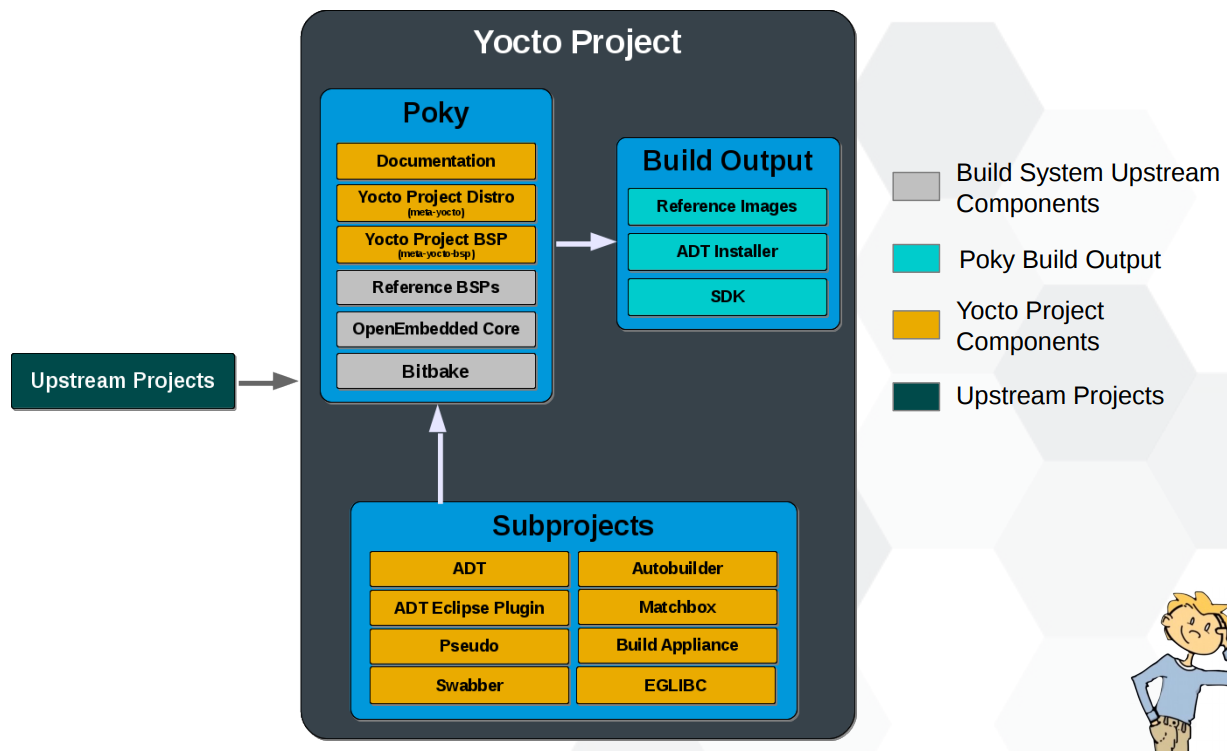
# Introduction

## What Is Yocto?

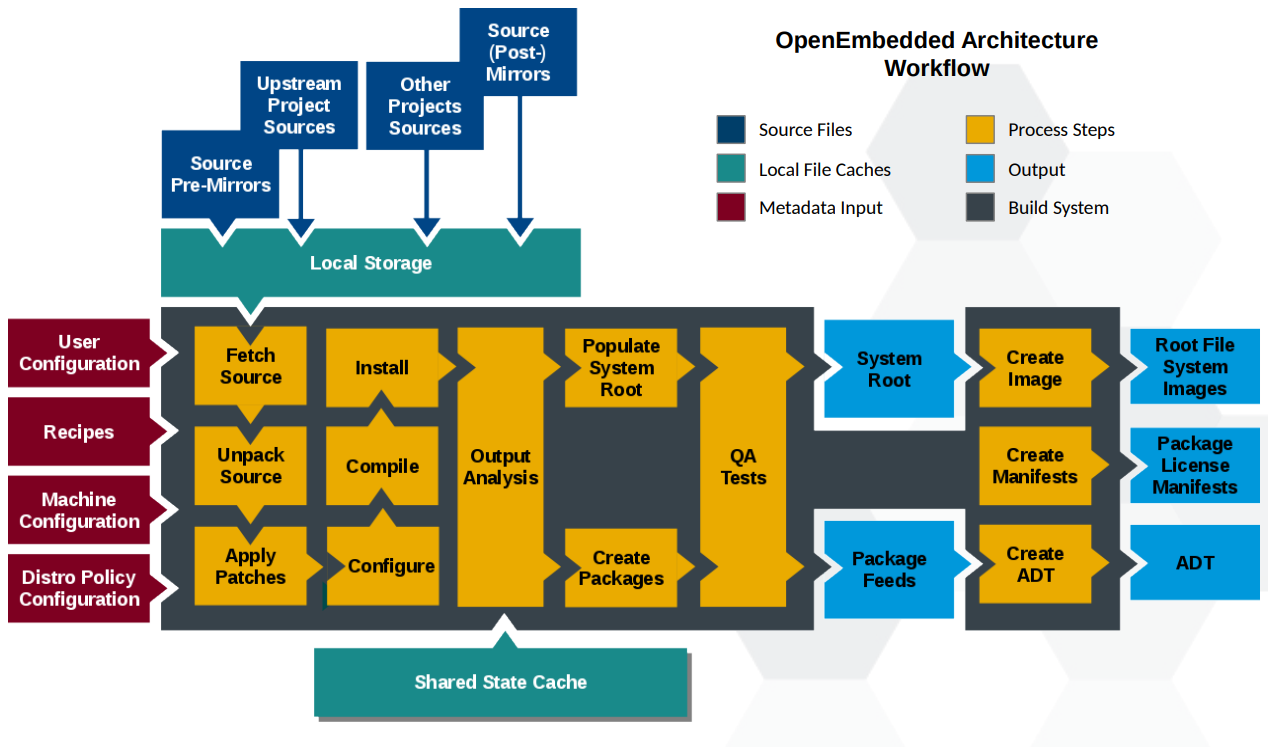
* Open-source project with a strong community
* A collection of embedded projects and tooling:
* Place for Industry to publish BSPs
* Application Development Tools including Eclipse plug-ins and emulators
* Key project is the reference distribution build environment (Poky) with:
  + Complete build system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

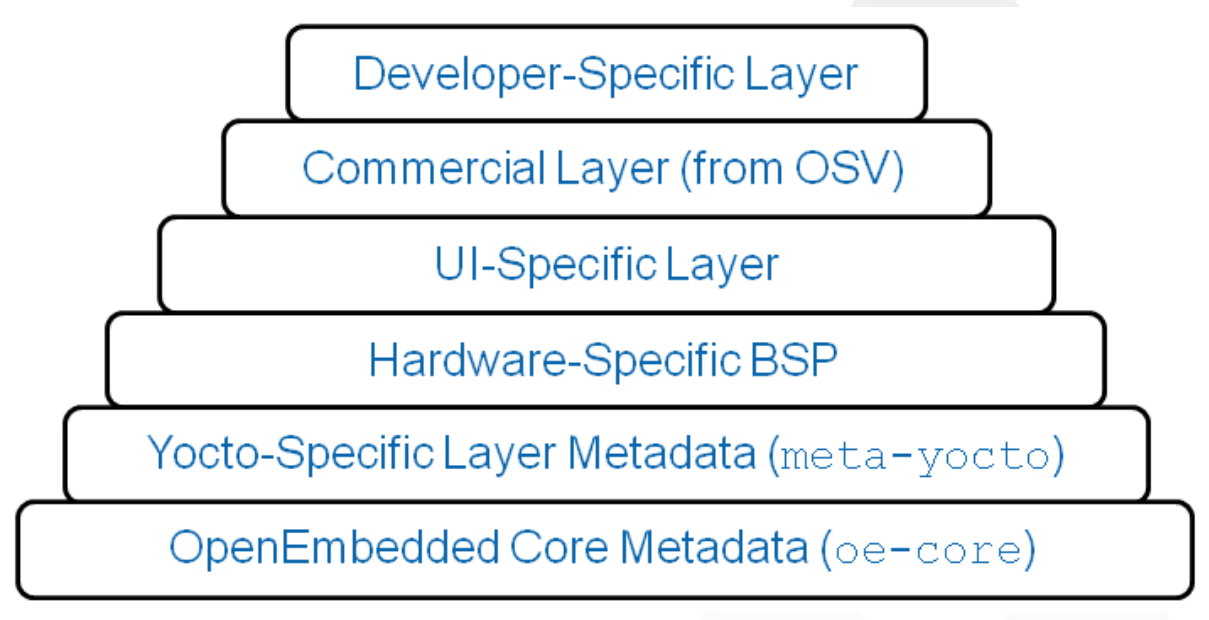
|  |  |
| --- | --- |
| **Concept** | **Description** |
| **BitBake** | It's a **build engine** used by the build system to build images. |
| **OpenEmbedded** | It's a **build system** which runs on a build engine. |
| **Recipe** | **Files that define how to build** software packages. Recipes specify the source code, dependencies, build instructions, metadata.  They are written in a domain-specific language based on Python. They have .bb extension. |
| **Append file** | Files that **extend or override** information in an existing recipe file.  They have .bbappend extension. |
| **Included file** | Files that be included in other recipe files, using the include or require directives, to break down complex recipes into smaller, reusable components.  They have .inc 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, the layer name starts with meta-.  Layers are actually folders. |
| **Configuration file** | 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, build options.  They have .conf extension. |
| **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.  *This is where the results of the build are stored (images, packages, logs, etc.) and where temporary build files are created.* |
| **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** | 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.

Layers can be used, extended, configured and it is also possible to partial overwrite parts of existing layers.

## 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:

|  |  |
| --- | --- |
| **File** | **Description** |
| build/conf/bblayers.conf | * Specify **paths for config files and class files** that should be available during the build via BBPATH variable. * Specify **paths for layers** that will be processed during the build via BBLAYERS variable. * Specify **names for recipe files and append files** that will be processed during the build via BBFILES variable. |
| build/conf/local.conf |  |
| <layer-name>/conf/layer.conf | * Specify **paths for config files and class files** that should be available during the build via BBPATH variable. * Specify **names for recipe files and append files** that will be processed during the build via BBFILES variable. |
| <layer-name>/conf/bitbake.conf |  |

## Recipe Files

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.

## Class Files

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).

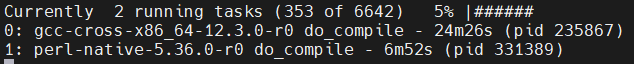
# BitBake

## What Is 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:



## CLI

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

Options:

|  |  |
| --- | --- |
| **Option** | **Description** |
| -b BUILDFILE,  --buildfile=BUILDFILE | **Execute tasks** from a specific .bb recipe directly.  WARNING: Does not handle any dependencies from other recipes. |
| -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. Very helpful to verify Bitbake syntax. |
| -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). |

## Installation

1. Clone BitBake from <https://github.com/openembedded/bitbake>
2. Add BitBake executables to PATH environment:

$ export PATH=/path/to/bitbake/bin:$PATH

Note: The export command only sets environment variables for the current shell session. When you close the terminal or restart your system, those changes are lost. So, to make them persistent, add the export command to ~/.bashrc (assumpt your shell is Bash – can check with command echo $SHELL).

1. Check if bitbake is installed:

$ bitbake --version

## Syntax

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

## Built-In Variables

|  |  |  |
| --- | --- | --- |
| **Option** | **Description** | **Example** |
| **build/conf/bblayers.conf, <layer-name>/conf/layer.conf** | | |
| BBPATH | Specify **paths for config files** (under the conf directory) **and class files** (under the classes directory)that should be available during the build. Multiple paths are separated by colons.   * *This variable is analogous to the PATH variable. If you run BitBake from a directory outside of the build directory, you must be sure to set BBPATH to point to the build directory. Set the variable as you would any environment variable and then run BitBake.* * *BBPATH is* ***recursive*** *searched by default. This means if we tell BitBake to search for some path, it will search all directives in BBPATH for that path.* * *This variable is used in build/conf/bblayers.conf* *and in <layer-name>/conf/layer.conf.* | $ BBPATH=”build\_directory”  $ export BBPATH  $ bitbake target |
| BBLAYERS | Specify **paths for layers** to enable during the build. Multiple paths are separated by spaces.  *This variable is used in build/conf/bblayers.conf.* | BBLAYERS = " \  /home/scottrif/poky/meta \  /home/scottrif/poky/meta-yocto \  /home/scottrif/poky/meta-yocto-bsp \  /home/scottrif/poky/meta-mykernel \ |
| BBFILES | Specify **names for recipe files** (.bb) **and append files** (.bbappend) that will be processed during the build. Multiple paths are separated by spaces. Wildcard is supported.  *This variable is used in build/conf/bblayers.conf* *and in <layer-name>/conf/layer.conf.* | BBFILES +=  "${LAYERDIR}/recipes-\*/\*/\*.bb" |
| LAYERDIR | Specify the path of the current layer.  *This variable is used in <layer-name>/conf/layer.conf.* |  |
| LAYERDEPENDS | Specify **layers upon which the recipe depends**. Multiple layers are separated by spaces.  *This variable is used in <layer-name>/conf/layer.conf. It must be also used with the specific layer name as a suffix (e.g. LAYERDEPENDS\_mylayer).* |  |
| BBFILE\_COLLECTIONS | Specify **names of configured layers**. These names are used to find the other BBFILE\_\* variables.  *Typically, each layer appends its name to this variable in its <layer-name>/conf/layer.conf file.* |  |
| BBFILE\_PATTERN | Expand to match files from BBFILES in a particular layer.  *This variable is used in the conf/layer.conf file and must be suffixed with the name of the specific layer (e.g. BBFILE\_PATTERN\_mylayer).* |  |
| BBFILE\_PRIORITY | Specify the **priority for recipe files in each layer**.  This variable is useful in situations where **the same recipe appears in more than one layer**. Setting this variable allows you to prioritize a layer against other layers that contain the same recipe — effectively letting you control the precedence for the multiple layers.  A larger value for the BBFILE\_PRIORITY variable results in a higher precedence.  If not specified, the BBFILE\_PRIORITY variable is set based on layer dependencies (see the LAYERDEPENDS)  The default priority, if unspecified for a layer with no dependencies, is the lowest defined priority + 1 (or 1 if no priorities are defined).  *This variable is used in the conf/layer.conf file and must be suffixed with the name of the specific layer (e.g. BBFILE\_PRIORITY\_mylayer).* |  |
| **meta/conf/bitbake.conf** | | |
| CACHE | Specify the path of **directory to store a cache** of the metadata so that Bitbake does not need to be parsed every time BitBake is started. |  |
| **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" |  |
| BBMULTICONFIG |  |  |
| **Directory** | | |
| TOPDIR | Get the path to the build directory. |  |
| SRC\_URI | Specify paths (local or remote) to source files, patches, or additional resources.  Each recipe must have a SRC\_URI. | SRC\_URI = "http://example.com/mysoftware-${PV}.tar.gz" |
| **Project Info** | | |
| PN | Recipe name or package name. | PN = "mysoftware" |
| PV | Recipe version or package version. |  |
| PR | Recipe revision or package revision. | PV = "1.0.0" |
| **Recipe Info** | | |
| DESCRIPTION | A long description for the recipe. |  |
| **Dependencies** | | |
| DEPENDS | Specify a list of **recipe’s build-time dependencies** (i.e. other recipe files). | DEPENDS = "libfoo" |
| RDEPENDS | Specify a list of **package’s runtime dependencies** (i.e. other packages) that must be installed in order for the built package to run correctly. If a package in this list cannot be found during the build, you will get a build error. | # The development package depends on the perl package  RDEPENDS:${PN}-dev += "perl" |

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

## Examples

### Quick Start

quickstart

├── meta-quickstart # This is a layer

│   ├── **conf** # Compulsary config directory.

│   │   ├── **layer.conf** # Compulsary layer file. One per layer.

│   │   │ # Missing file or wrong name will cause error " FileNotFoundError: [Errno 2] file conf/layer.conf not found".

│   │   └── **bitbake.conf** # Compulsary config.

│ │ # Missing file or wrong name will cause error "FileNotFoundError: [Errno 2] file conf/bitbake.conf not found".

│   ├── **classes** # Compulsary class directory.

│ │ │ # Missing file or wrong name will cause error "ParseError: Could not inherit file classes/base.bbclass".

│   │   ├── **base.bbclass** # Compulsary base class.

│ │ │ # Missing file or wrong name will cause error "ParseError: Could not inherit file classes/base.bbclass".

│   │   ├── mybuild.bbclass # Custom class which inherits from base.bbclass.

│   └── recipes-quickstart

│     ├── first

│     │   └── first\_0.1.bb

│     └── second

│         └── second\_0.1.bb

├── meta-two

│   ├── conf

│   │   └── layer.conf

│   ├── classes

│   │   └── confbuild.bbclass

│   └── recipes-base

│     ├── third

│     │   └── third\_0.1.2.bb

│     └── first

│         └── first\_0.1.bbappend

└── build # Build directory. You can name it any name.

    ├── **conf** # Compulsary config directory.

    │   └── **bblayers.conf** # Compulsary layer file.

│ # Missing file or wrong name will cause error "Bitbake did not find a conf/bblayers.conf file in the expected location."

    ├── cache # Generated cache by Bitbake.

    └── tmp # Generated output directory by Bitbake.

    ├── cache

    ├── stamps

    └── work

        ├── first\_0.1-r1 # Generated output directory for recipe "first".

        │ └── temp # Contain all the logs of recipe "first".

        └── second\_0.1-r1

        └── temp

#### Minimal Configuration

There are crucial files, including config files and base class files, that are required for setting up every Bitbake project:

* bblayers.conf
* bitbake.conf
* layer.conf
* base.bbclass

Also, the build directory will be our working directory. You can name it any, but it is a good practice to start the name with build.

|  |  |
| --- | --- |
| **quickstart/build/conf/bblayers.conf:**  # Paths for config files (under the conf directory) and class files (under the classes directory).  # BBPATH is recursive. This means all sub-directives in BBPATH will be searched.  # TOPDIR is the build directory.  BBPATH := "${TOPDIR}"  # Names for recipe files (.bb) and append files (.bbappend).  # Now just let it empty. Later we will add recipe files.  BBFILES ?= ""  # Paths for layers to enable during the build.  # Now we have one layer only. Later we will add more.  BBLAYERS = " ${TOPDIR}/../meta-quickstart"  # Path of directory to store a cache of the metadata.  # Thanks to it, Bitbake does not need to be parsed every time BitBake is started.  CACHE = "${TOPDIR}/cache" | **quickstart/meta-quickstart/conf/layer.conf:**  # Paths for config files and class files.  # BBPATH is recursive. All sub-dir will be searched.  # LAYERDIR is the layer directory.  BBPATH .= ":${LAYERDIR}"  # Names for recipe files (.bb) and append files (.bbappend).  # We use wildcard here. All files with ".bb" ext will be get.  BBFILES += "${LAYERDIR}/recipes-\*/\*/\*.bb" |
| **quickstart/meta-quickstart/conf/bitbake.conf:**  # This file contains a bunch of variables which, for now, we just take as they are.  B = "${S}"  CVSDIR = "${DL\_DIR}/cvs"  DEPENDS = ""  DEPLOY\_DIR = "${TMPDIR}/deploy"  DEPLOY\_DIR\_IMAGE = "${DEPLOY\_DIR}/images"  DL\_DIR = "${TMPDIR}/downloads"  FETCHCOMMAND = ""  FETCHCOMMAND\_cvs = "/usr/bin/env cvs -d${CVSROOT} co ${CVSCOOPTS} ${CVSMODULE}"  FETCHCOMMAND\_svn = "/usr/bin/env svn co ${SVNCOOPTS} ${SVNROOT} ${SVNMODULE}"  FETCHCOMMAND\_wget = "/usr/bin/env wget -t 5 --passive-ftp -P ${DL\_DIR} ${URI}"  FILESDIR = "${@bb.utils.which(d.getVar('FILESPATH', True), '.')}"  FILESPATH = "${FILE\_DIRNAME}/${PF}:${FILE\_DIRNAME}/${P}:${FILE\_DIRNAME}/${PN}:${FILE\_DIRNAME}/files:${FILE\_DIRNAME}"  FILE\_DIRNAME = "${@os.path.dirname(d.getVar('FILE', False))}"  GITDIR = "${DL\_DIR}/git"  IMAGE\_CMD = "\_NO\_DEFINED\_IMAGE\_TYPES\_"  IMAGE\_ROOTFS = "${TMPDIR}/rootfs"  MKTEMPCMD = "mktemp -q ${TMPBASE}"  MKTEMPDIRCMD = "mktemp -d -q ${TMPBASE}"  OVERRIDES = "local:${MACHINE}:${TARGET\_OS}:${TARGET\_ARCH}"  P = "${PN}-${PV}"  PERSISTENT\_DIR = "${TMPDIR}/cache"  PF = "${PN}-${PV}-${PR}"  PN = "${@bb.parse.vars\_from\_file(d.getVar('FILE', False),d)[0] or 'defaultpkgname'}"  PR = "${@bb.parse.vars\_from\_file(d.getVar('FILE', False),d)[2] or 'r0'}"  PROVIDES = ""  PV = "${@bb.parse.vars\_from\_file(d.getVar('FILE', False),d)[1] or '1.0'}"  RESUMECOMMAND = ""  RESUMECOMMAND\_wget = "/usr/bin/env wget -c -t 5 --passive-ftp -P ${DL\_DIR} ${URI}"  S = "${WORKDIR}/${P}"  SRC\_URI = "file://${FILE}"  STAMP = "${TMPDIR}/stamps/${PF}"  SVNDIR = "${DL\_DIR}/svn"  T = "${WORKDIR}/temp"  TARGET\_ARCH = "${BUILD\_ARCH}"  TMPDIR = "${TOPDIR}/tmp"  UPDATECOMMAND = ""  UPDATECOMMAND\_cvs = "/usr/bin/env cvs -d${CVSROOT} update ${CVSCOOPTS}"  UPDATECOMMAND\_svn = "/usr/bin/env svn update ${SVNCOOPTS}"  WORKDIR = "${TMPDIR}/work/${PF}"  PERSISTENT\_DIR = "${TMPDIR}/cache"  CACHE = "${TMPDIR}/cache/default" | **quickstart/meta-quickstart/classes/base.bbclass:**  die() {      bbfatal "$\*"  }  bbnote() {      echo "NOTE:" "$\*"  }  bbwarn() {      echo "WARNING:" "$\*"  }  bbfatal() {      echo "FATAL:" "$\*"      exit 1  }  addtask showdata  do\_showdata[nostamp] = "1"  python do\_showdata() {      import sys      # emit variables and shell functions      bb.data.emit\_env(sys.\_\_stdout\_\_, d, True)      # emit the metadata which isnt valid shell      for e in bb.data.keys(d):          if d.getVarFlag(e, 'python', False):              bb.plain("\npython %s () {\n%s}" % (e, d.getVar(e, True)))  }  addtask listtasks  do\_listtasks[nostamp] = "1"  python do\_listtasks() {      import sys      for e in bb.data.keys(d):          if d.getVarFlag(e, 'task', False):              bb.plain("%s" % e)  }  addtask build  do\_build[dirs] = "${TOPDIR}"  do\_build[nostamp] = "1"  python base\_do\_build () {      bb.note("The included, default BB base.bbclass does not define a useful default task.")      bb.note("Try running the 'listtasks' task against a .bb to see what tasks are defined.")  }  EXPORT\_FUNCTIONS do\_clean do\_mrproper do\_build |

After having these files, you can test our initial setup:

|  |  |
| --- | --- |
| # In build directory  $ bitbake | If our setup is correct, Bitbake will report:  Nothing to do. Use 'bitbake world' to build everything,  or run 'bitbake --help' or usage information. |

#### First Recipe

Create a recipe file named meta-quickstart/recipes-quickstart/first/first\_0.1.bb. Note that name of recipe files have a common pattern: {recipe}\_{version}.bb.

|  |
| --- |
| **quickstart/meta-quickstart/recipes-quickstart/first/first\_0.1.bb:**  DESCRIPTION = "I am the first recipe"  # Revision number (should be updated after each major change)  PR = "r1"  # Overrides the empty global build task from base.bbclass.  do\_build () {      echo "first: some shell script running as build"  } |

Test our recipe by running:

|  |  |
| --- | --- |
| # Show recipes' version  $ bitbake -s  # See all tasks in the "first" recipe  $ bitbake -c listtasks first  # Run all tasks from the "first" recipe  $ bitbake first | Recipe Name Latest Version Preferred Version Required Version  =========== ============== ================= ================  first :0.1-r1  do\_listtasks  do\_build  do\_showdata  Loading cache: 100% |############################################################################################| Time: 0:00:00  Loaded 1 entries from dependency cache.  NOTE: Resolving any missing task queue dependencies  Initialising tasks: 100% |############################################################################################| Time: 0:00:00  NOTE: No setscene tasks  NOTE: Executing Tasks  NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded. |

After that you will see a build/tmp/work/first-0.1-r1 directory where all logs of executed tasks are stored:

└── work

    ├── first-0.1-r1

    │   └── temp

    │       ├── log.do\_build -> log.do\_build.486952

    │       ├── log.do\_build.486952

    │       ├── log.do\_listtasks -> log.do\_listtasks.534068

    │       ├── log.do\_listtasks.534068

    │       ├── log.task\_order

    │       ├── run.do\_build -> run.do\_build.486952

    │       ├── run.do\_build.486952

    │       ├── run.do\_listtasks -> run.do\_listtasks.534068

    │       └── run.do\_listtasks.534068

#### Custom Class

Create a new class file named meta-quickstart/classes/mybuild.bbclass:

|  |
| --- |
| **meta-quickstart/classes/mybuild.bbclass:**  # Declares a task named "build".  # Note: Bitbake will add the "do\_" prefix to the function name, if it was not given.  addtask build  # The "mybuild\_do\_" prefix is for following the conventions: classname\_do\_functionname for a task in a class.  mybuild\_do\_build () {      echo "running mybuild\_do\_build."  }  # Make the build function available to users of this class.  # The function, which will execute the task "build", is "mybuild\_do\_build"  # because the base name matches (i.e., do\_build).  EXPORT\_FUNCTIONS do\_build |

We'll use this class with our second recipe.

#### Second Recipe

Create a recipe file named meta-quickstart/recipes-quickstart/second/second\_1.0.bb:

|  |
| --- |
| **meta-quickstart/recipes-quickstart/second/second\_1.0.bb:**  DESCRIPTION = "I am the second recipe"  PR = "r1"  # Inherit "mybuild" class. So "myclass\_do\_build" becomes the default build task.  inherit mybuild  # A pure python function takes an argument and runs the python "dir" function on this argument and prints the result.  def pyfunc(o):      print (dir(o))  # A bitbake python function is added and registered as a task that needs to be executed before the build function.  python do\_mypatch () {      bb.note ("Running mypatch")      # Calls pyfunc and passes the global bitbake variable d.      # Note: d (datastore) is defined by bitbake and is always available.      pyfunc(d)  }  # The "mypatch" function is registered as a task that needs to be executed before the "build" function.  addtask mypatch before do\_build |

Test our recipe by running:

|  |  |
| --- | --- |
| # Show recipes version  $ bitbake -s  # See all tasks in the "second" recipe  $ bitbake -c listtasks second  # Run all tasks from the "second" recipe  $ bitbake second  # Run a specific task for the "second" recipe  $ bitbake -c mypatch second  # Run everything (all tasks for all recipes)  $ bitbake world | Recipe Name Latest Version Preferred Version Required Version  =========== ============== ================= ================  first :0.1-r1  second :0.1-r1  do\_showdata  do\_mypatch  do\_build  do\_listtasks  Loading cache: 100% |########################################################################################| Time: 0:00:00  Loaded 2 entries from dependency cache.  NOTE: Resolving any missing task queue dependencies  Initialising tasks: 100% |########################################################################################| Time: 0:00:00  NOTE: No setscene tasks  NOTE: Executing Tasks  NOTE: Tasks Summary: Attempted 2 tasks of which 0 didn't need to be rerun and all succeeded. |

And logs:

└── work

└── second-1.0-r1

└── temp

├── log.do\_build -> log.do\_build.534808

├── log.do\_build.534808

├── log.do\_listtasks -> log.do\_listtasks.534015

├── log.do\_listtasks.534015

├── log.do\_mypatch -> log.do\_mypatch.534805

├── log.do\_mypatch.534805

├── log.task\_order

├── run.do\_build -> run.do\_build.534808

├── run.do\_build.534808

├── run.do\_listtasks -> run.do\_listtasks.534015

├── run.do\_listtasks.534015

├── run.do\_mypatch -> run.do\_mypatch.534805

└── run.do\_mypatch.534805

#### Adding More Layer

Create a new layer called meta-two. We'll need to create a new layer.conf, as well as telling BitBake about the new layer recipes in bblayers.conf.

|  |  |
| --- | --- |
| **quickstart/meta-two/conf/layer.conf:**  BBPATH .= ":${LAYERDIR}"  BBFILES += "      ${LAYERDIR}/recipes-\*/\*/\*.bb \  " | **quickstart/build/conf/bblayers.conf:**  ...  BBLAYERS = " \    ${TOPDIR}/../meta-quickstart \    ${TOPDIR}/../meta-two \  "  ... |

#### Extending Layer Configuration

Extend the layer configuration by adding a *layer collection name*, a *search pattern for files* to add, and a *layer priority*. We will do this for both meta-quickstart and meta-two:

|  |  |
| --- | --- |
| **quickstart/meta-quickstart/conf/layer.conf:**  ...  # Append layer name to list of configured layers  # Then we can use the layer name as suffix for other properties  BBFILE\_COLLECTIONS += "quickstart"  # Expand to match files from BBFILES in a particular layer.  BBFILE\_PATTERN\_quickstart = "^${LAYERDIR}/"  # Specify the priority for recipe files in each layer.  BBFILE\_PRIORITY\_quickstart = "5"  ... | **quickstart/meta-two/conf/layer.conf:**  ...  BBFILE\_COLLECTIONS += "two"  BBFILE\_PATTERN\_two = "^${LAYERDIR}/"  BBFILE\_PRIORITY\_two = "5"  LAYERVERSION\_two = "1"  ... |

Now running:

|  |  |
| --- | --- |
| $ bitbake-layers show-layers | layer path priority  ========================================================================================================  quickstart /home/centos/SharedFolder/bitbake/quickstart/build/../meta-quickstart 5  two /home/centos/SharedFolder/bitbake/quickstart/build/../meta-two 5 |

#### Layer Compatibility

A Bitbake project might be composed out of many layers. To ensure they are compatible with a project version, a project can define a *layer series* name, and layers can specify to be compatible to one or multiple layer series.

In practice, for the Yocto project, each release defines its release name as its *layer series core name*. Layers that are tested for this release can add the compatibility name in its config. If a layer is added that does not have the compatibility name specified, Bitbake will tell about this by showing a warning.

We can easily verify this. So far there is no core layer series name specified in our example. Running, for example, bitbake-layers show-recipes will give 2 warnings:

|  |  |
| --- | --- |
| $ bitbake-layers show-recipes | ...  WARNING: Layer quickstart should set LAYERSERIES\_COMPAT\_quickstart in its conf/layer.conf file to list the core layer names it is compatible with.  WARNING: Layer two should set LAYERSERIES\_COMPAT\_two in its conf/layer.conf file to list the core layer names it is compatible with.  ...  Summary: There was 2 WARNING message.  === Available recipes: ===  first:  meta-quickstart 0.1  second:  meta-quickstart 1.0 |

These warnings reflect the fact that the project has no layer series compatibility specified. So we need to define a *project core name*. This is done by setting the LAYERSERIES\_CORENAMES variable.

We also need to specify that the quickstart layer is compatible with the bitbakeguilde. This can be done by setting LAYERSERIES\_COMPAT\_...​ variable in the in the layer.conf files of each layer.

|  |  |
| --- | --- |
| **quickstart/meta-quickstart/conf/layer.conf:**  ...  # Set the core name for the layer series.  # Here we put this variable in meta-quickstart/conf/layer.conf,  # we can also define it in build/conf/bblayers.conf file, or in any other layer.  LAYERSERIES\_CORENAMES = "bitbakeguilde"  # Specify that the "quickstart" layer is compatible with the "bitbakeguilde" layer series  LAYERSERIES\_COMPAT\_quickstart = "bitbakeguilde"  # Specify the layer version  LAYERVERSION\_quickstart = "1"  ... | **quickstart/meta-two/conf/layer.conf:**  ...  # Specify that the "two" layer is compatible with the "bitbakeguilde" layer series  LAYERSERIES\_COMPAT\_two = "bitbakeguilde"  ... |

#### Layer Dependency

Make the meta-two layer depending on the meta-quickstart layer via LAYERDEPENDS\_two variable:

|  |
| --- |
| **quickstart/meta-two/conf/layer.conf:**  ...  # This layer has a dependency to the "quickstart" layer  LAYERDEPENDS\_two = "quickstart"  ... |

We will see in the next chapter, when we add more content to the meta-two layer.

#### Extending Tasks and Configurations

So far we used classes and config files to encapsulate configuration and tasks. But there are more ways to reuse and extend tasks and configurations. These are:

* Class Inheritance
* Append files
* Include files

We are going to add an additional class to meta-two. The new class will introduce a configure-build chain and will reuse the existing mybuild class by using class inheritance. Then we will use this new class within a new recipe.

|  |  |
| --- | --- |
| **quickstart/meta-two/classes/confbuild.class:**  inherit mybuild  confbuild\_do\_configure () {    echo "Running configbuild\_do\_configure."  }  addtask do\_configure before do\_build  EXPORT\_FUNCTIONS do\_configure | **quickstart/meta-two/recipes-base/third/third\_0.1.2.bb:**  DESCRIPTION = "I am the third recipe"  PR = "r1"  inherit confbuild |

If we run now bitbake third it will execute the configure and build tasks for third\_0.1.2.bb.

We not stop here. We will extend our first recipe by using an append file:

|  |  |
| --- | --- |
| **quickstart/meta-two/conf/layer.conf:**  ...  BBFILES += " \      ${LAYERDIR}/recipes-\*/\*/\*.bb \      ${LAYERDIR}/recipes-\*/\*/\*.bbappend \  "  ... | **quickstart/meta-two/recipes-base/first/first\_0.1.bbappend:**  python do\_patch () {    bb.note ("first:do\_patch")  }  addtask patch before do\_build |

Running bitbake first will now run both tasks, patch and build.

|  |  |
| --- | --- |
| $ bitbake -c listtasks first | ....  do\_showdata  do\_build  do\_listtasks  **do\_patch**  ... |

Another way to extend an existing recipe is using included files. We're going to mimic a build/conf/local.conf which usually contains special setups for the current build target. This is the typical Yocto setup.

|  |
| --- |
| **quickstart/meta-quickstart/conf/bitbake.conf:**  ...  require conf/local.conf  ... |

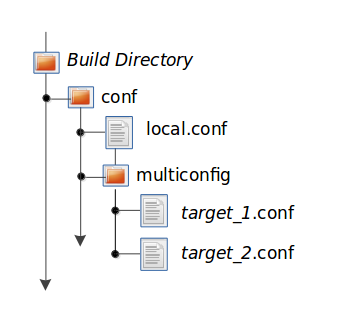
### Hello World

<https://tutorialadda.com/yocto/yocto-hello-world-recipe-compile-using-makefile>

### Multi Configuration

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

# Dependency Graph Generation

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.

Generate a dependency graph for package foo:

$ bitbake -g foo

Generate a dependency graph for package foo, but omits depends common in OpenEmbedded from the graph:

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

You can convert these graphs into images using the dot tool from [Graphviz](http://www.graphviz.org/).

# 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>

# GUI

<https://wiki.yoctoproject.org/wiki/BitBake/GUI>

# Supported Tools

## VSCode Extension

### [Yocto Project BitBake](https://marketplace.visualstudio.com/items?itemName=yocto-project.yocto-bitbake)

Official BitBake recipe language support in Visual Studio Code with following features:

* Syntax highlighting
* Autocompletion
* Goto definition
* Context menu to run Bitbake command and Bitbake Devtool
* Recipe view (on the side tab)
* Problem matcher
* Etc.

# References