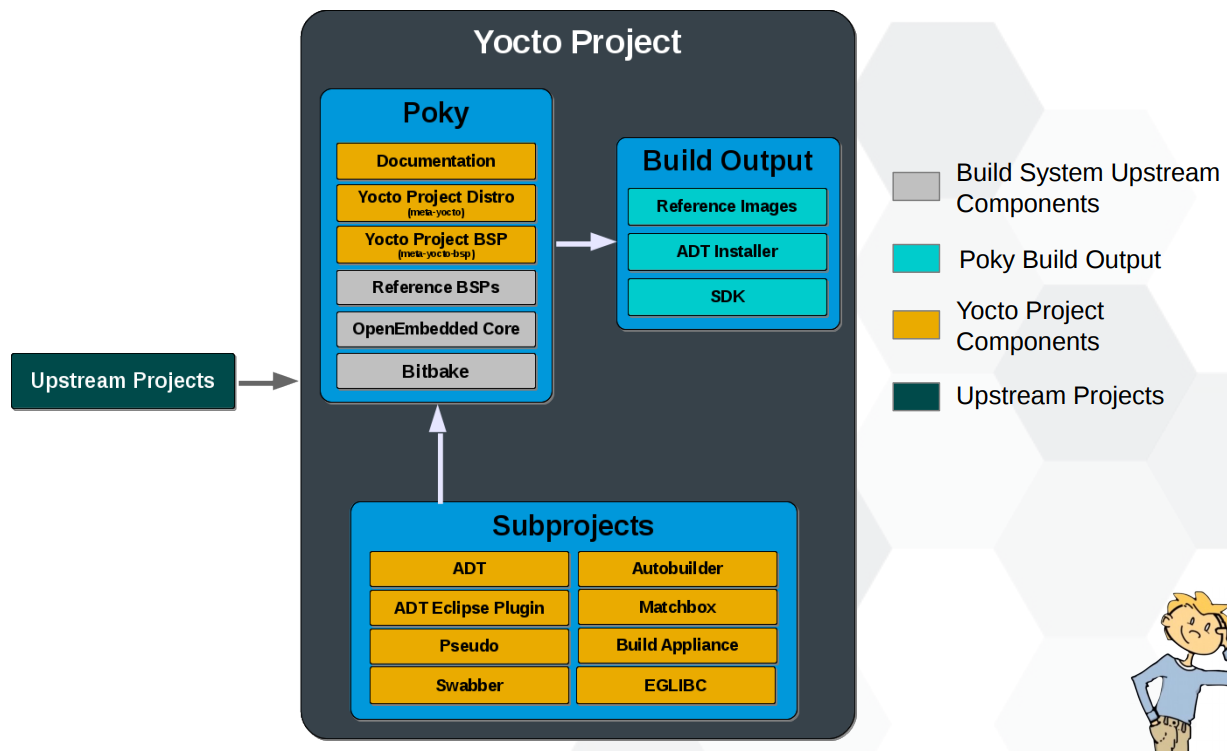
# Introduction

## What Is Yocto?

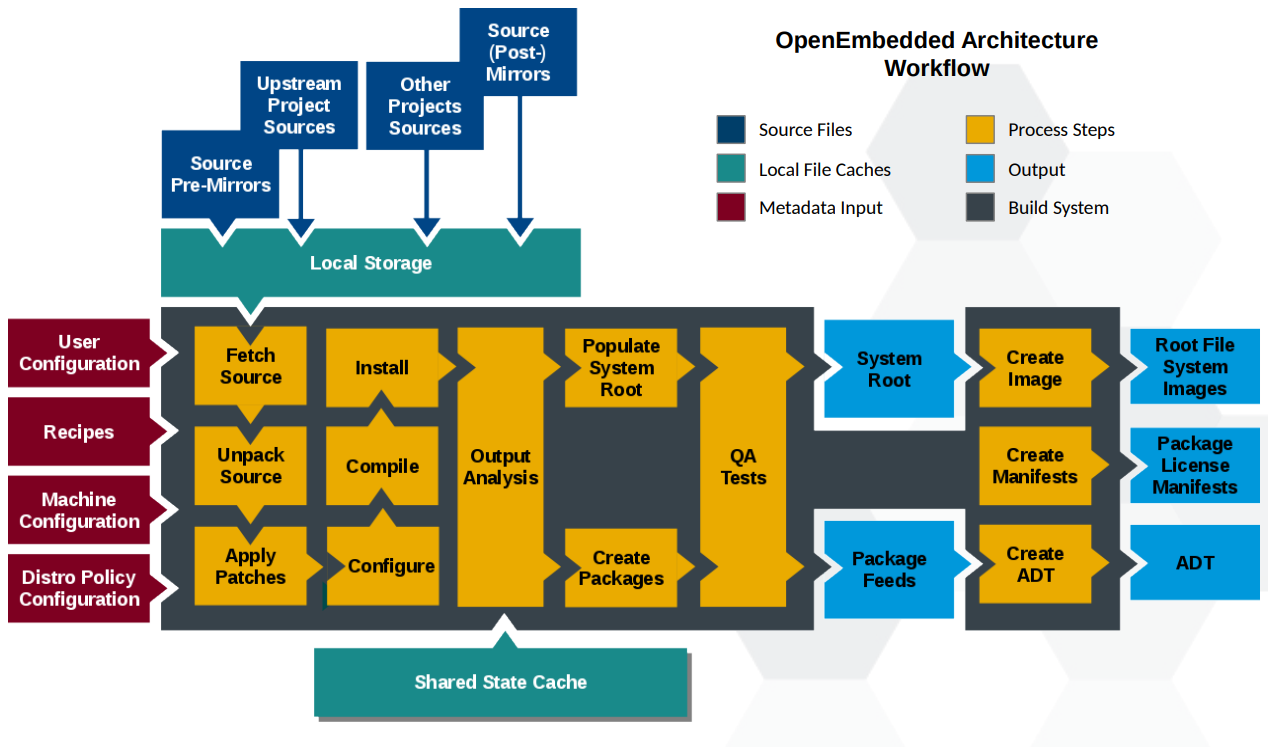
* Open-source project with a strong community
* A collection of embedded projects and tooling:
* Place for Industry to publish BSPs
* Application Development Tools including Eclipse plug-ins and emulators
* Key project is the reference distribution build environment (Poky) with:
  + Complete build system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

# Installation

## Hardware Preparation

It's hard to know how much disk space or RAM you'll need for a Yocto project because this mostly depends on what you'll build.

One of the key factors that affects the hardware requirement is which embedded linux build system you choose for your project. Here are common build systems:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **What Is It** | **Pros** | **Cons** |
| **Poky** | The reference distribution of the Yocto Project. | Highly customizable; extensive community support; rich ecosystem. | Can be resource-intensive; complex setup for beginners. |
| **Buildroot** | A simple, efficient tool to generate embedded Linux systems. | Lightweight; fast builds; easy to configure. | Less flexible than Yocto; fewer features and packages. |
| **OpenEmbedded** | A build framework for embedded Linux; Poky is a part of it. | Supports a wide range of architectures; modular design. | More complex than some alternatives; requires learning curve. |
| **Alpine Linux** | A security-oriented, lightweight Linux distribution. | Very small footprint; fast; minimalistic approach. | Limited package availability; not as flexible for embedded as Yocto. |
| **Fedora IoT** | A variant of the Fedora Project tailored for IoT devices. | Up-to-date packages; good community support. | Larger base image; may be overkill for minimal applications. |
| **Raspberry Pi OS** | Debian-based OS optimized for Raspberry Pi devices. | Easy to use; great community support; ready-made images. | Limited to Raspberry Pi hardware; not as customizable as Yocto. |
| **Yocto Project (without Poky)** | Custom Yocto setups using layers and recipes. | Highly customizable; tailored to specific needs. | Requires significant setup and maintenance; can be complex. |
| **OpenWrt** | A Linux distribution for embedded devices, primarily routers. | Lightweight; good for networking applications. | Limited general-purpose applications; focused on networking. |
| **Tiny Core Linux** | A minimal Linux distribution focused on being small and efficient. | Extremely lightweight; modular design; fast boot times. | Not as user-friendly; limited support and packages. |

## Software Preparation

Here’s a list of the key tools and dependencies typically required to set up a minimal Yocto environment:

|  |  |
| --- | --- |
| **Tool** | **Description** |
| **Linux OS** | Yocto is designed for Linux.  Note: If you have to run Yocto on Windows OS, consider using a virtual machine, or WSL, or Docker. |
| **Git** | Version control for cloning Yocto repositories. |
| **Python** | Required for running BitBake and various scripts.  Recommended: Python 3.x. |
| **GCC** | C/C++ compiler used for building packages. |
| **Make** | Build automation tool often used with BitBake. |
| **Diffutils** | Utilities for comparing files and directories. |
| **Coreutils** | Basic file, shell, and text manipulation utilities. |
| **Sed** | Stream editor for modifying files and scripts. |
| **Awk** | Language for pattern scanning and processing. |
| **Wget/Curl** | Utilities for downloading files from the web. |
| **Tar** | Used for extracting source packages and archives. |
| **Bash** | Shell that runs many scripts in the Yocto environment. |
| **CMake** | Optional, required for projects using CMake for configuration. |

Ubuntu: $ sudo apt-get install git python3 gcc g++ make diffutils coreutils sed awk wget tar bash

# Glossary

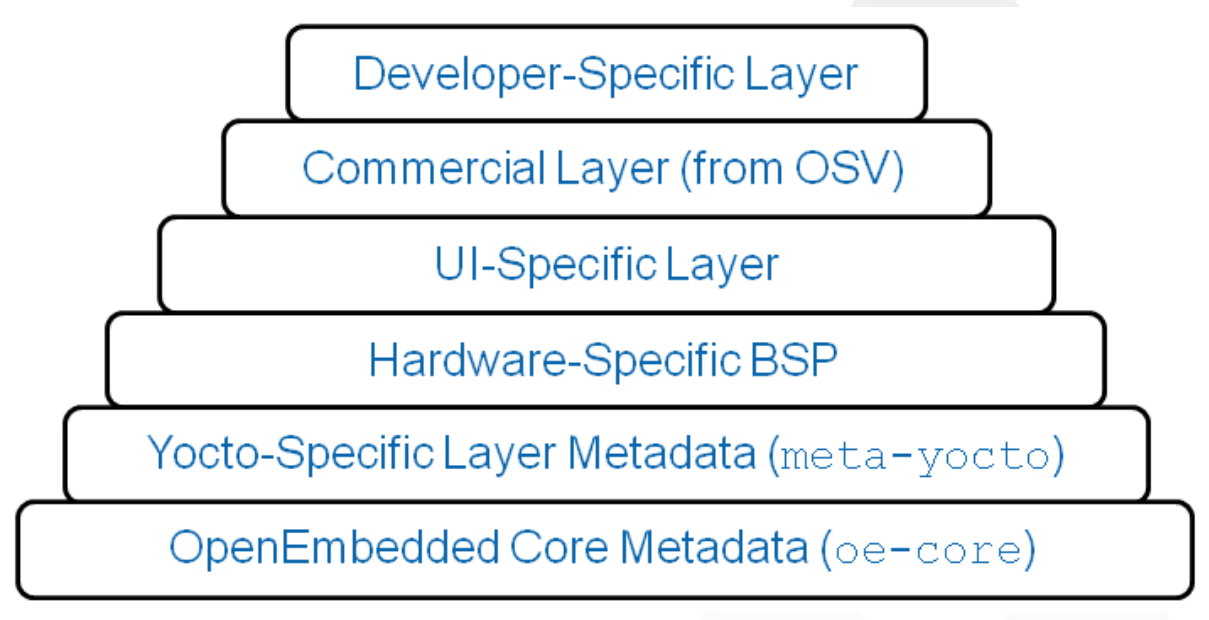
|  |  |
| --- | --- |
| **Concept** | **Description** |
|  |  |
| **BitBake** | It's a **build engine** used by the build system to build images. |
| **OpenEmbedded** | It's a **build system** which runs on a build engine. |
| **Recipe** | **Files that define how to build** software packages. Recipes specify the source code, dependencies, build instructions, and metadata.  They are written in a domain-specific language based on Python. They have .bb extension. |
| **Class** | Files that provide commonly used patterns which are defined once but then easily used (inherited) in multiple recipes.  They have .bbclass extension. |
| **Layer** | **Collections of** **related recipes**, configuration files, dependencies, and other metadata. By convention, layer names start with meta-.  They're actually folders. |
| **Configuration files** | Files that define **build settings** and specify which layers to include in the build.  They control various aspects of the build, such as target architecture, package selection, and build options. |
| **Metadata** | A general term which includes recipes, configuration files, etc. |
| **OpenEmbedded-Core** | OE-Core is a **shared base metadata** that is meant to be common among OpenEmbedded-derived systems.  It's considered as a low-level layer of Yocto. |
| **Poky** | It's a Reference Embedded Distribution Build Environment which contains the BitBake build engine, OpenEmbedded build system, recipes, layers, configruation files, toolchain (compilers, linkers, and libraries), etc. Together they help create custom Linux distributions specifically for embedded systems. |
| **Build host** | It's the **system used to build images**. |
| **Package** | It's the **compiled binaries** produced from the recipe’s sources. You "bake" something by running it through BitBake. |
| **Source directory** | A directory containing metadata (recipes, configuration files), source code, documents, etc.  Note: The build system does not support file or directory names with spaces. |
| **Build directory** | A directory where the build process takes place. It is created when you run the bitbake command.  It contains the output of the build process, including compiled binaries, images, and logs. |
| **Sysroot** | A directory looking like the target filesystem and can be used to cross-compile against.  It includes libraries, headers, and other files needed to compile applications for the target platform.  The SYSROOT\_\* variables controll how sysroots are created and stored. |
| **Package feed** | A directory containing pre-built packages (RPM, DEB or IPK) that can be used for deployment or installation on target devices. |
| **Image** | A binary output that run on specific hardware or QEMU. It's an artifact of the BitBake build process. |
| **ADT** | Application Development Toolkit.  Development environment for user-space applications to run on OS stacks built by Poky . |
| **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?

A layer is a collection of related recipes, configuration files, dependencies, and other metadata. By convention, its name start with meta-.

A complete build system is composed of multiple layers:



In a simple way, a layer is similar to a *folder* that groups related components and files together.

## Best Practices

### 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 (i.e. meta-toolchains)
* Distribution specifications (i.e. meta-yocto)
* BSP/Machine settings (i.e. meta-yocto-bsp)
* Functional areas (i.e. meta-security, meta-networking)
* 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 in meta-poky. 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.

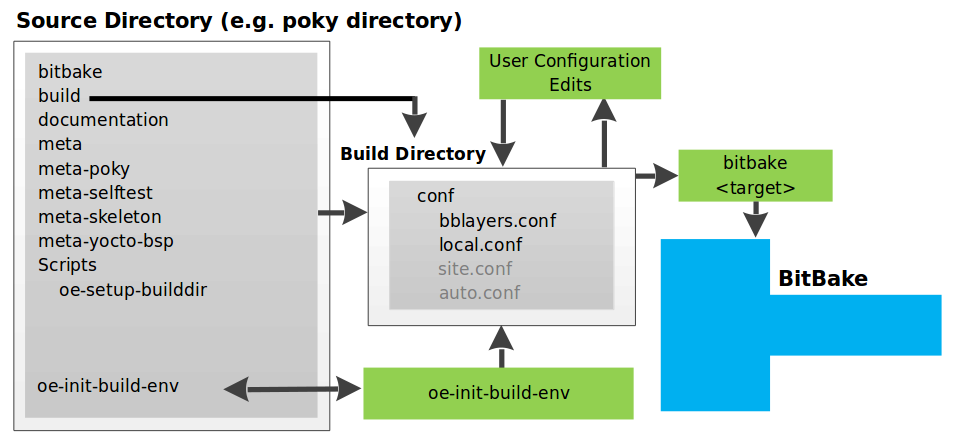
For a discussion specifically on BSP Layers, see the [BSP Layers](https://docs.yoctoproject.org/bsp-guide/bsp.html#bsp-layers) section in the Yocto Project Board Support Packages (BSP) Developer’s Guide.

<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. * It lists the paths to the layers that should be available during the build process. |  |
| \*/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. * It contains various parameters such as the target machine, package management options, and customization settings. * It's 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/\*.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/\*.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 in 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. |

|  |  |  |  |
| --- | --- | --- | --- |
| **Group** | **Variable** | **Description** | **Example** |
| Basic | PV | The version of the recipe, used for versioning and can include revision information. | PV = "1.0.0" |
| Basic | PN | The name of the package, typically derived from the recipe file name. | PN = "mysoftware" |
| Source | SRC\_URI | List of URIs for fetching source files, patches, or additional resources. | SRC\_URI = "http://example.com/mysoftware-${PV}.tar.gz" |
| Source | S | The source directory where the package source files are located after extraction. | S = "${WORKDIR}/mysoftware" |
| Destination | D | The destination directory for installation; specifies where files will be installed in the image. | D = "${prefix}/usr" |
| Licensing | LICENSE | The license under which the package is distributed, important for compliance and distribution. | LICENSE = "MIT" |
| Licensing | LIC\_FILES\_CHKSUM | Checksums for license files, used to verify that the correct license files are present. | LIC\_FILES\_CHKSUM = "file://LICENSE;md5=..." |
| Dependencies | DEPENDS | Lists build-time dependencies required for building the package. | DEPENDS = "libfoo" |
| Dependencies | RDEPENDS | Lists runtime dependencies needed for the package after it has been built. | RDEPENDS:${PN} = "libbar" |
| Image | IMAGE\_INSTALL | Packages to include in the final image, defining what software will be present in the output. | IMAGE\_INSTALL = "mysoftware" |
| Configuration | EXTRA\_OEMAKE | Additional parameters passed to the make command during the build process. | EXTRA\_OEMAKE = "DEBUG=1" |
| Build Options | CFLAGS | Additional flags to pass to the C compiler. | CFLAGS += "-O2" |
| Build Options | LDFLAGS | Additional flags for the linker. | LDFLAGS += "-static" |
| File Locations | WORKDIR | The working directory where the package is built; contains the source, build, and install directories. | WORKDIR = "${TOPDIR}/work/${PN}-${PV}" |
| Output | PACKAGES | A list of packages that will be created from the recipe. | PACKAGES = "${PN} ${PN}-dev" |
| Output | BBFILES | A space-separated list of recipe files BitBake uses to build software. | BBFILES = "${LAYERDIR}/recipes-example/\*.bb" |
| Fetchers | BB\_FETCH\_PREMIRRORONLY | When set, causes BitBake’s fetcher to only search PREMIRRORS for files. | BB\_FETCH\_PREMIRRORONLY = "1" |
| Environment | BB\_ENV\_PASSTHROUGH | Specifies which environment variables are allowed through into BitBake’s datastore. | BB\_ENV\_PASSTHROUGH = "VAR1 VAR2" |
| Build Control | BB\_NO\_NETWORK | Disables network access in fetcher modules, useful for offline builds. | BB\_NO\_NETWORK = "1" |
| Debugging | BBDEBUG | Sets the BitBake debug output level to a specific value. | BBDEBUG = "1" |
| Logging | BB\_CONSOLELOG | Specifies the path to a log file for BitBake’s user interface output during the build. | BB\_CONSOLELOG = "${TOPDIR}/build.log" |
| Task Management | BB\_CURRENTTASK | Contains the name of the currently running task (without the do\_ prefix). | BB\_CURRENTTASK = "fetch" |
| Task Management | BB\_RUNFMT | Specifies the format for naming run files saved into ${T}. | BB\_RUNFMT = "run.${TASK}.${PID}" |
| Overrides | OVERRIDES | Controls variable overrides after BitBake parses recipes and configuration files. | OVERRIDES = "machine:arch" |
| Multi-Configuration | BBMULTICONFIG | Enables BitBake to perform multiple configuration builds. | BBMULTICONFIG = "configA configB" |

This table includes both common and less common BitBake variables, providing a comprehensive overview of their roles and examples of usage.

## 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.

Think of them like *base classes* in OOP.

|  |  |  |  |
| --- | --- | --- | --- |
| **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.

Think of them like *derived classes* in OOP.

The append files and corresponding recipe files **must have the same root file name**. The difference is the file extension, e.g. abc.bb and abc.bbappend. Also you can use the % wildcard in append files, e.g. abc\_1.21.%.bbappend and abc\_1.21.0.bb recipe. Note that the % character only works in front of the .bbappend portion; you cannot use it in any other portion of the name.

## Include Files

Include files have .inc extension. They're included in other recipe files using the include or require directives. Thus, they help to break down complex recipes into smaller, reusable components.

While the first directive includes the file regardless of its existence (build continues if the file is missing), the later requires the file to be present (build stops with an error if the file is missing).

# 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)

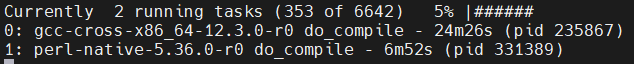
## What Is BitBake?

Check [this session](#_Bitbake).

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

* BitBake executes **tasks** according to the provided **metadata**.
* BitBake controls how software is built through "**recipes**", while GNU Make uses "makefiles".
* BitBake **extends** thecapabilities of GNU Make with much more complex tasks, such as fetching library from the internet, managing version control, assembling entire embedded Linux distributions, etc.
* BitBake is written in **Python**.

While source code is compiling, if you see something like that, you know that Bibake is executing 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

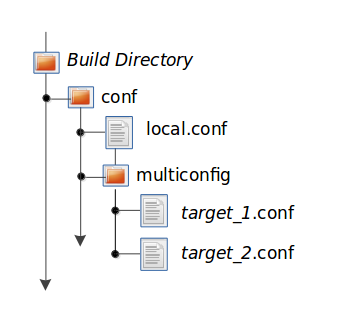
The following comand generates a dependency graph for package foo, but it omits depends common in OpenEmbedded from the graph:

$ 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

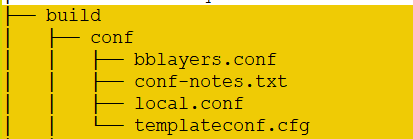
**Know that you can generate a dependency graph and learn how to do it:** A dependency graph shows dependencies between recipes, tasks, and targets. You can use the “-g” option with BitBake to generate this graph. When you start a build and the build breaks, you could see packages you have no clue about 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:



# References

**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>