easily used (inherited) in multiple recipes.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **File** | **Description** | **Notes** |
| Base class | classes/  base.bbclass | Contains definitions for standard basic tasks:   * Fetching * Unpacking * Configuring (empty by default) * Compiling (runs any Makefile present) * Installing (empty by default) * Packaging (empty by default)   These tasks are often overridden or extended by other classes added during the project development process. | Always included automatically for all recipes and classes |

## Append Files

Append files have .bbappend extension. They **extend or override information in an existing recipe file**.

The append files and corresponding recipe files **must use the same root filename**. The filenames can differ only in the file type suffix used (e.g. formfactor\_0.0.bb and formfactor\_0.0.bbappend).

When you name an append file, you can use the % wildcard character to allow for matching recipe names. For example, the append file busybox\_1.21.%.bbappend would match any busybox\_1.21.x.bb recipe. Note that the % character only works directly in front of the .bbappend portion of the append file’s name. You cannot use it in any other portion of the name.

## Underlying Include Files

They have .inc extension.

# BitBake

[https://docs.yoctoproject.org/BitBake/2.4/BitBake-user-manual/BitBake-user-manual-intro.html](https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-intro.html)

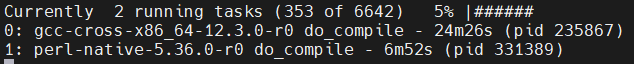
## What Is BitBake?

Check [this session](#_Bitbake).

Conceptually, BitBake is similar to GNU Make. But has some differences:

* BitBake executes tasks according to the provided metadata.
* BitBake includes a fetcher library for obtaining source code from various places such as local files, source control systems, or websites.
* BitBake controls how software is built. It achieves this through "recipes", while GNU Make uses "makefiles".
* BitBake extends the capabilities of GNU Make by allowing much more complex tasks, such as assembling entire embedded Linux distributions.
* BitBake is written in Python.

Notice when you compile source code, you see something like that. This is when Bibake execute tasks.



## Where To Get BitBake?

$ git clone git://git.openembedded.org/bitbake

## Commands

### Usage

Usage: bitbake [options] [recipename/target recipe:do\_task ...]

Options:

|  |  |
| --- | --- |
| **Option** | **Description** |
| --version | Show program's version number |
| -h, --help | Show help message |
| -b BUILDFILE,  --buildfile=BUILDFILE | **Execute tasks** from a specific .bb recipe directly.  WARNING: Does not handle any dependencies from other recipes 🡺 Should not be used. |
| -k, --continue | Continue building as much as possible after an error. |
| -f, --force | Force the specified targets/task to run (invalidating any existing stamp file). |
| -c CMD,  --cmd=CMD | **Specify the task** to execute.  The exact CMD depend on the metadata. Some examples might be 'compile' or 'populate\_sysroot' or 'listtasks' may give a list of the tasks available. |
| -C INVALIDATE\_STAMP,  --clear-stamp=INVALIDATE\_STAMP | Invalidate the stamp for the specified task such as 'compile' and then run the default task for the specified target(s). |
| -r PREFILE,  --read=PREFILE | Read the specified file before BitBake.conf. |
| -R POSTFILE,  --postread=POSTFILE | Read the specified file after BitBake.conf. |
| -v, --verbose | Enable tracing of shell tasks (with 'set -x').  Also print bb.note(...) messages to stdout. |
| -D, --debug | Print debug logs. You can also set debug level:   * -D sets the debug level to 1, where only bb.debug(1, ...) messages are printed to stdout. * -DD sets the debug level to 2, where both bb.debug(1, ...) and bb.debug(2, ...) messages are printed * Etc.   Note: -D only affects output to stdout. All debug messages are written to ${T}/log.do\_taskname, regardless of the debug level. |
| -q, --quiet | Output less log data to the terminal. You can specify this more than once. |
| -n, --dry-run | **Don't execute**, just go through the motions. |
| -S SIGNATURE\_HANDLER,  --dump-signatures=SIGNATURE\_HANDLER | Dump out the signature construction information, with no task execution.  The SIGNATURE\_HANDLER param is passed to the handler.  Common values are:   * none (only dump the signature) * printdiff (compare the dumped signature with the cached one) * more * less |
| -p, --parse-only | Quit after parsing the BB recipes. |
| -s, --show-versions | Show current and preferred versions of all recipes. |
| -e, --environment | Show the global or per-recipe environment complete with information about where variables were set/changed. |
| -g, --graphviz | Save dependency tree information for the specified targets in the dot syntax. |
| -I EXTRA\_ASSUME\_PROVIDED,  --ignore-deps=EXTRA\_ASSUME\_PROVIDED | Assume these dependencies don't exist and are already provided (equivalent to ASSUME\_PROVIDED).  Useful to make dependency graphs more appealing |
| -l DEBUG\_DOMAINS,  --log-domains=DEBUG\_DOMAINS | Show debug logging for the specified logging domains |
| -P, --profile | Profile the command and save reports. |
| -u UI, --ui=UI | The user interface to use (knotty, ncurses, taskexp or teamcity - default knotty). |
| --token=XMLRPCTOKEN | Specify the connection token to be used when connecting to a remote server. |
| --revisions-changed | Set the exit code depending on whether upstream floating revisions have changed or not. |
| --server-only | Run BitBake without a UI, only starting a server (cooker) process. |
| -B BIND, --bind=BIND | The name/address for the BitBake xmlrpc server to bind to. |
| -T SERVER\_TIMEOUT,  --idle-timeout=SERVER\_TIMEOUT | Set timeout to unload BitBake server due to inactivity, set to -1 means no unload, default: Environment variable BB\_SERVER\_TIMEOUT. |
| --no-setscene | Do not run any setscene tasks. sstate will be ignored and everything needed, built. |
| --skip-setscene | Skip setscene tasks if they would be executed. Tasks previously restored from sstate will be kept, unlike --no-setscene |
| --setscene-only | Only run setscene tasks, don't run any real tasks. |
| --remote-server=REMOTE\_SERVER | Connect to the specified server. |
| -m, --kill-server | Terminate any running BitBake server. |
| --observe-only | Connect to a server as an observing-only client. |
| --status-only | Check the status of the remote BitBake server. |
| -w WRITEEVENTLOG,  --write-log=WRITEEVENTLOG | Writes the event log of the build to a BitBake event json file.  Use '' (empty string) to assign the name automatically. |
| --runall=RUNALL | Run the specified task for any recipe in the taskgraph of the specified target (even if it wouldn't otherwise have run). |
| --runonly=RUNONLY | Run only the specified task within the taskgraph of the specified targets (and any task dependencies those tasks may have). |

### Examples

#### Execute Packages

**1.** BitBake, when not using -b, only accepts a PROVIDES. You cannot provide anything else. By default, a recipe file generally PROVIDES its packagename.

The following command executes package foo:

$ bitbake foo

The following command executes package foo, and also just execute the do\_clean task:

$ bitbake -c clean foo

**2.** Sometimes, when dealing with multiple targets, you want to **specify different tasks for different targets**.

The following command runs taskA for myfirstrecipe and taskB for mysecondrecipe:

$ bitbake myfirstrecipe:do\_taskA mysecondrecipe:do\_taskB

#### Generate Dependency Graph

BitBake can generate dependency graphs using the dot syntax which can be converted into graph using the dot tool from Graphviz.

BitBake writes two files to the current directory:

* task-depends.dot: Shows dependencies between tasks. These match BitBake’s internal task execution list.
* pn-buildlist: Shows a simple list of targets that are to be built.

You can omit common depends from the graph (to have a more readable graph) by using the -I option. This way, you can remove from the graph DEPENDS from inherited classes such as base.bbclass.

The following command generates a dependency graph for package foo:

$ bitbake -g foo

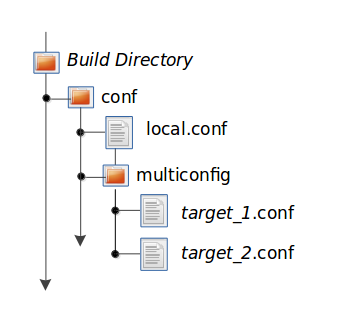
The following comand generates a dependency graph for package foo, but it omits depends common in OpenEmbedded from the graph:

$ bitbake -g -I virtual/kernel -I eglibc foo

#### Separate Configuration Files for Each Target

BitBake can **build multiple images or packages using a single command** where the different targets require different configurations (multiple configuration builds). Each target, in this scenario, is referred to as a "multiconfig".

Following is an example for two separate targets – target1 and target2:



In local.conf, enable multiple configuration builds and specifies the two extra multiconfigs:

BBMULTICONFIG = "target1 target2"

Start the builds with following syntax:

$ bitbake [mc:multiconfigname:]target [[[mc:multiconfigname:]target] ... ]

Here is an example for two extra multiconfigs – target1 and target2:

$ bitbake mc::target mc:target1:target mc:target2:target

#### Hello World

<https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-hello.html>

## Syntax

BitBake files have their own syntax.

<https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-metadata.html#syntax-and-operators>

## Steps

Prior to parsing configuration files, BitBake looks at certain variables, including:

BB\_ENV\_PASSTHROUGH

BB\_ENV\_PASSTHROUGH\_ADDITIONS

BB\_PRESERVE\_ENV

BB\_ORIGENV

BITBAKE\_UI

0. source oe-init-build-env, which is the build environment script.

Once sourcing is completed, a Build Directory (named build) is created (if not already exist). BitBake uses this directory for all its work during builds.

1. BitBake parses conf/blayers.conf file the current directory (BBLAYERS), conf/layer.conf files (BBPATH, BBFILES, LAYERDIR), and conf/bitbake.conf file.

2. BitBake parses class files in a classes subdirectory under the paths in BBPATH. The base.bbclass file is always included. Other classes that are specified in the configuration using the INHERIT variable are also included.

Tip: You can get paths of configuration files and class files in your execution environment by running:

$ bitbake -e > mybb.log

3. BitBake parses recipes file and append files.

Once all recipes are completely parsed, BitBake has a list of tasks, a set of data consisting of keys and values, and dependency info about the tasks.

4. BitBake figures out how to build the target

a) First, BitBake looks through each recipe’s PROVIDES list.

Each list is created implicitly through the PN variable and explicitly through the PROVIDES variable. When a recipe uses PROVIDES, that recipe’s functionality can be found under an alternative name or names other than the implicit PN name.

For example, suppose a recipe named keyboard\_1.0.bb contained the following:

PROVIDES += "fullkeyboard"

The PROVIDES list for this recipe becomes keyboard which is implicit, and fullkeyboard which is explicit. Consequently, the functionality found in keyboard\_1.0.bb can be found under two different names.

b) Then, BitBake needs to prioritize providers by determining provider preferences (because targets might have multiple providers)

<https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-execution.html#preferences>

c) Then, BitBake calculates dependencies

Each target consists of multiple tasks such as fetch, unpack, patch, configure, and compile. BitBake considers each task as an independent entity with its own set of dependencies.

Dependencies are defined through several variables. At a basic level, it's enough to know that BitBake uses the DEPENDS and RDEPENDS variables when calculating dependencies.

For more information on how BitBake handles dependencies, see the [Dependencies](https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-metadata.html#dependencies) section.

5. BitBake prepares to execute tasks

Based on provider list and dependency information, BitBake can now calculate exactly what tasks it needs to run and in what order. The [Executing Tasks](https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-execution.html#executing-tasks) section has more information on how BitBake chooses which task to execute next.

a) First, BitBake forking off threads up to the limit set in the  BB\_NUMBER\_THREADS variable.

b) As each task completes, a timestamp is written to the directory specified by the STAMP variable.

On subsequent runs, BitBake looks in the build directory within tmp/stamps and does not rerun tasks that are already completed unless a timestamp is found to be invalid. Currently, invalid timestamps are only considered on a per recipe file basis. So, for example, if the configure stamp has a timestamp greater than the compile timestamp for a given target, then the compile task would rerun. Running the compile task again, however, has no effect on other providers that depend on that target.

6. BitBake executes tasks

Tasks can be either a shell task or a Python task.

For shell tasks, BitBake writes a shell script to ${[T](https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-ref-variables.html#term-T)}/run.do\_taskname.pid and then executes the script. The shell script contains all exported variables and functions. Output from the script goes to the file ${[T](https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-ref-variables.html#term-T)}/log.do\_taskname.pid.

For Python tasks, BitBake executes the task internally and logs information to the controlling terminal. Future versions of BitBake will write the functions to files similar to the way shell tasks are handled. Logging will be handled in a way similar to shell tasks as well.

The order in which BitBake runs the tasks is controlled by its task scheduler. It is possible to configure the scheduler and define custom implementations for specific use cases with the [BB\_SCHEDULER](https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-ref-variables.html#term-BB_SCHEDULER) and [BB\_SCHEDULERS](https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-ref-variables.html#term-BB_SCHEDULERS) variables.

## Checksums (Signatures)

<https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-execution.html#checksums-signatures>

## Setscene

<https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-execution.html#setscene>

## Logging

<https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-execution.html#logging>

# Beagle Bone Black

# CROPS

When using Yocto, you're assumed to use a Build Host which is a native Linux system running a recent Ubuntu Linux distribution. But if it's not a native Linux system, you can still perform these steps by using **CROss PlatformS (CROPS)** and setting up a Poky container.

Details: <https://docs.yoctoproject.org/dev-manual/start.html#setting-up-to-use-cross-platforms-crops>

# Dependency Graph

**Know that you can generate a dependency graph and learn how to do it:** A dependency graph shows dependencies between recipes, tasks, and targets. You can use the “-g” option with BitBake to generate this graph. When you start a build and the build breaks, you could see packages you have no clue about or have any idea why the build system has included them. The dependency graph can clarify that confusion. You can learn more about dependency graphs and how to generate them in the [Generating Dependency Graphs](https://docs.yoctoproject.org/bitbake/2.4/bitbake-user-manual/bitbake-user-manual-intro.html#generating-dependency-graphs) section in the BitBake User Manual.

# Hob – Graphical User Interface for BitBake

# TODO

1. **Depending on what you primary interests are with the Yocto Project, you could consider any of the following reading:**
   * **Look Through the Yocto Project Development Tasks Manual**: This manual contains procedural information grouped to help you get set up, work with layers, customize images, write new recipes, work with libraries, and use QEMU. The information is task-based and spans the breadth of the Yocto Project. See the [Yocto Project Development Tasks Manual](https://docs.yoctoproject.org/dev-manual/index.html).
   * **Look Through the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual**: This manual describes how to use both the standard SDK and the extensible SDK, which are used primarily for application development. The [Using the Extensible SDK](https://docs.yoctoproject.org/sdk-manual/extensible.html) also provides example workflows that use devtool. See the section [Using devtool in Your SDK Workflow](https://docs.yoctoproject.org/sdk-manual/extensible.html#using-devtool-in-your-sdk-workflow) for more information.
   * **Learn About Kernel Development**: If you want to see how to work with the kernel and understand Yocto Linux kernels, see the [Yocto Project Linux Kernel Development Manual](https://docs.yoctoproject.org/kernel-dev/index.html). This manual provides information on how to patch the kernel, modify kernel recipes, and configure the kernel.
   * **Learn About Board Support Packages (BSPs)**: If you want to learn about BSPs, see the [Yocto Project Board Support Package Developer’s Guide](https://docs.yoctoproject.org/bsp-guide/index.html). This manual also provides an example BSP creation workflow. See the [Board Support Packages (BSP) — Developer’s Guide](https://docs.yoctoproject.org/bsp-guide/bsp.html) section.
   * **Learn About Toaster**: Toaster is a web interface to the Yocto Project’s OpenEmbedded build system. If you are interested in using this type of interface to create images, see the [Toaster User Manual](https://docs.yoctoproject.org/toaster-manual/index.html).
   * **Have Available the Yocto Project Reference Manual**: Unlike the rest of the Yocto Project manual set, this manual is comprised of material suited for reference rather than procedures. You can get build details, a closer look at how the pieces of the Yocto Project development environment work together, information on various technical details, guidance on migrating to a newer Yocto Project release, reference material on the directory structure, classes, and tasks. The [Yocto Project Reference Manual](https://docs.yoctoproject.org/ref-manual/index.html) also contains a fairly comprehensive glossary of variables used within the Yocto Project.

After source:

