BitBake

Overview

Welcome to the BitBake User Manual. This manual provides information on the BitBake tool. The information attempts to be as independent as possible regarding systems that use BitBake, such as the Yocto Project and OpenEmbedded. In some cases, scenarios or examples that within the context of a build system are used in the manual to help with understanding. For these cases, the manual clearly states the context.

Introduction

Fundamentally, BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints. One of BitBake's main users, OpenEmbedded, takes this core and builds embedded Linux software stacks using a task-oriented approach.

Conceptually, BitBake is similar to GNU Make in some regards but has significant differences:

- BitBake executes tasks according to provided metadata that builds up the tasks. Metadata is stored in recipe (.bb), configuration (.conf), and class (.bbclass) files and provides BitBake with instructions on what tasks to run and the dependencies between those tasks.
- BitBake includes a fetcher library for obtaining source code from various places such as source control systems or websites.
- The instructions for each unit to be built (e.g. a piece of software) are known as recipe files and contain all the information about the unit (dependencies, source file locations, checksums, description and so on).
- BitBake includes a client/server abstraction and can be used from a command line or used a service over XMLRPC and has several different user interfaces.

History and Goals

BitBake was originally a part of the OpenEmbedded project. It was inspired by the Portage package management system used by the Gentoo Linux distribution. On December 7, 2004, OpenEmbedded project team member, Chris Larson split the project into two distinct pieces:

- BitBake, a generic task executor.
- OpenEmbedded, a metadata set utilized by BitBake.

Today, BitBake is the primary basis of the OpenEmbedded project, which is being used to build and maintain Linux distributions such as the Angstrom Distribution and which is used as the build tool for Linux projects such as the Yocto Project.

Prior to BitBake, no other build tool adequately met the needs of an aspiring embedded Linux distribution. All of the build systems used by traditional desktop Linux distributions lacked important functionality, and none of the ad-hoc Buildroot-based systems, prevalent in the imbedded space, were scalable or maintainable.

Some important original goals for BitBake were:

- Handle cross-compilation.
- Handle inter-package dependencies (build time on target architecture, build time on native architecture, and runtime).
- Support running any number of tasks within a given package, including, but not limited to, fetching upstream sources, unpacking them, patching them, configuring them and so forth.
- Be Linux distribution agnostic for both build and target systems.
- Be architecture agnostic.
- Support multiple build and target operating systems (e.g Cygwin, the BSDs, and so forth).
- Be self-contained, rather than tightly integrated into the build machine's root filesystem.
- Handle conditional metadata on the target architecture, operating system, distribution, and machine.
- Be easy to use the tools to supply local metadata and packages against which to operate.
- Be easy to use BitBake to collaborate between multiple projects for their builds.
- Provide an inheritance mechanism that share common metadata between many packages.

Over time it became apparent that some further requirements were necessary:

- Handle variants of a base recipe (e.g native, sdk, and multilib).
- Split metadata into layers and allow layers to override each other.
- Allow representation of a given set of input variables to a task as a checksum. Based on that checksum, allow acceleration of builds with prebuilt components.

BitBake satisfies all the original requirements and many more with extensions being made to the basic functionality to reflect the additional requirements. Flexibility and power have always been the priorities. BitBake is highly extensible and supports embedded Python code and execution of any arbitrary tasks.

Concepts

BitBake is a program written in the Python language. At the highest level, BitBake interprets metadata, decides what tasks are required to run, and executes those tasks. Similar to GNU Make, BitBake controls how software is built. GNU Make achieves this control through "makefiles". BitBake uses "recipes".

BitBake extends the capabilities of a simple tool like GNU Make by allowing for much more complex tasks to be completed, such as assembling entire embedded Linux distributions.

Recipes

BitBake Recipes, which are denoted by the file extension .bb, are the most basic metadata files. These recipe files provide BitBake with the following:

- Descriptive information about the package.
- The version of the recipe.
- Existing Dependencies.
- Where the source code resides.
- Whether the source code requires any patches.

- How to compile the source code?
- Where on the target machine to install the package being compiled.

Within the context of BitBake, or any project utilizing BitBake as its build system, files with the .bb extension are referred to as recipes.

Configuration Files

Configuration files, which are denoted by the .conf extension, define various configuration variables that govern the project's build process. These files fall into several areas that define machine configuration options, distribution configuration options, compiler tuning options, general common configuration options, and user configuration options. The main configuration file is the sample bitbake.conf file, which is located within the BitBake source tree conf directory.

Classes

Class files, which are denoted by the .bbclass extension, contain information that is useful to share between metadata files. The BitBake source tree currently comes with one class metadata file called base.bbclass. You can find this file in the classes directory. The base.bbclass is special since it is always included automatically for all recipes and classes. This class contains definitions for standard basic tasks such as fetching, unpacking, configuring (empty by default), compiling (runs any Makefile present), installing (empty by default) and packaging (empty by default). These tasks are often overridden or extended by other classes added during the project development process.

Layers

Layers allow you to isolate different types of customizations from each other. While you might find it tempting to keep everything in one layer when working on a single project, the more modular you organize your metadata, the easier it is to cope with future changes.

To illustrate how you can use layers to keep things modular, consider customizations you might make to support a specific target machine. These types of customizations typically reside in a special layer, rather than a general layer, called a Board Specific Package (BSP) Layer. Furthermore, the machine customizations should be isolated from recipes and metadata that support a new GUI environment, for example. This situation gives you a couple of layers: one for the machine configurations and one for the GUI environment. It is important to understand, however, that the BSP layer can still make machine-specific additions to recipes within the GUI environment layer without polluting the GUI layer itself with those machine-specific changes. You can accomplish this through a recipe that is a BitBake append (.bbappend) file.

Append Files

Append files, which are files that have the .bbappend file extension, add or extend build information to an existing recipe file.

BitBake expects every append file to have a corresponding recipe file. Furthermore, the append file and corresponding recipe file 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).

Information in append files overrides the information in the similarly-named recipe file.

When you name an append file, you can use the wildcard character (%) to allow for matching recipe names. For example, suppose you have an append file named as follows:

busybox_1.21.%.bbappend

That append file would match any busybox_1.21.x.bb version of the recipe. So, the append file would match the following recipe names:

busybox_1.21.1.bb

busybox_1.21.2.bb

busybox_1.21.3.bb

If the busybox recipe was updated to busybox_1.3.0.bb, the append name would not match. However, if you named the append file busybox_1.%.bbappend, then you would have a match.

Obtaining BitBake

You can use git to clone, download a snapshot, or use package manager to install.

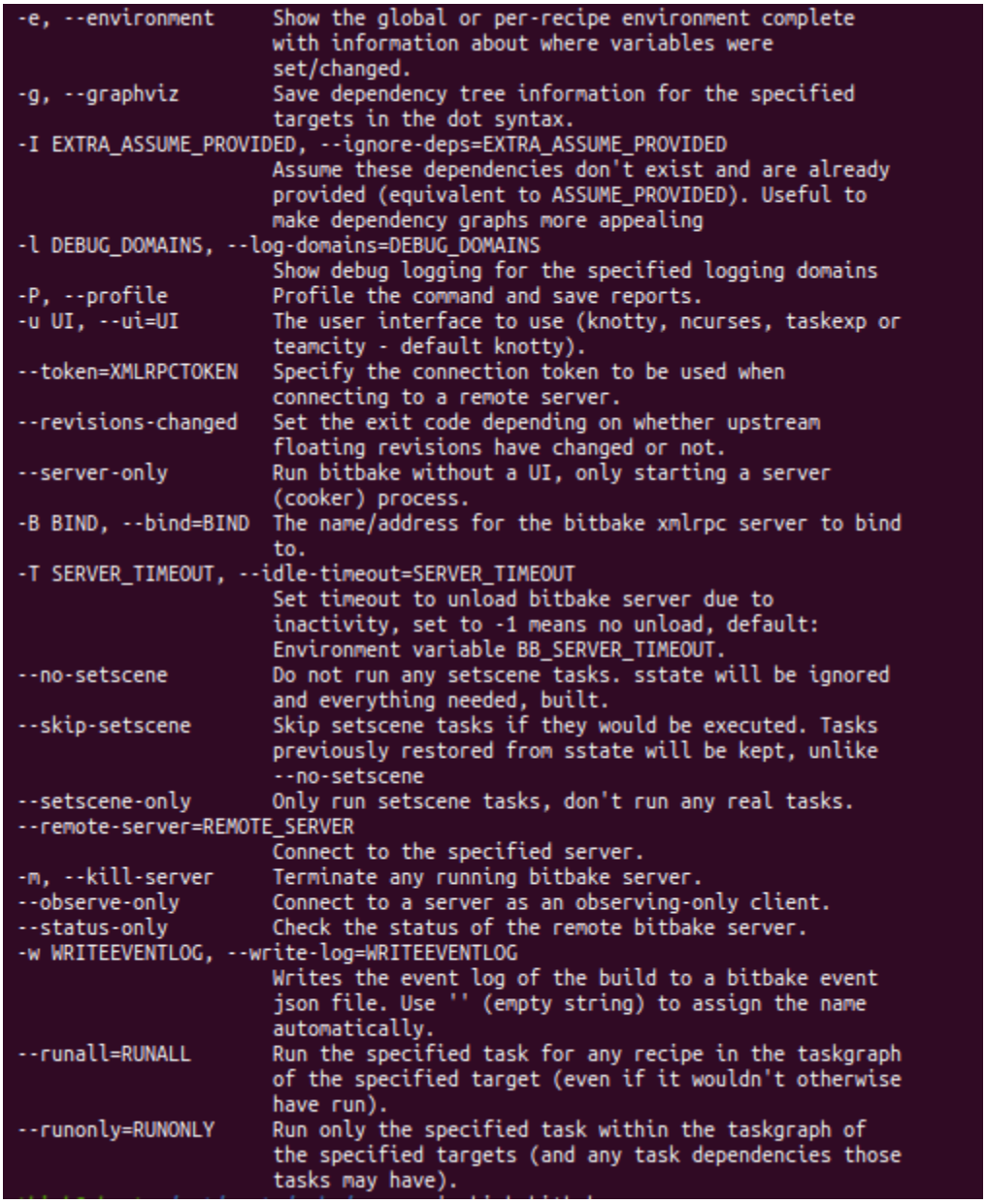
The BitBake Command

The bitbake command is the primary interface to the BitBake tool.

```
tpthinh@ubuntu:/opt/yocto/poky/qemuarm$ bitbake -h
Usage: bitbake [options] [recipename/target recipe:do_task ...]

    Executes the specified task (default is 'build') for a given set of target recipes (.bb files).
    It is assumed there is a conf/bblayers.conf available in cwd or in BBPATH which
    will provide the layer, BBFILES and other configuration information.

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -b BUILDFILE, --buildfile=BUILDFILE
                        Execute tasks from a specific .bb recipe directly.
                        WARNING: Does not handle any dependencies from other
                        recipes.
  -k, --continue        Continue as much as possible after an error. While the
                        target that failed and anything depending on it cannot
                        be built, as much as possible will be built before
                        stopping.
  -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 options
                        available 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 (in addition to
                        writing them to ${T}/log.do_<task>).
  -D, --debug           Increase the debug level. You can specify this more
                        than once. -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.
                        Without -D, no debug messages are printed. Note that
                        -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 message 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 parameter is
                        passed to the handler. Two common values are none and
                        printdiff but the handler may define more/less. none
                        means only dump the signature, printdiff means compare
                        the dumped signature with the cached one.
  -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
```



Examples

Executing tasks for a single recipe file is relatively simple. You specify the file in question, and BitBake parses it and executes the specified task. If you do not specify a task, BitBake executes the default task, which is “build”. BitBake obeys inter-task dependencies when doing so.

Executing a Task against a single recipe

The following command runs the build task, which is the default task, on the foo_1.0.bb recipe file:

$ bitbake –b foo_1.0.bb

The following command runs the clean task on the foo.bb recipe file:

$ bitbake –b foo.bb –c clean

Note: The “-b” option explicitly does not handle recipe dependencies. Other than for debugging purposes, it is instead recommended that you use the syntax presented in the next section.

Executing tasks against a set of recipe files

There are a number of additional complexities introduced when one wants to manage multiple .bb files. Clearly there needs to be a way to tell BitBake what files are available, and of those, which you want to execute. There also needs to be a way for each recipe to express its dependencies, both for build-time and runtime. There must be a way for you to express recipe preferences when multiple recipes provide the same functionality, or when there are multiple versions of a recipe.

The bitbake command, when not using “--buildfile” or “-b” only accepts a “PROVIDES”. You cannot provide anything else. By default, a recipe file generally “PROVIDES” its “packagename” as shown in the following example:

$ bitbake foo

This next example “PROVIDES” the package name and also uses the “-c” option to tell BitBake to just execute the do_clean task:

$ bitbake –c clean foo

Generating dependency graphs

BitBake is able to generate dependency graphs using the dot syntax. You can convert these graphs into images using the dot tool from Graphviz

When you generate a dependency graph, BitBake writes four files to the current working directory:

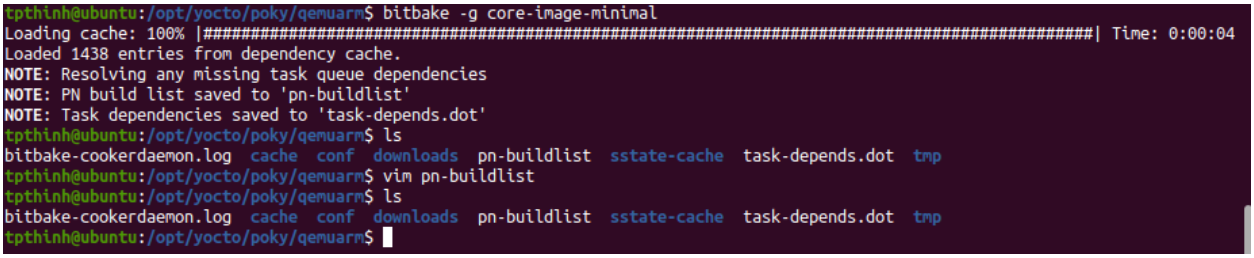
- package-depends.dot: Show BitBake’s knowledge of dependencies between runtime targets.
- pn-depends.dot: Shows dependencies between build-time targets.
- Task-depends.dot: Shows dependencies between tasks.
- pn-buildlist: Show a simple list of targets that are to be built.

To stop depending on common depends, use the “-I” depend option and BitBake omits them from the graph. Leaving this information out can produce more readable graphs. This way, you can remove from the graph DEPENDS from inherited classes such as base.bbclass.

Here are two example that create dependency graphs. The second example omits depends common in OpenEmbedded from the graph.

$ bitbake –g foo

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



It generates pn-buildlist of 238 lines

```
core-image-minimal
quilt-native
patch-native
pseudo-native
linux-yocto
rpm-native
dnf-native
createrepo-c-native
opkg-native
xz-native
cross-localedef-native
qemuwrapper-cross
mklibs-native
prelink-native
makedevs-native
ldconfig-native
opkg-utils-native
update-rc.d-native
pbzip2-native
pigz-native
qemu-system-native
qemu-helper-native
e2fsprogs-native
depmodwrapper-cross
packagegroup-core-boot
run-postinsts
attr-native
automake-native
libtool-native
autoconf-native
sqlite3-native
pkgconfig-native
binutils-cross-arm
dwarfsrcfiles-native
kmod-native
ncurses-native
kern-tools-native
gcc-cross-arm
bc-native
bison-native
openssl-native
util-linux-native
gmp-native
python3-native
bzip2-native
db-native
elfutils-native
gettext-minimal-native
file-native
libgcrypt-native
popt-native
cmake-native
```

Execution

The primary purpose for running BitBake is to produce some kind of output such as an image, a kernel, or a software development kit. Of course, you can execute the bitbake command with options that cause it to execute single tasks, compile single recipe files, capture or clear data, or simply return information about the execution environment.

This part describes BitBake's execution process from start to finish when you use it to create an image. The execution process is launched using the following command form:

$ bitbake <target>

Note

Prior to executing BitBake, you should take advantage of parallel thread execution by setting the BB_NUMBER_THREADS variable in your local.conf configuration file.

Parsing the base configuration metadata

The first thing BitBake does is parse base configuration metadata. Base configuration meta data consists of the bblayers.conf file to determine what layers BitBake needs to recognize, all necessary layer.conf files (one for each layer), and bitbake.conf. The data itself is of various types:

- Recipes: Details about particular pieces of software.
- Class Data: An abstraction of common build information (e.g how to build a Linux kernel).
- Configuration Data: Machine-specific settings, policy decisions, and so forth. Configuration data acts as the glue to bind everything together.

The layer.conf files are used to construct key variables such as BBPATH and BBFILES. BBPATH is used to search for configuration and class files under conf/ and class/ directories, respectively.

bblayer.conf

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /opt/yocto/poky/meta \
  /opt/yocto/poky/meta-poky \
  /opt/yocto/poky/meta-yocto-bsp \
  "
```

Local.conf

```
#
# Where to place downloads
#
# During a first build the system will download many different source code tarballs
# from various upstream projects. This can take a while, particularly if your network
# connection is slow. These are all stored in DL_DIR. When wiping and rebuilding you
# can preserve this directory to speed up this part of subsequent builds. This directory
# is safe to share between multiple builds on the same machine too.
#
# The default is a downloads directory under TOPDIR which is the build directory.
#
#DL_DIR ?= "${TOPDIR}/downloads"

#
# Where to place shared-state files
#
# BitBake has the capability to accelerate builds based on previously built output.
# This is done using "shared state" files which can be thought of as cache objects
# and this option determines where those files are placed.
#
# You can wipe out TMPDIR leaving this directory intact and the build would regenerate
# from these files if no changes were made to the configuration. If changes were made
# to the configuration, only shared state files where the state was still valid would
# be used (done using checksums).
#
# The default is a sstate-cache directory under TOPDIR.
#
#SSTATE_DIR ?= "${TOPDIR}/sstate-cache"

#
# Where to place the build output
#
# This option specifies where the bulk of the building work should be done and
# where BitBake should place its temporary files and output. Keep in mind that
# this includes the extraction and compilation of many applications and the toolchain
# which can use Gigabytes of hard disk space.
#
# The default is a tmp directory under TOPDIR.
#
#TMPDIR = "${TOPDIR}/tmp"

#
# Default policy config
#
# The distribution setting controls which policy settings are used as defaults.
# The default value is fine for general Yocto project use, at least initially.
```

```
#
# This file is your local configuration file and is where all local user settings
# are placed. The comments in this file give some guide to the options a new user
# to the system might want to change but pretty much any configuration option can
# be set in this file. More adventurous users can look at local.conf.extended
# which contains other examples of configuration which can be placed in this file
# but new users likely won't need any of them initially.
#
# Lines starting with the '#' character are commented out and in some cases the
# default values are provided as comments to show people example syntax. Enabling
# the option is a question of removing the # character and making any change to the
# variable as required.

#
# Machine Selection
#
# You need to select a specific machine to target the build with. There are a selection
# of emulated machines available which can boot and run in the QEMU emulator:
#
MACHINE ?= "qemuarm"
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemumips"
#MACHINE ?= "qemumips64"
#MACHINE ?= "qemuppc"
#MACHINE ?= "qemux86"
#MACHINE ?= "qemux86-64"
#
# There are also the following hardware board target machines included for
# demonstration purposes:
#
#MACHINE ?= "beaglebone-yocto"
#MACHINE ?= "genericx86"
#MACHINE ?= "genericx86-64"
#MACHINE ?= "edgerouter"
#
# This sets the default machine to be qemux86-64 if no other machine is selected:
MACHINE ??= "qemux86-64"
```

BBFILES is used to find recipe files (.bb and .bbappend). If there is no bblayers.conf file, it is assumed the user has set the BBPATH and BBFILES directly in the environment.

Next, the bitbake.conf file is searched using the BBPATH variable that was just constructed.

```
tpthinh@ubuntu:/opt/yocto/poky$ find ./ -name "bitbake.conf"
./meta/conf/bitbake.conf
./bitbake/lib/bb/tests/runqueue-tests/conf/bitbake.conf
tpthinh@ubuntu:/opt/yocto/poky$
```

The bitbake.conf file may also include other configuration files using the include or require directives.

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

- BB_ENV_WHITELIST
- BB_PRESERVE_ENV
- BB_ENV_EXTRAWHITE
- BITBAKE_UI

The base configuration metadata is global and therefore affects all recipes and tasks that are executed.

BitBake first searches the current working directory for an optional conf/bblayers.conf configuration file. This file is expected to contain a BBLAYERS variable that is a space delimited list of 'layer' directories. Recall that if BitBake cannot find a bblayers.conf file then it is assumed the user has set the BBPATH and BBFILES directly in the environment.

For each directory (layer) in this list, a conf/layer.conf file is searched for and parsed with the LAYERDIR variable being set to the directory where the layer was found. The idea is these files automatically setup BBPATH and other variables correctly for a given build directory.

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /opt/yocto/poky/meta \
  /opt/yocto/poky/meta-poky \
  /opt/yocto/poky/meta-yocto-bsp \
  "
```

```
tpthinh@ubuntu:/opt/yocto/poky$ ls ./meta*
./meta:
classes      files        recipes-connectivity  recipes-extended  recipes-kernel      recipes-sato     site
conf         lib          recipes-core          recipes-gnome     recipes-multimedia  recipes-support
COPYING.MIT  recipes-bsp  recipes-devtools      recipes-graphics  recipes-rt          recipes.txt

./meta-poky:
classes  conf  README.poky  recipes-core

./meta-selftest:
classes  conf  files  lib  README  recipes-devtools  recipes-test  wic

./meta-skeleton:
conf  recipes-baremetal  recipes-core  recipes-kernel  recipes-multilib  recipes-skeleton

./meta-yocto-bsp:
conf  lib  README.hardware  recipes-bsp  recipes-graphics  recipes-kernel  wic
```

BitBake then expects to find the conf/bitbake.conf file somewhere in the user-specified BBPATH. That configuration file generally has include directives to pull in any other metadata such as files specific to the architecture, the machine, the local environment, and so forth.

Only variable definitions and include directives are allowed in .conf files. Some variables directly influence BitBake's behavior. These variables might have been set from the environment depending on the environment variables previously mentioned or set in the configuration files. The "Variables Glossary" chapter presents a full list of variables.

After parsing configuration files, BitBake uses its rudimentary inheritance mechanism, which is through class files, to inherit some standard classes. BitBake parses a class when the inherit directive responsible for getting that class is encountered

The base.bbclass file is always included. Other classes that are specified in the configuration using the INHERIT variable are also included. BitBake searches for class files in a "classes" subdirectory under the paths in BBPATH in the same way as configuration files. A good way to get an idea of the configuration files and the class files used in your execution environment is to run the following BitBake command:

$ bitbake -e > mybb.log

Examining the top of the mybb.log shows you the many configuration files and class files used in your execution environment.

```
#
# INCLUDE HISTORY:
#
# /opt/yocto/poky/qemuarm/conf/bblayers.conf
# /opt/yocto/poky/meta/conf/layer.conf
# /opt/yocto/poky/meta-poky/conf/layer.conf
# /opt/yocto/poky/meta-yocto-bsp/conf/layer.conf
# conf/bitbake.conf includes:
#   /opt/yocto/poky/meta/conf/abi_version.conf
#   conf/site.conf
#   conf/auto.conf
#   /opt/yocto/poky/qemuarm/conf/local.conf
#   /opt/yocto/poky/meta/conf/multiconfig/default.conf
#   /opt/yocto/poky/meta/conf/machine/qemuarm.conf includes:
#     /opt/yocto/poky/meta/conf/machine/include/tune-cortexa15.inc includes:
#       /opt/yocto/poky/meta/conf/machine/include/arm/arch-armv7ve.inc includes:
#         /opt/yocto/poky/meta/conf/machine/include/arm/arch-armv7a.inc includes:
#           /opt/yocto/poky/meta/conf/machine/include/arm/arch-armv6.inc includes:
#             /opt/yocto/poky/meta/conf/machine/include/arm/arch-armv5-dsp.inc includes:
#               /opt/yocto/poky/meta/conf/machine/include/arm/arch-armv5.inc includes:
#                 /opt/yocto/poky/meta/conf/machine/include/arm/arch-armv4.inc includes:
#                   /opt/yocto/poky/meta/conf/machine/include/arm/arch-arm.inc
#                   /opt/yocto/poky/meta/conf/machine/include/arm/feature-arm-thumb.inc
#                 /opt/yocto/poky/meta/conf/machine/include/arm/feature-arm-vfp.inc
#           /opt/yocto/poky/meta/conf/machine/include/arm/feature-arm-neon.inc
#     /opt/yocto/poky/meta/conf/machine/include/qemu.inc
#   /opt/yocto/poky/meta/conf/machine-sdk/x86_64.conf
#   /opt/yocto/poky/meta-poky/conf/distro/poky.conf includes:
#     /opt/yocto/poky/meta-poky/conf/distro/include/poky-world-exclude.inc
#     /opt/yocto/poky/meta/conf/distro/include/no-static-libs.inc
#     /opt/yocto/poky/meta/conf/distro/include/yocto-uninative.inc
#     /opt/yocto/poky/meta/conf/distro/include/security_flags.inc
#   /opt/yocto/poky/meta/conf/distro/defaultsetup.conf includes:
#     /opt/yocto/poky/meta/conf/distro/include/default-providers.inc
#     /opt/yocto/poky/meta/conf/distro/include/default-versions.inc
#     /opt/yocto/poky/meta/conf/distro/include/default-distrovars.inc
#     /opt/yocto/poky/meta/conf/distro/include/maintainers.inc
#     /opt/yocto/poky/meta/conf/distro/include/tcmode-default.inc
#     /opt/yocto/poky/meta/conf/distro/include/tclibc-glibc.inc
#     /opt/yocto/poky/meta/conf/distro/include/uninative-flags.inc
#     /opt/yocto/poky/meta/conf/distro/include/init-manager-none.inc
#   /opt/yocto/poky/meta/conf/documentation.conf
#   /opt/yocto/poky/meta/conf/licenses.conf
#   /opt/yocto/poky/meta/conf/sanity.conf
# /opt/yocto/poky/meta/classes/base.bbclass includes:
#   /opt/yocto/poky/meta/classes/patch.bbclass includes:
#     /opt/yocto/poky/meta/classes/terminal.bbclass
#   /opt/yocto/poky/meta/classes/staging.bbclass
#   /opt/yocto/poky/meta/classes/mirrors.bbclass
#   /opt/yocto/poky/meta/classes/utils.bbclass
#   /opt/yocto/poky/meta/classes/utility-tasks.bbclass
#   /opt/yocto/poky/meta/classes/metadata_scm.bbclass
#   /opt/yocto/poky/meta/classes/logging.bbclass
# /opt/yocto/poky/meta-poky/classes/poky-sanity.bbclass
# /opt/yocto/poky/meta/classes/uninative.bbclass
# /opt/yocto/poky/meta/classes/reproducible_build.bbclass includes:
#   /opt/yocto/poky/meta/classes/reproducible_build_simple.bbclass
# /opt/yocto/poky/meta/classes/package_rpm.bbclass includes:
#   /opt/yocto/poky/meta/classes/package.bbclass includes:
#     /opt/yocto/poky/meta/classes/packagedata.bbclass
#     /opt/yocto/poky/meta/classes/chrpath.bbclass
```

```
#     /opt/yocto/poky/meta/classes/package_pkgdata.bbclass
#     /opt/yocto/poky/meta/classes/insane.bbclass
# /opt/yocto/poky/meta/classes/buildstats.bbclass
# /opt/yocto/poky/meta/classes/image-mklibs.bbclass includes:
#   /opt/yocto/poky/meta/classes/linuxloader.bbclass
# /opt/yocto/poky/meta/classes/image-prelink.bbclass
# /opt/yocto/poky/meta/classes/debian.bbclass
# /opt/yocto/poky/meta/classes/devshell.bbclass
# /opt/yocto/poky/meta/classes/sstate.bbclass
# /opt/yocto/poky/meta/classes/license.bbclass
# /opt/yocto/poky/meta/classes/remove-libtool.bbclass
# /opt/yocto/poky/meta/classes/blacklist.bbclass
# /opt/yocto/poky/meta/classes/sanity.bbclass
#
# $ABIEXTENSION [3 operations]
#   set /opt/yocto/poky/meta/conf/bitbake.conf:126
#     [_defaultval] ""
#   set /opt/yocto/poky/meta/conf/machine/include/arm/arch-arm.inc:14
#     "eabi"
#   override[class-nativesdk]:set /opt/yocto/poky/meta/conf/machine-sdk/x86_64.conf:2
#     ""
# pre-expansion value:
#   "eabi"
ABIEXTENSION="eabi"
#
# $ABIEXTENSION_class-nativesdk
#   set /opt/yocto/poky/meta/conf/machine-sdk/x86_64.conf:2
#     ""
ABIEXTENSION_class-nativesdk=""
#
# $ALLOW_EMPTY [5 operations]
#   set /opt/yocto/poky/meta/conf/documentation.conf:62
#     [doc] "Specifies if an output package should still be produced if it is empty."
#   override[defaultpkgname-dbg]:set /opt/yocto/poky/meta/conf/bitbake.conf:329
#     "1"
#   override[defaultpkgname-dbg]:rename from ALLOW_EMPTY_${PN}-dbg data.py:104 [expandKeys]
#     "1"
#   override[defaultpkgname-dev]:set /opt/yocto/poky/meta/conf/bitbake.conf:320
#     "1"
#   override[defaultpkgname-dev]:rename from ALLOW_EMPTY_${PN}-dev data.py:104 [expandKeys]
#     "1"
# pre-expansion value:
#   "None"
#
# $ALLOW_EMPTY_defaultpkgname-dbg [2 operations]
#   set /opt/yocto/poky/meta/conf/bitbake.conf:329
#     "1"
#   rename from ALLOW_EMPTY_${PN}-dbg data.py:104 [expandKeys]
#     "1"
# pre-expansion value:
#   "1"
ALLOW_EMPTY_defaultpkgname-dbg="1"
```

```
# $ALLOW_EMPTY_defaultpkgname-dev [2 operations]
#   set /opt/yocto/poky/meta/conf/bitbake.conf:320
#     "1"
#   rename from ALLOW_EMPTY_${PN}-dev data.py:104 [expandKeys]
#     "1"
# pre-expansion value:
#   "1"
ALLOW_EMPTY_defaultpkgname-dev="1"
#
# $ALL_MULTILIB_PACKAGE_ARCHS
#   set /opt/yocto/poky/meta/classes/package.bbclass:53
#     "${@all_multilib_tune_values(d, 'PACKAGE_ARCHS')}"
ALL_MULTILIB_PACKAGE_ARCHS="all any noarch armv5hf-vfp armv5thf-vfp armv5ehf-vfp armv5tehf-vfp armv6hf-vfp armv6thf-vfp armv7ahf-vfp armv7at2hf-vfp armv
7vehf-vfp armv7vet2hf-vfp armv7vehf-neon armv7vet2hf-neon qemuarm"
#
# $ALL_QA
#   set /opt/yocto/poky/meta/classes/insane.bbclass:49
#     "${WARN_QA} ${ERROR_QA}"
ALL_QA=" libdir xorg-driver-abi            textrel incompatible-license files-invalid            infodir build-deps src-uri-bad symlink-to-sysroot mul
tilib            invalid-packageconfig host-user-contaminated uppercase-pn patch-fuzz            mime mime-xdg unlisted-pkg-lics unhandled-features-ch
eck            missing-update-alternatives native-last            dev-so debug-deps dev-deps debug-files arch pkgconfig la            perms dep-cmp
pkgvarcheck perm-config perm-line perm-link            split-strip packages-list pkgv-undefined var-undefined            version-going-backwards expan
ded-d invalid-chars            license-checksum dev-elf file-rdeps configure-unsafe            configure-gettext perllocalpod shebang-size
 already-stripped installed-vs-shipped ldflags compile-host-path            install-host-path pn-overrides unknown-configure-option            useless
-rpaths rpaths staticdev            "
#
```

```
#
# $ALTERNATIVE
#   set /opt/yocto/poky/meta/conf/documentation.conf:63
#     [doc] "Lists commands in a package that need an alternative binary naming scheme."
#
# $ALTERNATIVE_LINK_NAME
#   set /opt/yocto/poky/meta/conf/documentation.conf:64
#     [doc] "Used by the alternatives system to map duplicated commands to actual locations."
#
# $ALTERNATIVE_PRIORITY
#   set /opt/yocto/poky/meta/conf/documentation.conf:65
#     [doc] "Used by the alternatives system to create default priorities for duplicated commands."
#
# $ALTERNATIVE_TARGET
#   set /opt/yocto/poky/meta/conf/documentation.conf:66
#     [doc] "Used by the alternatives system to create default link locations for duplicated commands."
#
# $ANY_OF_COMBINED_FEATURES
#   set /opt/yocto/poky/meta/conf/documentation.conf:67
#     [doc] "When a recipe inherits the features_check class, at least one item in this variable must be included in COMBINED_FEATURES."
#
# $ANY_OF_DISTRO_FEATURES
#   set /opt/yocto/poky/meta/conf/documentation.conf:68
#     [doc] "When a recipe inherits the features_check class, at least one item in this variable must be included in DISTRO_FEATURES."
#
# $ANY_OF_MACHINE_FEATURES
#   set /opt/yocto/poky/meta/conf/documentation.conf:69
#     [doc] "When a recipe inherits the features_check class, at least one item in this variable must be included in MACHINE_FEATURES."
#
# $APACHE_MIRROR
#   set /opt/yocto/poky/meta/conf/bitbake.conf:637
#     "https://archive.apache.org/dist"
APACHE_MIRROR="https://archive.apache.org/dist"
#
# $AR [2 operations]
#   exported /opt/yocto/poky/meta/conf/bitbake.conf:523
#     [export] "1"
#   set /opt/yocto/poky/meta/conf/bitbake.conf:523
#     "${HOST_PREFIX}gcc-ar"
# pre-expansion value:
#   "${HOST_PREFIX}gcc-ar"
export AR="arm-poky-linux-gnueabi-gcc-ar"
#
# $ARCH_DEFAULT_KERNELIMAGETYPE [3 operations]
#   set /opt/yocto/poky/meta/conf/distro/include/default-distrovars.inc:41
#     "zImage"
#   override[x86]:set /opt/yocto/poky/meta/conf/distro/include/default-distrovars.inc:42
#     "bzImage"
#   override[x86-64]:set /opt/yocto/poky/meta/conf/distro/include/default-distrovars.inc:43
#     "bzImage"
# pre-expansion value:
#   "zImage"
ARCH_DEFAULT_KERNELIMAGETYPE="zImage"
#
# $ARCH_DEFAULT_KERNELIMAGETYPE_x86
#   set /opt/yocto/poky/meta/conf/distro/include/default-distrovars.inc:42
#     "bzImage"
ARCH_DEFAULT_KERNELIMAGETYPE_x86="bzImage"
```

…

Most of environment variables are set in meta/conf/bitbake.conf

Locating and parsing recipes

During the configuration phase, BitBake will have set BBFILES. BitBake now uses it to contruct a list of recipes to parse, along with any append files (.bbappend) to apply. BBFILES is a space-separated list of available files and supports wildcards. An example would be:

```
BBFILES = "/path/to/bbfiles/*.bb /path/to/appends/*.bbappend"
```

BitBake parses each recipe and append file located with BBFILES and stores the values of various variables into the datastore.

One common convention is to use the recipe filename to define pieces of metadata. For example, in bitbake.conf the recipe name and version set PN and PV: PV = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE'),d)[1] or '1.0'}" PN = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE'),d)[0] or 'defaultpkgname'}" In this example, a recipe called "something_1.2.3.bb" sets PN to "something" and PV to "1.2.3".

BitBake does not need all of this information. It only needs a small subset of the information to make decisions about the recipe. Consequently, BitBake caches the values in which it is interested and does not store the rest of the information. Experience has shown it is faster to re-parse the metadata than to try and write it out to the disk and then reload it.

Where possible, subsequent BitBake commands reuse this cache of recipe information. The validity of this cache is determined by first computing a checksum of the base configuration data (see BB_HASHCONFIG_WHITELIST) and then checking if the checksum matches. If that checksum matches what is in the cache and the recipe and class files have not changed, Bitbake is able to use the cache. BitBake then reloads the cached information about the recipe instead of reparsing it from scratch.

Preferences and Providers

Assuming BitBake has been instructed to execute a target and that all the recipe files have been parsed. BitBake starts to figure out how to build the target. BitBake starts by looking through the PROVIDES set in recipe files. The default PROVIDES for a recipe is its name (PN), however, a recipe can provide multiple things.

As an example of adding an extra provider, suppose a recipe named foo_1.0.bb contained the following:

PROVIDES += "virtual/bar_1.0"

The recipe now provides both "foo_1.0" and "virtual/bar_1.0". The "virtual/" namespace is often used to denote cases where multiple providers are expected with the user choosing between them. Kernels and toolchain components are common cases of this in OpenEmbedded.

Sometimes a target might have multiple providers. A common example is "virtual/kernel", which is provided by each kernel recipe. Each machine often selects the best kernel provider by using a line similar to the following in the machine configuration file:

PREFERRED_PROVIDER_virtual/kernel = "linux-yocto"

The default PREFERRED_PROVIDER is the provider with the same name as the target. Bitbake iterates through each target it needs to build and resolves them and their dependencies using this process.

Understanding how providers are chosen is made complicated by the fact that multiple versions might exist. BitBake defaults to the highest version of a provider. Version comparisons are made using the same method as Debian. You can use the PREFERRED_VERSION variable to specify a particular version. You can influence the order by using the DEFAULT_PREFERENCE variable. By default, files have a preference of "0". Setting the DEFAULT_PREFERENCE to "-1" makes the recipe unlikely to be used unless it is explicitly referenced. Setting the DEFAULT_PREFERENCE to "1" makes it likely the recipe is used. PREFERRED_VERSION overrides any DEFAULT_PREFERENCE setting. DEFAULT_PREFERENCE is often used to mark newer and more experimental recipe versions until they have undergone sufficient testing to be considered stable.

When there are multiple “versions” of a given recipe, BitBake defaults to selecting the most recent version, unless otherwise specified. If the recipe in question has a DEFAULT_PREFERENCE set lower than the other recipes (default is 0), then it will not be selected. This allows the person or persons maintaining the repository of recipe files to specify their preference for the default selected version. In addition, the user can specify their preferred version.

If the first recipe is named a_1.1.bb, then the PN variable will be set to “a”, and the PV variable will be set to 1.1. If we then have a recipe named a_1.2.bb, BitBake will choose 1.2 by default. However, if we define the following variable in a .conf file that BitBake parses, we can change that. PREFERRED_VERSION_a = "1.1" In summary, BitBake has created a list of providers, which is prioritized, for each target.

Dependencies

Each target BitBake builds consists of multiple tasks such as fetch, unpack, patch, configure, and compile. For best performance on multi-core systems, BitBake considers each task as an independent entity with its own set of dependencies.

Dependencies are defined through several variables. You can find information about variables BitBake uses in the Variables Glossary near the end of this manual. At a basic level, it is sufficient to know that BitBake uses the DEPENDS and RDEPENDS variables when calculating dependencies.

The task list

Based on the generated list of providers and the dependency information. BitBake can now calculate exactly what tasks it needs to run and in what order it needs to run them. The “Executing Tasks” section has more information on how BitBake chooses which task to execute next.

The build now starts with BitBake forking off threads up to the limit set in the BB_NUMBER_THREADS variable. BitBake continues to fork threads as long as there are tasks ready to run, those tasks have all their dependencies met, and the thread threshold has not been exceeded.

It is worth noting that you can greatly speed up the build time by properly setting the BB_NUMBER_THREADS variable.

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.

The exact format of the stamps is partly configurable. In modern versions of BitBake, a hash is appended to the stamp so that if the configuration changes, the stamp becomes invalid and the task is automatically rerun. This hash, or signature used, is governed by the signature policy that is configured (see the "Checksums (Signatures)" section for information). It is also possible to append extra metadata to the stamp using the "stamp-extra-info" task flag. For example, OpenEmbedded uses this flag to make some tasks machine-specific.

```
tpthinh@ubuntu:/opt/yocto/poky/qemuarm/tmp/stamps/x86_64-linux/automake-native$ ls
1.16.3-r0.do_compile.7154148feddc3dc23adeaedbb40da9663095c789e5806a8a3a946ed747104897
1.16.3-r0.do_compile.sigdata.7154148feddc3dc23adeaedbb40da9663095c789e5806a8a3a946ed747104897
1.16.3-r0.do_configure.cc3ba44917a563c1a694a6ebdb531125ea6b46934c62243f9f78cd312d133740
1.16.3-r0.do_configure.sigdata.cc3ba44917a563c1a694a6ebdb531125ea6b46934c62243f9f78cd312d133740
1.16.3-r0.do_deploy_source_date_epoch.86b68a334be20123f045575312bb1edcae58da0842d1269fff6d64ab7254069f
1.16.3-r0.do_deploy_source_date_epoch.sigdata.86b68a334be20123f045575312bb1edcae58da0842d1269fff6d64ab7254069f
1.16.3-r0.do_fetch.955f77a0f0eb9a7d6942d2b6fcd812dc4780a3d13cc0d605031c67a906dfd416
1.16.3-r0.do_fetch.sigdata.955f77a0f0eb9a7d6942d2b6fcd812dc4780a3d13cc0d605031c67a906dfd416
1.16.3-r0.do_install.2774662890b15d166e83dad5570f86f0a4df5a0bb869102eb61ec4156ddeca33
1.16.3-r0.do_install.sigdata.2774662890b15d166e83dad5570f86f0a4df5a0bb869102eb61ec4156ddeca33
1.16.3-r0.do_patch.94e8444aa1a7781bfb97118670158854dd2175f53f23b3d056485e43b9a1af90
1.16.3-r0.do_patch.sigdata.94e8444aa1a7781bfb97118670158854dd2175f53f23b3d056485e43b9a1af90
1.16.3-r0.do_populate_sysroot.86383bcc63235ff421a3c780e3549b99103cfef78fe271f0912e127dd51e8055
1.16.3-r0.do_populate_sysroot.sigdata.86383bcc63235ff421a3c780e3549b99103cfef78fe271f0912e127dd51e8055
1.16.3-r0.do_prepare_recipe_sysroot.086288690bf03cfd0300b769b619bc662288e2fa953ed2d84ee663241efb2bdc
1.16.3-r0.do_prepare_recipe_sysroot.sigdata.086288690bf03cfd0300b769b619bc662288e2fa953ed2d84ee663241efb2bdc
1.16.3-r0.do_unpack.7cb746dccaf4e3ee00925b28c65ebc44e23e9e69325641b5dd5303c9b0ea3d8c
1.16.3-r0.do_unpack.sigdata.7cb746dccaf4e3ee00925b28c65ebc44e23e9e69325641b5dd5303c9b0ea3d8c
tpthinh@ubuntu:/opt/yocto/poky/qemuarm/tmp/stamps/x86_64-linux/automake-native$
```

```
tpthinh@ubuntu:/opt/yocto/poky/qemuarm/tmp/stamps/x86_64-linux/automake-native$ vim 1.16.3-r0.do_fetch.sigdata.955f77a0f0eb9a7d694
2d2b6fcd812dc4780a3d13cc0d605031c67a906dfd416
```

Syntax and Operators

Basic syntax

Basic variable setting

The following example sets VARIABLE to "value". This assignment occurs immediately as the statement is parsed. It is a "hard" assignment.

VARIABLE = "value"

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

VARIABLE = " value"

VARIABLE = "value "

Setting VARIABLE to "" sets it to an empty string, while setting the variable to " " sets it to a blank space (i.e. these are not the same values).

VARIABLE = ""

VARIABLE = " "

Variable Expansion

BitBake supports variables referencing one another's contents using a syntax that is similar to shell scripting. Following is an example that results in A containing "aval" and B evaluating to "preavalpost" based on that current value of A.

```
A = "aval"
B = "pre${A}post"
```

You should realize that whenever B is referenced, its evaluation will depend on the state of A at that time. Thus, later evaluations of B in the previous example could result in different values depending on the value of A.

Setting a default value (?=)

You can use the "?=" operator to achieve a "softer" assignment for a variable. This type of assignment allows you to define a variable if it is undefined when the statement is parsed, but to leave the value alone if the variable has a value. Here is an example:

A ?= "aval"

If A is set at the time this statement is parsed, the variable retains its value. However, if A is not set, the variable is set to "aval".

Note This assignment is immediate. Consequently, if multiple "?=" assignments to a single variable exist, the first of those ends up getting used.

Setting a weak default value (??=)

It is possible to use a "weaker" assignment than in the previous section by using the "??=" operator. This assignment behaves identical to "?=" except that the assignment is made at the end of the parsing process rather than immediately. Consequently, when multiple "??=" assignments exist, the last one is used. Also, any "=" or "?=" assignment will override the value set with "??=". Here is an example:

A ??= "somevalue"

A ??= "someothervalue"

If A is set before the above statements are parsed, the variable retains its value. If A is not set, the variable is set to "someothervalue". Again, this assignment is a "lazy" or "weak" assignment because it does not occur until the end of the parsing process.

Immediate variable expansion (:=)

The ":=" operator results in a variable's contents being expanded immediately, rather than when variable is actually used.

```
T = "123"
A := "${B} ${A} test ${T}"
T = "456"
B = "${T} bval"
C = "cval"
C := "${C}append"
```

In this example, A contains “test 123” because ${B} and ${A} at the time of parsing are undefined, which leaves “test 123”. And, the variable C contains “cvalappend” since ${C} immediately expands to “cval”.

Appending (+=) and prepending (=+) with Spaces

Appending and prepending values is common and can be accomplished using the "+=" and "=+" operators. These operators insert a space between the current value and prepended or appended value. Here are some examples:

```
B = "bval"
B += "additionaldata"
C = "cval"
C =+ "test"
```

The variable B contains “bval additionaldata” and C contains “test cval”.

Appending (.=) and Prepending (=.) without spaces

If you want to append or prepend values without an inserted space, use the “.=” and “=.” Operators. Here are some examples:

```
B = "bval"
B .= "additionaldata"
C = "cval"
C =. "test"
```

The variable B contains "bvaladditionaldata" and C contains "testcval".

Appending and Prepending (Override Style Syntax)

You can also append and prepend a variable’s value using an override style syntax. When you use this syntax, no spaces are inserted. Here are some examples:

```
B = "bval"
B_append = " additional data"
C = "cval"
C_prepend = "additional data "
D = "dval"
D_append = "additional data"
```

The variable B becomes “bval additional data” and c becomes “additional data cval”. The variable D becomes “additional data cval”. The variable D becomes “dvaladditional data”

Note: You must control all spacing when you use the override syntax.

The operators “_append” and “_prepend” differ from the operators “.=” and “=.” In that they are deferred until after parsing completes rather than being immediately applied.

Removal (Override Style Syntax)

You can remove values from lists using the removal override style syntax. Specifying a value for removal causes all occurrences of that value to be removed from the variable.

When you use this syntax, BitBake expects one or more strings. Surrounding spaces are removed as well. Here is an example:

```
F00 = "123 456 789 123456 123 456 123 456"
F00_remove = "123"
F00_remove = "456"
F002 = "abc def ghi abcdef abc def abc def"
F002_remove = "abc def"
```

The variable F00 becomes “789 123456” and F002 becomes “ghi abcdef”.

Variable Flag Syntax

Variable flags are BitBake’s implementation of variable properties or attributes. It is a way of tagging extra information onto a variable.

You can define, append, and prepend values to variable flags. All the standard syntax operations previously mentioned work for variable flags except for override style syntax (i.e. _prepend, _append, and _remove).

Here are some examples showing how to set variable flags:

F00[a] = “a”

F00[b]= “123”

F00[a] += “456”

The variable FOO has two flags: a and b. The flags are immediately set to "abc" and "123", respectively. The a flag becomes "abc456".

Inline Python Variable Expansion

You can use inline Python variable expansion to set variables. Here is an example:

```
DATE = "${@time.strftime('%Y%m%d',time.gmtime())}"
```

This example results in the DATE variable becoming the current date.

Providing Pathnames

When specifying pathnames for use with BitBake, do not use the tilde (“~”) characters as a shortcut for your home directory. Doing so might cause BitBake to not recognize the path since BitBake does not expand this character in the same way a shell would.

Instead, provide a fuller path as the following example illustrates:

BBLAYERS ?= “ \

/home/tpthinh/Programming \

“

Conditional Syntax (Overrides)

BitBake uses OVERRIDES to control what variables are overridden after BitBake parses recipes and configuration files. This section describes how you can use OVERRIDES as conditional metadata, talks about key expansion in relationship to OVERRIDES, and provides some examples to help with understanding.

Conditional Metadata

You can use OVERRIDES to conditionally select a specific version of a variable and to conditionally append or prepend the value of a variable.

- Selecting a Variable: The OVERRIDES variable is a colon-character-separated list that contains items for which you want to satisfy conditions. Thus, if you have a variable that is conditional on “arm”, and “arm” is in OVERRIDES, then the “arm”-specific version of the variable is used rather than the non-conditional version. Here is an example:

```
OVERRIDES = "architecture:os:machine"
TEST = "default"
TEST_os = "osspecific"
TEST_nooverride = "othercondvalue"
```

In this example, the OVERRIDES variable lists three overrides: "architecture", "os", and "machine". The variable TEST by itself has a default value of "default". You select the os-specific version of the TEST variable by appending the "os" override to the variable (i.e.TEST_os).

- Appending and Prepending: BitBake also supports append and prepend operations to variable values based on whether a specific item is listed in OVERRIDES. Here is an example:

```
DEPENDS = "glibc ncurses"
OVERRIDES = "machine:local"

DEPENDS_append_machine = "libmad"
```

In this example, DEPENDS becomes “glibc ncurses libmad”

Key Expansion

Key expansion happens when the BitBake datastore is finalized just before BitBake expands overrides. To better understand this, consider the following example:

```
A${B} = "X"
B = "2"
A2 = "Y"
```

In this case, after all the parsing is complete, and before any overrides are handled, BitBake expands ${B} into "2". This expansion causes A2, which was set to "Y" before the expansion, to become "X".

Examples

Despite the previous explanations that show the different forms of variable definitions, it can be hard to work out exactly what happens when variable operators, conditional overrides, and unconditional overrides are combined. This section presents some common scenarios along with explanations for variable interactions that typically confuses users.

There is often confusion concerning the order in which overrides and various "append" operators take effect. Recall that an append or prepend operation using "_append" and "_prepend" does not result in an immediate assignment as would "+=", ".=", "=+", or "=.". Consider the following example:

```
OVERRIDES = "foo"
A = "Z"
A_foo_append = "X"
```

For this case, A is unconditionally set to "Z" and "X" is unconditionally and immediately appended to the variable A_foo. Because overrides have not been applied yet, A_foo is set to "X" due to the append and A simply equals "Z".

Applying overrides, however, changes things. Since "foo" is listed in OVERRIDES, the conditional variable A is replaced with the "foo" version, which is equal to "X". So effectively, A_foo replaces A.

This next example changes the order of the override and the append:

```
OVERRIDES = "foo"
A = "Z"
A_append_foo = "X"
```

For this case, before overrides are handled, A is set to "Z" and A_append_foo is set to "X". Once the override for "foo" is applied, however, A gets appended with "X". Consequently, A becomes "ZX". Notice that spaces are not appended.

This next example has the order of the appends and overrides reversed back as in the first example:

```
OVERRIDES = "foo"
A = "Y"
A_foo_append = "Z"
A_foo_append += "X"
```

For this case, before any overrides are resolved, A is set to "Y" using an immediate assignment. After this immediate assignment, A_foo is set to "Z", and then further appended with "X" leaving the variable set to "Z X". Finally, applying the override for "foo" results in the conditional variable A becoming "Z X" (i.e. A is replaced with A_foo).

This final example mixes in some varying operators:

```
A = "1"
A_append = "2"
A_append = "3"
A += "4"
A .= "5"
```

For this case, the type of append operators are affecting the order of assignments as BitBake passes through the code multiple times. Initially, A is set to "1 45" because of the three statements that use immediate operators. After these assignments are made, BitBake applies the _append operations. Those operations result in A becoming "1 4523".

Sharing Functionality

BitBake allows for metadata sharing through include files (.inc) and class files (.bbclass). For example, suppose you have a piece of common functionality such as a task definition that you want to share between more than one recipe. In this case, creating a .bbclass file that contains the common functionality and then using the inherit directive in your recipes to inherit the class would be a common way to share the task.

This section presents the mechanisms BitBake provides to allow you to share functionality between recipes. Specifically, the mechanisms include include, inherit, INHERIT, and require directives.

Locating include and class files

BitBake uses the BBPATH variable to locate needed include and class files. The BBPATH variable is analogous to the environment variable PATH.

In order for include and class files to be found by BitBake, they need to be located in a “classes” subdirectory that can be found in BBPATH.

Inherit Directive

When writing a recipe or class file, you can use the inherit directive to inherit the functionality of a class (.bbclass). BitBake only supports this directive when used within recipe and class files (i.e. .bb and .bbclass).

The inherit directive is a rudimentary means of specifying what classes of functionality your recipes require. For example, you can easily abstract out the tasks involved in building a package that uses Autoconf and Automake and put those tasks into a class file that can be used by your recipe.

As an example, your recipes could use the following directive to inherit an autotools.bbclass file. The class file would contain common functionality for using Autotools that could be shared across recipes:

inherit autotools

In this case, BitBake would search for the directory classes/autotools.bbclass in BBPATH.

Note

You can override any values and functions of the inherited class within your recipe by doing so after the "inherit" statement.

include Directive

BitBake understands the include directive. This directive causes BitBake to parse whatever file you specify, and to insert that file at that location. The directive is much like its equivalent in Make except that if the path specified on the include line is a relative path, BitBake locates the first file it can find within BBPATH.

As an example, suppose you needed a recipe to include some self-test definitions:

Include test_defs.inc

Note: The include directive does not produce an error when the file cannot be found. Consequently, it is recommended that if the file you are including is expected to exist, you should use require instead of include. Doing so makes sure that an error is produced if the file cannot be found.

require Directive

BitBake understands the require directive. This directive behaves just like that include directive with the exception that BitBake raises a parsing error if the file to be included cannot be found. Thus, any file you require is inserted into the file that is being parsed at the location of the directive.

Similar to how BitBake handles include, if the path specified on the require line is a relative path, BitBake locates the first file it can find within BBPATH.

As an example, suppose you have two versions of a recipe (e.g foo_1.2.2.bb and foo_2.0.0.bb) where each version contains some identical functionality that could be shared. You could create an include file named foo.inc that contains the common definitions needed to build "foo". You need to be sure foo.inc is located in the same directory as your two recipe files as well. Once these conditions are set up, you can share the functionality using a require directive from within each recipe:

require foo.inc

INHERIT Configuration Directive

When creating a configuration file (.conf), you can use the INHERIT directive to inherit a class. BitBake only supports this directive when used within a configuration file.

As an example, suppose you needed to inherit a class file called abc.bbclass from a configuration file as follows:

INHERIT += "abc"

This configuration directive causes the named class to be inherited at the point of the directive during parsing. As with the inherit directive, the .bbclass file must be located in a "classes" subdirectory in one of the directories specified in BBPATH.

Functions

As with most languages, functions are the building blocks that are used to build up operations into tasks. BitBake supports these types of functions:

- Shell Functions: Functions written in shell script and executed either directly as functions, tasks, or both. They can also be called by other shell functions.
- BitBake Style Python Functions: Functions written in Python and executed by BitBake or other Python functions using bb.build.exec_func().
- Python Functions: Functions written in Python and executed by Python.
- Anonymous Python Functions: Python functions executed automatically during parsing.

Regardless of the type of function, you can only define them in class (.bbclass) and recipe (.bb or inc) files.

Shell functions

Functions written in shell script and executed either directly as functions, tasks, or both. They can also be called by other shell function definition:

some_function () {

    echo “Hello World”

}

When you create these types of functions in your recipe or class files, you need to follow the shell programming rules. The scripts are executed by /bin/sh, which may not be a bash shell but might be something such as dash. You should not use Bash-specific script (bashisms).

BitBake Style Python Functions

These functions are written in Python and executed by BitBake or other Python functions using bb.build.exec_func().

An example BitBake function is:

```
python some_python_function () {
    d.setVar("TEXT", "Hello World")
    print d.getVar("TEXT", True)
}
```

Because the Python “bb” and “os” modules are already imported, you do not need to import these modules. Also in these types of functions, the datastore (“d”) is a global variable and its always automatically available.

Python Functions

These functions are written in Python and are executed by other Python code. Examples of Python functions are utility functions that you intend to call from in-line Python or from within other Python functions. Here is an example:

```
def get_depends(d):
    if d.getVar('SOMECONDITION', True):
        return "dependencywithcond"
    else:
        return "dependency"
SOMECONDITION = "1"
DEPENDS = "${@get_depends(d)}"
```

This would result in DEPENDS containing dependencywithcond

Here are some things to know about Python functions:

- Python functions can take parameters.
- The BitBake datastore is not automatically available. Consequently, you must pass it in as a parameter to the function.
- The “bb” and “os” Python modules are automatically available. You do not need to import them.

Anonymous Python Functions

Sometimes it is useful to run some code during parsing to set variables or to perform other operations programmatically. To do this, you can define an anonymous Python function. Here is an example that conditionally sets a variable based on the value of another variable:

```
python __anonymous () {
    if d.getVar('SOMEVAR', True) == 'value':
        d.setVar('ANOTHERVAR', 'value2')
}
```

The "__anonymous" function name is optional, so the following example is functionally equivalent to the above:

```
python () {
    if d.getVar('SOMEVAR', True) == 'value':
        d.setVar('ANOTHERVAR', 'value2')
}
```

Because unlike other Python functions anonymous Python functions are executed during parsing, the "d" variable within an anonymous Python function represents the datastore for the entire recipe. Consequently, you can set variable values here and those values can be picked up by other functions.

Flexible Inheritance for Class Functions

Through coding techniques and the use of EXPORT_FUNCTIONS, BitBake supports exporting a function from a class such that the class function appears as the default implementation of the function, but can still be called if a recipe inheriting the class needs to define its own version of the function.

To understand the benefits of this feature, consider the basic scenario where a class defines a task function and your recipe inherits the class. In this basic scenario, your recipe inherits the task function as defined in the class. If desired, your recipe can add to the start and end of the function by using the “_prepend” or “_append” operations respectively, or it can redefine the function completely. However, if it redefines the function, there is no means for it to call the class version of the function.

EXPORT_FUNCTIONS provides a mechanism that enables the recipe's version of the function to call the original version of the function.

To make use of this technique, you need the following things in place:

- The class needs to define the function as follows:

<classname>_<functionname>

For example, if you have a class file bar.bbclass and a function named do_foo, the class must define the function as follows:

bar_do_foo

- The class needs to contain the EXPORT_FUNCTIONS statement as follows:

EXPORT_FUNCTIONS <functionname>

For example, continuing with the same example, the statement in the bar.bbclass would be as follows:

EXPORT_FUNCTIONS do_foo

- You need to call the function appropriately from within your recipe. Continuing with the same example, if your recipe needs to call the class version of the function, it should call bar_do_foo. Assuming do_foo was a shell function and EXPORT_FUNCTIONS was used as above, the recipe's function could conditionally call the class version of the function as follows:

```
do_foo() {
        if [ somecondition ] ; then
                bar_do_foo
        else
                # Do something else
        fi
}
```

To call your modified version of the function as defined in your recipe, call it as do_foo.

With these conditions met, your single recipe can freely choose between the original function as defined in the class file and the modified function in your recipe. If you do not set up these conditions, you are limited to using one function or the other.

Tasks

Tasks are BitBake execution units that originate as functions and make up the steps that BitBake needs to run for given recipe. Tasks are only supported in recipe (.bb or .inc) and class (.bbclass) files. By convention, task names begin with the string "do_".

Here is an example of a task that prints out the date:

```
python do_printdate () {
    import time
    print time.strftime('%Y%m%d', time.gmtime())
}
addtask printdate after do_fetch before do_build
```

Promoting a Function to a Task

Any function can be promoted to a task by applying the addtask command. The addtask command also describes inter-task dependencies. Here is the function from the previous section but with the addtask command promoting it to a task and defining some dependencies:

```
python do_printdate () {
    import time
    print time.strftime('%Y%m%d', time.gmtime())
}
addtask printdate after do_fetch before do_build
```

In the example, the function is defined and then promoted as a task. The do_printdate task becomes a dependency of the do_build task, which is the default task. And, the do_printdate task is dependent upon the do_fetch task. Execution of the do_build task results in the do_printdate task running first.

Deleting a Task

As well as being able to add tasks, tasks can also be deleted. This is done simply with deltask command. For example, to delete the example task used in the previous sections, you would use:

deltask printdate

Passing Information into the Build Task Environment

When running a task, BitBake tightly controls the execution environment of the build tasks to make sure unwanted contamination from the build machine cannot influence the build. Consequently, if you do want something to get passed into the build task environment, you must take these two steps:

- Tell BitBake to load what you want from the environment into the datastore. You can do so through the BB_ENV_EXTRAWHITE variable. For example, assume you want to prevent the build system from accessing your $HOME/.ccache directory. The following command tells BitBake to load CCACHE_DIR from the environment into the datastore:

  ```
  export BB_ENV_EXTRAWHITE="$BB_ENV_EXTRAWHITE CCACHE_DIR"
  ```

- Tell BitBake to export what you have loaded into the datastore to the task environment of every running task. Loading something from the environment into the datastore (previous step) only makes it available in the datastore. To export it to the task environment of every running task, use a command similar to the following in your local configuration file local.conf or your distribution configuration file:

  ```
  export CCACHE_DIR
  ```

  Note: A side effect of the previous steps is that BitBake records the variable as a dependency of the build process in things like the setscene checksums. If doing so results in unnecessary

rebuilds of tasks, you can whitelist the variable so that the setscene code ignores the dependency when it creates checksums.

Sometimes, it is useful to be able to obtain information from the original execution environment. BitBake saves a copy of the original environment into a special variable named BB_ORIGENV.

The BB_ORIGENV variable returns a datastore object that can be queried using the standard datastore operators such as getVar(). The datastore object is useful, for example, to find the original DISPAY variable. Here is an example.

```
BB_ORIGENV - add example?

origenv = d.getVar("BB_ORIGENV", False)
bar = origenv.getVar("BAR", False)
```

The previous example returns BAR from the original execution environment.

By default, BitBake cleans the environment to include only those things exported or listed in its whitelist to ensure that the build environment is reproducible and consistent.

Variable Flags

Variable flags (varflags) help control a task's functionality and dependencies. BitBake reads and writes varflags to the datastore using the following command forms:

<variable> = d.getVarFlags("<variable>")

self.d.setVarFlags("F00", {"func":True})

When working with varflags, the same syntax, with the exception of overrides, applies. In other words, you can set, append, and prepend varflags just like variables. See the "Variable Flag Syntax" section for details.

Hello World Example

The simplest example commonly used to demonstrate any new programming language or tool is the "Hello World [http://en.wikipedia.org/wiki/Hello_world_program]" example. This appendix demonstrates, in tutorial form, Hello World within the context of BitBake. The tutorial describes how to create a new project and the applicable metadata files necessary to allow BitBake to build it.

Obtain BitBake

Git clone https://github.com/openembedded/bitbake.git

See the "Obtaining BitBake" section for information on how to obtain BitBake. Once you have the source code on your machine, the BitBake directory appears as follows:

```
$ ls -al
total 100
drwxrwxr-x. 9 wmat wmat  4096 Jan 31 13:44 .
drwxrwxr-x. 3 wmat wmat  4096 Feb  4 10:45 ..
-rw-rw-r--. 1 wmat wmat   365 Nov 26 04:55 AUTHORS
drwxrwxr-x. 2 wmat wmat  4096 Nov 26 04:55 bin
drwxrwxr-x. 4 wmat wmat  4096 Jan 31 13:44 build
-rw-rw-r--. 1 wmat wmat 16501 Nov 26 04:55 ChangeLog
drwxrwxr-x. 2 wmat wmat  4096 Nov 26 04:55 classes
drwxrwxr-x. 2 wmat wmat  4096 Nov 26 04:55 conf
drwxrwxr-x. 3 wmat wmat  4096 Nov 26 04:55 contrib
-rw-rw-r--. 1 wmat wmat 17987 Nov 26 04:55 COPYING
drwxrwxr-x. 3 wmat wmat  4096 Nov 26 04:55 doc
-rw-rw-r--. 1 wmat wmat    69 Nov 26 04:55 .gitignore
-rw-rw-r--. 1 wmat wmat   849 Nov 26 04:55 HEADER
drwxrwxr-x. 5 wmat wmat  4096 Jan 31 13:44 lib
-rw-rw-r--. 1 wmat wmat   195 Nov 26 04:55 MANIFEST.in
-rwxrwxr-x. 1 wmat wmat  3195 Jan 31 11:57 setup.py
-rw-rw-r--. 1 wmat wmat  2887 Nov 26 04:55 TODO
```

At this point, you should have BitBake cloned to a directory that matches the previous listing except for dates and user names.

Setting up the BitBake Environment

```
tpthinh@ubuntu:~/Programming/Yocto/bitbake$ bin/bitbake --version
BitBake Build Tool Core version 1.51.0
```

```
tpthinh@ubuntu:~/Programming/Yocto/bitbake$ echo $PATH
/home/tpthinh/Programming/Yocto/bitbake/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin:/usr/games:/usr/local/games:/snap/bin
tpthinh@ubuntu:~/Programming/Yocto/bitbake$ which bitbake
/home/tpthinh/Programming/Yocto/bitbake/bin/bitbake
tpthinh@ubuntu:~/Programming/Yocto/bitbake$
```

The Hello World Example

The overall goal of this exercise is to build a complete “Hello World” example utilizing task and layer concepts. Because this is how modern projects such as OpenEmbedded and the Yocto Project utilize BitBake, the example provides an excellent starting point for understanding BitBake.

To help you understand how to use BitBake to build targets, the example starts with nothing but the bitbake command, which causes BitBake to fail and report problems. The example progresses by adding pieces to the build to eventually conclude with a working, minimal “Hello World” example.

While every attempt is made to explain what is happening during the example, the descriptions cannot cover everything. You can find further information throughout this manual. Also, you can actively participate in the http://lists.openembedded.org/mailman/listinfo/bitbake-devel discussion mailing list about the BitBake build tool.

- This example was inspired by and drew heavily from these sources: • Mailing List post - The BitBake equivalent of "Hello, World!" [http://www.mail-archive.com/yocto@yoctoproject.org/msg09379.html]

- Hambedded Linux blog post - From Bitbake Hello World to an Image [https://web.archive.org/web/20150325165911/http://hambedded.org/blog/2012/11/24/from-bitbake-hello-world-to-an-image/]

As stated earlier, the goal of this example is to eventually compile “Hello World”. However, it is unknown what BitBake needs and what you have to provide in order to achieve that goal. Recall that BitBake utilizes three types of metadata files: Configuration Files, Classes, and Recipes. But where do they go? How does BitBake find them? BitBake’s error message helps you answer these types of questions and helps you better understand exactly what is going on.

Following is the complete “Hello World” example.

Create a Project Directory: First, set up a directory for the “Hello World” project. Here is how you can do so in your home directory:

$ mkdir ~/hello

$ cd ~/hello

This is the directory that BitBake will use to do all of its work. You can use this directory to keep all the metafiles needed by BitBake. Having a project directory is a good way to isolate your project.

Run Bitbake: At this point, you have nothing but a project directory. Run the bitbake command and see what it does:

```
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ bitbake
ERROR: Unable to find conf/bblayers.conf or conf/bitbake.conf. BBPATH is unset and/or not in a build
directory?
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$
```

The majority of this output is specific to environment variables that are not directly relevant to BitBake. However, the very first message regarding the BBPATH variable and the conf/ bblayers.conf file is relevant. When you run BitBake, it begins looking for metadata files. The BBPATH variable is what tells BitBake where to look for those files. BBPATH is not set and you need to set it. Without BBPATH, Bitbake cannot find any configuration files (.conf) or recipe files (.bb) at all. BitBake also cannot find the bitbake.conf file.

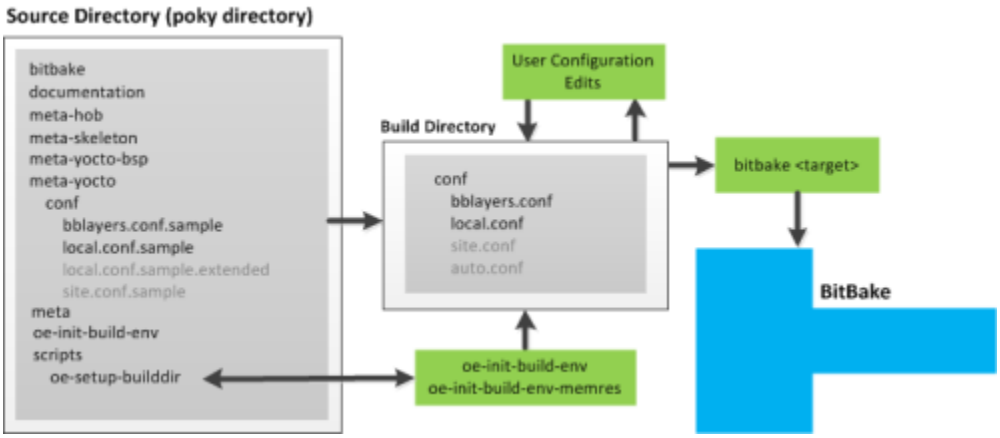
```
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ BBPATH="/home/tpthinh/Programming/Yocto/bitbake/hel
lo"
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ export BBPATH
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ echo BBPATH
BBPATH
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ echo $BBPATH
/home/tpthinh/Programming/Yocto/bitbake/hello
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$
```


Run BitBake again:

```
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ bitbake
ERROR: Unable to find conf/bblayers.conf or conf/bitbake.conf. BBPATH is unset and/or not in a build
directory?
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$
```

This shows that BitBake could not find the conf/bitbake.conf file in the project directory. This file is the first thing BitBake must find in order to build a target. And, since the project directory for this example is empty, you need to provide a conf/bitbake.conf file.

Create conf/bitbake.conf: The conf/bitbake.conf includes a number of configuration variables BitBake uses for metadata and recipe files. For this example, you need to create the file in your project directory and define some key BitBake variables. For more information on the bitbake.conf, see https://web.archive.org/web/20150325165911/ http://hambedded.org/ blog/2012/11/24/from-bitbake-hello-world-to-an-image/#an-overview-of-bitbakecon



First BitBake will search in the current directory for a file

Conf/bblayers.conf

In this file, BitBake search for a variable BBLAYERS which contains a list of folders. Each folder is one layer. In each folder, BitBake will search for conf/layers.conf first. In layers.conf, BitBake will read and assign to LAYERDIR which is used to identify directories of LAYER.

Let's make a conf/bblayers.conf

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /home/tpthinh/Programming/Yocto/bitbake/hello/meta \
"
```

```
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ bitbake
ERROR: Traceback (most recent call last):
  File "/home/tpthinh/Programming/Yocto/bitbake/lib/bb/cookerdata.py", line 162, in wrapped
    return func(fn, *args)
  File "/home/tpthinh/Programming/Yocto/bitbake/lib/bb/cookerdata.py", line 187, in parse_config_file
    return bb.parse.handle(fn, data, include)
  File "/home/tpthinh/Programming/Yocto/bitbake/lib/bb/parse/__init__.py", line 107, in handle
    return h['handle'](fn, data, include)
  File "/home/tpthinh/Programming/Yocto/bitbake/lib/bb/parse/parse_py/ConfHandler.py", line 121, in h
andle
    abs_fn = resolve_file(fn, data)
  File "/home/tpthinh/Programming/Yocto/bitbake/lib/bb/parse/__init__.py", line 131, in resolve_file
    raise IOError(errno.ENOENT, "file %s not found" % fn)
FileNotFoundError: [Errno 2] file /home/tpthinh/Programming/Yocto/bitbake/hello/meta/conf/layer.conf
not found

ERROR: Unable to parse /home/tpthinh/Programming/Yocto/bitbake/hello/meta/conf/layer.conf: [Errno 2]
file /home/tpthinh/Programming/Yocto/bitbake/hello/meta/conf/layer.conf not found
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ ls
bitbake-cookerdaemon.log  bitbake.lock  bitbake.sock  conf  meta
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ vim conf/bblayers.conf
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$ ls meta/
tpthinh@ubuntu:~/Programming/Yocto/bitbake/hello$
```


lib/bb/parse/parse_py/ConfHandler.py:36.

(...)

```
def init(data):
topdir = data.getVar('TOPDIR')
if not topdir:
data.setVar('TOPDIR', os.getcwd())
```

(...)

It means that if TOPDIR is not set, it will be set to the current path of executing BitBake.

Let's create file conf/layer.conf

Build folder

The build folder is the directory from which the `bitbake` command has to be executed. Here, BitBake expects its initial configuration file and it will place all files it creates in this folder.

To be able to run BitBake without getting any errors we need to create a build and a layer folder and place some required configuration files in there.

The minimal configuration will look like this:

```
bbTutorial/
├── build
│   ├── bitbake.lock
│   └── conf
│       └── bblayers.conf
└── meta-tutorial
    ├── classes
    │   └── base.bbclass
    └── conf
        ├── bitbake.conf
        └── layer.conf
```

build/conf/bblayers.conf

The first file BitBake expects is conf/bblayers.conf in its working directory, which is our build directory.
For now we create it with this content:

build/conf/bblayers.conf

```
BBPATH := "${TOPDIR}"
BBFILES ?= ""
BBLAYERS = "${TOPDIR}/../meta-tutorial"
```

meta-tutorial/conf/layer.conf

Each layer needs a conf/layer.conf file. For now we create it with this content:

meta-tutorial/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += ""
```

`meta-tutorial/classes/base.bbclass`

meta-tutorial/conf/bitbake.conf

For now, these files can be taken from the BitBake installation directory.
These files are located in the folders *bitbake-$version/conf* and *bitbake$version/classes*.
Simply copy them into the tutorial project.

`build/conf/bblayers.conf`

- Add the current directory to BBPATH.
  TOPDIR is internally set by BitBake to the current working directory.
- Initialize the variable BBFILES as empty. Recipes will be added later.

- Add the path of our meta-tutorial to the BBLAYERS variable.
  When executed, BitBake will search all given layer directories for additional configurations.

meta-tutorial/conf/layer.conf

- LAYERDIR is a variable BitBake passes to the layer it loads.
  We append this path to the BBPATH variable.
- BBFILES tells BitBake where recipes are.
  Currently we append nothing, but we will change this later.

tpthinh@ubuntu:~/Programming/Bitbake_Yocto_Experiment/Practice/bbTutorial/build$ bitbake -vDDD world

ERROR: no recipe files to build, check your BBPATH and BBFILES?

Summary: There was 1 ERROR message shown, returning a non-zero exit code.

tpthinh@ubuntu:~/Programming/Bitbake_Yocto_Experiment/Practice/bbTutorial/build$ bitbake

Nothing to do. Use 'bitbake world' to build everything, or run 'bitbake --help' for usage information.

tpthinh@ubuntu:~/Programming/Bitbake_Yocto_Experiment/Practice/bbTutorial/build$ ^C

tpthinh@ubuntu:~/Programming/Bitbake_Yocto_Experiment/Practice/bbTutorial/build$

BitBake needs recipes to do something. Currently there is none, so even if we run the `bitbake` command, without having a recipe it makes it not that much fun.

We can easily verify that there is nothing to do by running

[~/bbTutorial/build]

```
bitbake -s
```

Running this command will report:

```
NOTE: Not using a cache. Set CACHE = <directory> to enable.
Recipe Name    Latest Version    Preferred Version
===========    ==============    =================
```

This tells us 2 things:

1. BitBake tells us that it has no cache defined.

2. BitBake tells us that it has really nothing to do by showing us an empty list

BitBake caches meta data information in a directory, the cache. This help to speed up subsequent execution of commands.

We can fix the missing cache by simply adding a variable to bitbake.conf.
Therefore we edit the *meta-tutorial/conf/bitbake.conf* file and add at the end:

meta-tutorial/conf/bitbake.conf

```
...
CACHE = "${TMPDIR}/cache/default"
```

This is OK for now.

The next step is to add a recipe which requires 2 steps:

1. enable bitbake to find recipes
2. write a first recipe

Adding a recipe location to the tutorial layer

BitBake needs to know about which recipes a layer provides.
We edit our *meta-tutorial/conf/layer.conf* file and tell BitBake to load all recipe files by using a common pattern.

meta-tutorial/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb"
```

We make now use of the variable previously defined in *build/conf/bblayers.conf*. A recipe file has the extension *.bb, and if we respect the common pattern we can simply add all recipes to BitBake with one line.

Usually recipes have their own folder and are collected in groups, which means put recipes that are somehow related into the same directory.

It is common to name these folders recipes-'group', where group names a category of programs.

Now, since BitBake knows where to find recipes, we can actually add our first one.

Following the common pattern we create the folders *meta-tutorial/recipes-tutorial/first* and create the first recipe in there. Recipe-files also have a common name pattern which is `{recipe}_{version}.bb`.

Create the first recipe and task

Our first recipe will just print a log message. We put it into the first group, we will call it first, and it has the version 0.1.

So our **first** recipe in the first group is:

meta-tutorial/recipes-tutorial/first/first_0.1.bb

```
DESCRIPTION = "I am the first recipe"
PR = "r1"
do_build () {
  echo "first: some shell script running as build"
}
```

- The **task** do_build overrides the empty global build task from base.bbclass.
- **PR** is the internal revision number which should be updated after each change.
- Setting a **description** should be self explaining.

If everything is done correct we can ask bitbake to list the available recipes.

[~/bbTutorial/build]

```
bitbake  -s
Parsing recipes: 100% ...
Parsing of 1 .bb files complete...
Recipe Name    Latest Version   Preferred Version
===========    ==============   =================
first                 :0.1-r1
```

and we can run from the build directory

[~/bbTutorial/build]

```
bitbake first
```

Now check *tmp/work/first-0.1-r1/temp*, there are a lot of interesting files, for example:

build/tmp/work/first-0.1-r1/temp/log.do_build

```
DEBUG: Executing shell function do_build
first: some shell script running as build
DEBUG: Shell function do_build finished
```

Classes and functions

The next steps will be:

- Add a class
- Add a recipe that uses this class.

- Explore functions

Create the mybuild class

Let's create a different build function and share it.
We can do this by creating a class in the tutorial layer.
Therefore we create a new file called *meta-tutorial/classes/mybuild.bbclass*.

meta-tutorial/classes/mybuild.bbclass

```
addtask build
mybuild_do_build () {

  echo "running mybuild_do_build."

}

EXPORT_FUNCTIONS do_build
```

As in base.class, we add a build task. It is again a simple shell function.
*mybuild_do_* prefix is for following the conventions, *classname_do_functionname* for a task in a class.

For the `addtask` statement bitbake adds the *do_* prefix to the function name, if it was not given. This is, why in the example above, `addtask build` is used. You can also write `addtask do_build`, for bitbake it is the same.

EXPORT_FUNCTIONS makes the build function available to users of this class.
If we did not have this line it could not override the build function from base for.

For now this is enough to use this class with our second recipe.

Use myclass with the second recipe

Time to build a second recipe which shall use the build task defined in mybuild.
Say that this target needs to run a patch function before the build task.
And as an additional challenge second shall also demonstrate some python usage.

Following bitbakes naming conventions we add a new recipe folder and add the file *meta-tutorial/recipes-tutorial/second/second_1.0.bb* to it.

The new file wll look like this.

meta-tutorial/recipes-tutorial/second/second_1.0.bb

```
DESCRIPTION = "I am the second recipe"
PR = "r1"
inherit mybuild
```

```
def pyfunc(o):
    print(dir(o))

python do_mypatch () {
  bb.note ("runnin mypatch")
  pyfunc(d)
}

addtask mypatch before do_build
```

DESCRIPTION and PR are as usual.

The mybuild class becomes inherited and so myclass_do_build becomes the default build task.

The (pure python) function pyfunc takes some argument and runs the python dir function on this argumentm and prints the result.

The (bitbake python) mypatch function is added and registered as a task that needs to be executed before the build function.

mypatch calls pyfunc and passes the global bitbake variable d.
d (datastore) is defined by bitbake and is always available.

The mypatch function is registered as a task that needs to be executed before the build function.

Now we have an example that uses python functions.

bitbake -s

Loading cache: 100% |##| Time: 0:00:00

Loaded 2 entries from dependency cache.

Parsing recipes: 100% |##| Time: 0:00:00

Parsing of 2 .bb files complete (1 cached, 1 parsed). 2 targets, 0 skipped, 0 masked, 0 errors.

Recipe Name	Latest Version	Preferred Version	Required Version
first	:0.1-r1		
second	:1.0-r1		

Exploring recipes and tasks

Having now two recipes we are able to use and explore additional `bitbake` command options.

We can get information about recipes and their tasks and control what BitBake will execute.

List recipes and tasks

First we can check if BitBake really has both recipes. We can do this by using the *-s* option.

[~/bbTutorial/build]

```
bitbake -s
```

will now output

```
Recipe Name    Latest Version  Preferred Version
===========    ==============  =================
first                 :0.1-r1
second                :1.0-r1
```

If we would like to see all tasks a recipe provides we can explore them with `bitbake -c listtasks second`

This should give us a first overview on how to explore recipes and tasks.

Executing tasks or building the world

We have now several options on running builds or specified tasks for our recipes.

Build one recipe

> To run all tasks for our second recipe we simply call `bitbake second`

Execute one task

> We could also run a specific task for a recipe.
> Say we want only to run the mypatch task for the second recipe.
> This can be done by applying the command `bitbake -c mypatch second`

Build everything

> Simply running all tasks for all recipes can be done with `bitbake world`

You can play with the commands and see what happens.
The console output will tell you what was executed.

hecking the build logs

Bitbake creates a tmp/work directory in its actual build location where it stores all log files.
These log files contain interesting information and are worth to study.

An actual output after a first `bitbake world` run might look like this.

```
tmp/work/
  |- first-0.1-r1
    |- temp
      |-log.do_build -> log.do_build.20703
      |-log.do_build.20703
      |-log.task_order
      |-run.do_build -> run.do_build.20703
      |-run.do_build.20703
  |- second-1.0-r1
    |- second-1.0
    |- temp
      |-log.do_build -> log.do_build.20706
      |-log.do_build.20706
      |-log.do_mypatch -> log.do_mypatch.20705
      |-log.do_mypatch.20705
      |-log.task_order
      |-run.do_build -> run.do_build.20706
      |-run.do_build.20706
      |-run.do_mypatch -> run.do_mypatch.20705
      |-run.do_mypatch.20705
```

These log files contain useful information from BitBake about its actions as well as the output of the executed tasks.

BitBake layers

A typical BitBake project consists of more than one layer.

Usually layers contain recipes to a specific topic. Like basic system, graphical system, …and so on.

In some project there might also be more than one build target and each target is composed out of different layers.
A typical example would be to build a Linux distribution with and without GUI components.

Layers can be used, extended, configured and it is also possible to partial overwrite parts of existing layers.
This is useful since it allows reuse and customization for actual needs.

Working with multiple layers is the common case and therefore we also add an additional layer to the project.

Adding an additional layer

Adding a new layer can be done with the following steps:

1. Create the new layer folder
2. Create the layer configuration
3. Tell BitBake about the new layer
4. Add recipes to the layer

Adding the new layer folder

Create a new folder named *meta-two*.
We follow the common naming conventions and our working place looks now like this:

```
ls ~/bbTutorial
build  meta-tutorial  meta-two
```

Configure the new layer

Add *meta-two/conf/layer.conf* file. This file looks exactly like the one for the tutorial layer.

meta-two/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb"
```

Telling the BitBake about the new layer recipes

edit build/conf/bblayers.conf and extend the BBLAYERS variable.

build/conf/bblayers.conf

```
BBLAYERS = " \
  ${TOPDIR}/../meta-tutorial \
  ${TOPDIR}/../meta-two \
"
```

The bitbake-layers command

The tool to explore layer configurations is the `bitbake-layers` command.

The `bitbake-layers` command has a variety of useful options, the help text says it all.

```
Available commands:
  help
    display general help or help on a specified command
  show-recipes
    list available recipes, showing the layer they are provided by
  show-cross-depends
    figure out the dependency between recipes that crosses a layer boundary.
  show-appends
    list bbappend files and recipe files they apply to
```

```
  flatten
    flattens layer configuration into a separate output directory.
  show-layers
    show current configured layers
  show-overlayed
    list overlayed recipes (where the same recipe exists in another layer)
```

To explore, for example, which layers exists, you can run

[~/bbTutorial/build]

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

But before this will produce any useful output for our project we need to adopt our layers configurations.

Extending the layer configuration

We need to add some information to our layer configuration.

- a layer collection name
- a search pattern for files to add
- a layer priority

We start with *meta-tutorial/conf/layer.conf* and add

meta-tutorial/conf/layer.conf

```
# append layer name to list of configured layers
BBFILE_COLLECTIONS += "tutorial"
# and use name as suffix for other properties
BBFILE_PATTERN_tutorial = "^${LAYERDIR}/"
BBFILE_PRIORITY_tutorial = "5"
```

The used variables have an excellent description in the BitBake user manual, so there is no need to repeat this text here.

The patterns should be clear, we define the layer name and use this name to suffix some other variables.
This mechanism, using user defined domain suffixes in BitBake variable names, is used by BitBake on several locations.

Next we change meta-two/conf/layer.conf in the same way.

meta-two/conf/layer.conf

```
BBFILE_COLLECTIONS += "two"
BBFILE_PATTERN_two = "^${LAYERDIR}/"
BBFILE_PRIORITY_two = "5"
```

```
LAYERVERSION_two = "1"
```

If we now run `bitbake-layers show-layers` it will report

```
layer        path                                      priority
==========================================================
meta-tutorial   /path/to/work/build/../meta-tutorial  5
meta-two     /path/to/work/build/../meta-two    5
```

Layer compatibility

A project like yocto is composed out of very many layers. To ensure used layer 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 a 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 tutorial. Running, for example, `bitbake-layers show-recipes` will give 5 warnings.

```
 bitbake-layers show-recipes
NOTE: Starting bitbake server...
WARNING: Layer tutorial should set LAYERSERIES_COMPAT_tutorial 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.
WARNING: Layer tutorial should set LAYERSERIES_COMPAT_tutorial 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.
Parsing recipes: 100%
|##################################################################################
################################################| Time: 0:00:00
Parsing of 2 .bb files complete (0 cached, 2 parsed). 2 targets, 0 skipped, 0
masked, 0 errors.
WARNING: No bb files matched BBFILE_PATTERN_two
'^/home/bitbakeguide/ch07/build/../meta-two/'

Summary: There were 5 WARNING messages shown.
=== Available recipes: ===
first:
  meta-tutorial        0.1
second:
  meta-tutorial        1.0
```

The first 4 warning referee to the fact that the tutorial project has no layer series compatibility specified. The fifths warning is because layer two is empty, what will fix this in the next chapter. First, lets add a layer series name and specify that the 2 layers in the tutorial project are compatible.

Layer series core name

First we define a 'project core name'. This is done by setting the `LAYERSERIES_CORENAMES` variable. In yocto, this is done in the core layer, a layer with the name *core*.
We define the name in the tutorial layer because it is our first layer. The actual location does not matter, it could also be defined in the build/conf/bblayers.conf file.

We set the core name by adding

```
LAYERSERIES_CORENAMES = "bitbakeguilde"
```

to .meta-tutorial/conf/layer.conf

Layer series compatibility

We also need to specify that the tutorial 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. The variable ends on the layer name like we have seen it with the BBFILE_PATTERN or the BBFILE_PRIORITY variable.

Our layer configuration files looks now like that:

meta-tutorial/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb"
BBFILE_COLLECTIONS += "tutorial"
BBFILE_PATTERN_tutorial = "^${LAYERDIR}/"
BBFILE_PRIORITY_tutorial = "5"

LAYERSERIES_CORENAMES = "bitbakeguilde"

LAYERVERSION_tutorial = "1"
LAYERSERIES_COMPAT_tutorial = "bitbakeguilde"
```

meta-two/conf/layer.conf

```
BBPATH .= ":${LAYERDIR}"
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "two"
BBFILE_PATTERN_two = "^${LAYERDIR}/"
BBFILE_PRIORITY_two = "5"
LAYERVERSION_two = "1"
LAYERDEPENDS_two = "tutorial"

LAYERSERIES_COMPAT_two = "bitbakeguilde"
```

With these changes the four warnings about missing compatibility information are gone. All our layers are declared compatible to the core layer series.

Layer dependencies

You might have noticed that we also specified the `LAYERDEPENDS_two` variable in the layer.conf file of our second layer, meta-two.

By doing so we inform bitbake that this layer has a dependency to the tutorial layer. We will see in the next chapter, when we add more content to the meta-two layer, why this is the case.

Share and reuse 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
- bbappend files
- include files

To demonstrate these usages we are going to add an additional class to layer-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.
After that we will extend an existing recipe by using the append technique.

Class inheritance

To realize our configure/build chain we create a class that inherits the mybuild class and simply adds a configure task as a dependency of the build task.

We create this as another class, which we will then use to demonstrate a class and a recipe which make use of inheritance.

meta-two/classes/confbuild.bbclass

```
inherit mybuild

confbuild_do_configure () {

  echo "running configbuild_do_configure."

}
```

```
addtask do_configure before do_build

EXPORT_FUNCTIONS do_configure
```

use the mybuild class as a base

create the new function

define the order of the functions, configure before build

export the function so that it becomes available

We can now simply use this in our **third** recipe and use confbuild.

meta-two/recipes-base/third/third_0.1.2.bb

```
DESCRIPTION = "I am the third recipe"
PR = "r1"
inherit confbuild
```

This recipe inherits the confbuild class.

If we run now `bitbake third` it will execute the configure and build tasks for third.

bbappend files

An append file can be used to add functions to an existing recipe. The append happens via matching of the recipe file name. The content of the append file gets added to a recipe with the same name.

To be able to use append files the layer needs to be set up to load also them in addition to normal recipes.
Therefore we change our layer configuration and add loading of *.bbappend file to the BBFILES variable.

meta-two/conf/layer.conf

```
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"
```

Now we want to reuse and extend the existing first recipe.

This is why we added the `LAYERDEPENDS_two` in the previous chapter, we need this layer because it contains the recipe we want to extend.

What we want is running a patch function before running the build task, so we need to create the corresponding bbappend file and add our changes.
Therefore we need to create the *meta-two/recipes-base/first/* folder and the *first_0.1.bbappend* file.

meta-two/recipes-base/first/first_0.1.bbappend

```
python do_patch () {
  bb.note ("first:do_patch")
}

addtask patch before do_build
```

If we now list the tasks for the first recipe, we will see that it also has a patch task.

[~/bbTutorial/build]

```
bitbake -c listtasks first
Parsing recipes: 100% ...
....
do_showdata
do_build
do_listtasks
do_patch
...
```

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

Include files

BitBake has two directives to include files.

- `include` filename this is an optional include, if filename is not found no error is raised.
- `require` filename if filename is not found an error is raised.

It is worth mentioning that the include and require file name is relative to any directory in BBPATH.

Add a local.conf for inclusion

A common use case in BitBake projects is that the bitbake.conf includes a local.conf file which is usually placed in the build directory.
The local.conf file may contains special setups for the current build target.
This is the typical Yocto setup.

We mimic the typical usage of a local.conf and make bitbake.conf require a local.conf by adding the following lines to our `meta-tutorial/conf/bitbake.conf`.

meta-tutorial/conf/bitbake.conf

```
require local.conf
include conf/might_exist.conf
```

If we run now some build BitBake will show a long error message where the last line will be something like:

```
ERROR: Unable to parse conf/bitbake.conf: …Could not include required file local.conf
```

Adding a local.conf into the *build* directory, which can even be empty, will fix this.
The file given to the include statement is not required to exist.

Global variables

Global variables can be set by the user and existing recipes can use them.

Define global variables

An empty local.conf, as we have currently, is not very useful. So let's add some variable to local.conf. Assuming we are in the build directory we can run:

[~/bbTutorial/build]

```
echo 'MYVAR="hello from MYVAR"' > local.conf
```

or use our favorite editor to add this to the file.

Accessing global variables

We can access `MYVAR` in recipes or classes. For demonstration we create the new recipe group recipes-vars and a recipe myvar in it.

meta-two/recipes-vars/myvar/myvar_0.1.bb

```
DESCRIPTION = "Show access to global MYVAR"
PR = "r1"

do_build(){
  echo "myvar_sh: ${MYVAR}"
}

python do_myvar_py () {
  print ("myvar_py:" + d.getVar('MYVAR', True))
}

addtask myvar_py before do_build
```

Access the variable in a bash like syntax.

Access the variable via the global data store.

If we now run `bitbake myvar` and check the log output in the tmp directory, we will see that we indeed have access to the global MYVAR variable. If you are looking for the log file, search for a file like this: `build/tmp/work/myvar-0.1-r1/temp/log.do_myvar_py`.

Local variables

A typical recipe mostly consists only of variables that are used to set up functions defined in classes which the recipe inherits.

To have an idea how this work create

meta-two/classes/varbuild.bbclass

```
varbuild_do_build () {
  echo "build with args: ${BUILDARGS}"
}

addtask build

EXPORT_FUNCTIONS do_build
```

and use this class in a recipe

meta-two/recipes-vars/varbuild/varbuild_0.1.bb

```
DESCRIPTION = "Demonstrate variable usage \
  for setting up a class task"
PR = "r1"

BUILDARGS = "my build arguments"

inherit varbuild
```

Running `bitbake varbuild` will produce log files that shows that the build task respects the variable value which the recipe has set.

This is a very typical way of using BitBake. The general task is defined in a class, like for example download source, configure, make and others, and the recipe sets the needed variables for the task.

Summary

That's it with this tutorial, I am glad you made it until here and I hope you liked it.
After reading this tutorial you should have a solid foundation on the basics concepts BitBake is based on.

Topics covered are:

- BitBake as an engine that executes python and/or shell scripts.
- The common BitBake project layout and the default file locations.
- The basic understanding for layers and their relations to each other.
- The 5 file types BitBake uses (bb- bbclass- bbappend- conf- and include files).

- BitBake functions and tasks, show how to organize, group and call them.
- BitBake variables and the basic usage of them.

Being familiar with these topics will hopefully help if you start to use a project like Yocto and wonder what is going on.

Feel free to put a link to your BitBake project into the comment part below.

Yocto project

Yocto supports several Linux host distributions, and each Yocto release will document a list of the supported ones. Although the use of a supported Linux distribution is strongly advised, Yocto is able to run on any Linux system if it has the following dependencies:

- Git 1.8.3.1 or greater.
- Tar 1.27 or greater
- Python 3.4.0 or greater

```
tpthinh@ubuntu:~/Programming/Practice$ git --version
git version 2.25.1
tpthinh@ubuntu:~/Programming/Practice$ tar --version
tar (GNU tar) 1.30
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <https://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Written by John Gilmore and Jay Fenlason.
tpthinh@ubuntu:~/Programming/Practice$ python3 --version
Python 3.8.5
tpthinh@ubuntu:~/Programming/Practice$
```

To make sure you have the required package dependencies installed for Yocto and to follow the examples, run the following command from your shell:

$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib

Poky uses the OpenEmbedded build system and, as such, uses the BitBake tool, a task scheduler written in Python which is forked from Gentoo's Portage tool. You can think of BitBake as the make utility in Yocto. It will parse the configuration and recipe metadata, schedule a task list, and run through it.

BitBake is also the command-line interface to Yocto

Poky and BitBake are the two of the open source projects used by Yocto:

- The Poky project is maintained by the Yocto community. You can download Poky from its Git repository at http://git.yoctoproject.org/cgit/cgit.cgi/poky/
    - Development discussions can be followed and contributed to by visiting the development mailing list at https://lists.yoctoproject.org/listinfo/poky
    - Stable Yocto releases have their own branch. Yocto 2.4 is maintained in the rocko branch, and Yocto releases are tagged in that branch.
- BitBake, on the other hand, is maintained by both the Yocto and OpenEmbedded communities, as the tool is used by both. BitBake can be downloaded from tis Git repository at http://git.openembedded.org/bitbake/

The Poky distribution only supports virtualized QEMU machines for the following architectures:

- ARM (qemuarm, qemuarm64)
- X86 (qemux86)
- X86-64 (qemux86-64)
- PowerPC (qemuppc)
- MIPS (qemumips, qemumips64)

Codename	Yocto Project Version	Release Date	Current Version	Support Level	Poky Version	BitBake branch	Maintainer
Kirkstone	3.5	April 2022		Future	27.0		Richard Purdie <richard.purdie@linuxfoundation.org>
Honister	3.4	October 2021		Planning	26.0		Richard Purdie <richard.purdie@linuxfoundation.org>
Hardknott	3.3	April 2021	3.3.0 (April 2021)	Stable	25.0	1.50	Anuj Mittal <anuj.mittal@intel.com>
Gatesgarth	3.2	Oct 2020	3.2.3 (April 2021)	Stable	24.0	1.48	Anuj Mittal <anuj.mittal@intel.com>
Dunfell	3.1	April 2020	3.1.7 (April 2021)	Long Term Support	23.0	1.46	Steve Sakoman <steve@sakoman.com>
Zeus	3.0	October 2019	3.0.4 (August 2020)	EOL	22.0.3	1.44	Anuj/Armin
Warrior	2.7	April 2019	2.7.4 (June 2020)	EOL	21.0	1.42	Armin Kuster <akuster808@gmail.com>
Thud	2.6	Nov 2018	2.6.4 (October 2019)	EOL	20.0	1.40	Armin Kuster <akuster808@gmail.com>
Sumo	2.5	April 2018	2.5.3 (April 2019)	EOL	19.0	1.38	Armin Kuster <akuster808@gmail.com>
Rocko	2.4	Oct 2017	2.4.4 (November 2018)	EOL	18.0	1.36	Armin Kuster <akuster808@gmail.com>
Pyro	2.3	May 2017	2.3.4 (July 2018)	EOL	17.0	1.34	Armin Kuster <akuster808@gmail.com>
Morty	2.2	Nov 2016	2.2.4 (July 2018)	EOL	16.0	1.32	Armin Kuster <akuster808@gmail.com>
Krogoth	2.1	Apr 2016	2.1.3 (July 2017)	EOL	15.0	1.30	Armin Kuster <akuster808@gmail.com>
Jethro	2.0	Nov 2015	2.0.3 (January 2016)	EOL	14.0	1.28	Robert Yang <liezhi.yang@windriver.com>
Fido	1.8	Apr 2015	1.8.2	EOL	13.0	1.26	Joshua Lock <joshua.g.lock@intel.com>
Dizzy	1.7	Oct 2014	1.7.3	EOL	12.0	1.24	Armin Kuster <akuster808@gmail.com>
Daisy	1.6	Apr 2014	1.6.3	EOL	11.0	1.22	Saul Wold <sgw@linux.intel.com>
Dora	1.5	Oct 2013	1.5.4	EOL	10.0	1.20	Robert Yang <liezhi.yang@windriver.com>
Dylan	1.4	Apr 2013	1.4.3*	EOL	9.0	1.18	Paul Eggleton <paul.eggleton@linux.intel.com>
Danny	1.3	Oct 2012	1.3.2	EOL	8.0	1.16	Ross Burton <ross.burton@intel.com>
Denzil	1.2	Apr 2012	1.2.2	EOL	7.0	1.15	

The current Yocto release is 3.5, or Kirkstone, so we will install that into our host system. We will use the /opt/Yocto folder as the installation path

```
$ sudo install -o $(id -u) -g $(id -g) -d /opt/yocto
$ cd /opt/yocto
$ git clone --branch rocko git://git.yoctoproject.org/poky
```

Let use the last stable version instead, Hardknott

git clone –branch hardknott git://git.yoctoproject.org/poky

Poky contains three metadata directories, meta, meta-poky, and meta-yocto-bsp, as well as a template metadata layer, meta-skeleton, which can be used as a base for new layers. Poky's three meta data directories are explained here:

- meta: This directory contains the OpenEmbedded-core metadata, which supports the ARM, ARM64, x86, x86-64, PowerPC, MIPS, and MIPS64 architectures and the QEMU emulated hardware. You can download it from its Git repository at http://git.openembedded.org/openembedded-core/

Development discussions can be followed and contributed to by visiting the development mailing list at http://lists.openembedded.org/mailman/listinfo/openembedded-core

- meta-poky: This contains Poky's distribution-specific metadata.
- meta-yocto-bsp: This contains metadata for the reference hardware boards.

Create a build directory

Before building your first Yocto image, we need to create a build directory for it.

The build process, on a host system as outlined before, can take up to 1 hour and needs around 20 GB of hard drive space for a console-only image. A graphical image, such as core-image-sato, can take up to 4 hours for the build process and occupy around 50 GB of space.

There is no right way to structure the build directories when you have multiple projects, but a good practice is to have one build directory per architecture or machine type.

To create a build directory, we use the oe-init-build-env script provided by Poky. The script needs to be sourced into your current shell, and it will setup your environment to use the OpenEmbedded/Yocto build system, including adding the BitBake utility to your path.

You can specify a build directory to use or it will use build by default. We will use qemuarm for this example:

```
$ cd /opt/yocto/poky
$ source oe-init-build-env qemuarm
```

```
tpthinh@ubuntu:/opt/yocto/poky$ source oe-init-build-env qemuarm

### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'
tpthinh@ubuntu:/opt/yocto/poky/qemuarm$
```

The script will change to the specified directory.

As oe-init-build-env only configures the current shell, you will need to source it on every new shell. But if you point the script to an existing build directory, it will setup your environment but won't change any of your existing configurations.

BitBake is designed with a client/server abstraction, so we can also start a persistent server and connect a client to it. To instruct a BitBake server to stay resident, configure a timeout in seconds in your build directory's conf/local.conf configuration file as follows:

BB_SERVER_TIMEOUT = "n"

With n being the time in seconds for BitBake to stay resident.

With this setup, loading cache and configuration information each time is avoided, which saves some overhead.

The oe-init-build-env script calls scripts/oe-setup-builddir script inside the Poky directory to create the build directory.

On creation, the qemuarm build directory contains a conf directory with the following three files:

- bblayers.conf: This file lists the metadata layers to be considered for this project.
- local.conf: This file contains the project-specific configuration variables. You can set common configuration variables to different projects with a site.conf file, but this is not created by default. Similarly, there is also an auto.conf file which Is used by autobuilders. BitBake will first read site.conf, then auto.conf, and finally local.conf
- templateconf.cfg: This file contains the directory that includes the template configuration files used to create the project. By default, it uses the one pointed to by the templateconf file in your Poky installation directory, which is meta-poky/conf by default.

To start a build from scratch, that's all the build directory needs.

Erasing everything apart from these files will recreate your build from scratch, as shown here:

```
$ cd /opt/yocto/poky/qemuarm
$ rm -Rf tmp sstate-cache
```

You can specify different template configuration files to use when you create your build directory using the TEMPLATECONF variable, for example:

```
$ TEMPLATECONF=meta-custom/config source oe-init-build-env <build-dir>
```

The TEMPLATECONF variable needs to refer to a directory containing templates for both local.conf and bblayer.conf, but named local.conf.sample and bblayers.conf.sample. For now, we only use the unmodified default project configuration files.

Building your first image

Poky contains a set of default target images. You can list them by executing the following commands:

```
$ cd /opt/yocto/poky
$ ls meta*/recipes*/images/*.bb
```

```
tpthinh@ubuntu:/opt/yocto/poky/qemuarm$ cd ..
tpthinh@ubuntu:/opt/yocto/poky$ ls meta*/recipes*/images/*.bb
meta/recipes-core/images/build-appliance-image_15.0.0.bb
meta/recipes-core/images/core-image-base.bb
meta/recipes-core/images/core-image-minimal.bb
meta/recipes-core/images/core-image-minimal-dev.bb
meta/recipes-core/images/core-image-minimal-initramfs.bb
meta/recipes-core/images/core-image-minimal-mtdutils.bb
meta/recipes-core/images/core-image-tiny-initramfs.bb
meta/recipes-extended/images/core-image-full-cmdline.bb
meta/recipes-extended/images/core-image-kernel-dev.bb
meta/recipes-extended/images/core-image-testmaster.bb
meta/recipes-extended/images/core-image-testmaster-initramfs.bb
meta/recipes-graphics/images/core-image-clutter.bb
meta/recipes-graphics/images/core-image-weston.bb
meta/recipes-graphics/images/core-image-x11.bb
meta/recipes-rt/images/core-image-rt.bb
meta/recipes-rt/images/core-image-rt-sdk.bb
meta/recipes-sato/images/core-image-sato.bb
meta/recipes-sato/images/core-image-sato-dev.bb
meta/recipes-sato/images/core-image-sato-ptest-fast.bb
meta/recipes-sato/images/core-image-sato-sdk.bb
meta/recipes-sato/images/core-image-sato-sdk-ptest.bb
meta-selftest/recipes-test/images/error-image.bb
meta-selftest/recipes-test/images/oe-selftest-image.bb
meta-selftest/recipes-test/images/test-empty-image.bb
meta-selftest/recipes-test/images/wic-image-minimal.bb
meta-skeleton/recipes-multilib/images/core-image-multilib-example.bb
tpthinh@ubuntu:/opt/yocto/poky$
```

A full description of the different images can be found in the Yocto Project Reference Manual. Typically, these default images are used as a base and customized for your own project needs. The most frequently used base default images are:

- core-image-minimal: This is the smallest BusyBox, sysvinit, and udev-based console-only image.
- core-image-full-cmdline: This is the BusyBox-based console-only image with full hardware support and a more complete Linux system, including Bash.
- core-image-lsb: This is a console-only image that based on Linux Standard Base (LSB) compliance.
- core-image-x11: This is the basic X11 Windows-system-based image with a graphical terminal.
- core-image-sato: This is the X11 Window-system-based image with a SATO theme and a GNOME mobile desktop environment.
- core-image-weston: This is a Wayland protocol and Weston reference compositor-based image.

You will also find images with the following suffixes:

- dev: This image is suitable for development work, as it contains headers and libraries.
- sdk: This image includes a complete SDK that can be used for development on the target.

- Initramfs: This is an image that can be used for a RAM-based root filesystem, which can optionally be embedded with the Linux kernel.

  To build an image, we need to configure the machine we are building it for and pass its name to BitBake. For example, for the qemuarm machine, we would run the following:

```
$ cd /opt/yocto/poky/
$ source /opt/yocto/poky/oe-init-build-env qemuarm
$ MACHINE=qemuarm bitbake core-image-minimal
```

1. Or we could export the `MACHINE` variable to the current shell environment before sourcing the `oe-init-build-env` script with the following:

```
$ export MACHINE=qemuarm
```

1. On an already configured project, we could also edit the `conf/local.conf` configuration file to change the default machine to `qemuarm`:

```
- #MACHINE ?= "qemuarm"
+ MACHINE ?= "qemuarm"
```

1. Then, after setting up the environment, we execute the following:

```
$ bitbake core-image-minimal
```

With the preceding steps, BitBake will launch the build process for the specified target image.

How it works …

When you pass a target recipe to BitBake, it first parses the following configuration files in order:

- conf/bblayers.conf: This file is parsed to find all the configured layers.
- conf/layer.conf: This file is parsed on each configured layer.
- meta/conf/bitbake.conf: This file is parsed for its own configuration.
- conf/local.conf: This file is used for any other configuration the user may have for the current build.
- conf/machine/<machine>.conf: This file is the machine configuration; in our case, this is qemuarm.conf
- conf/distro/<distro>.conf: This file is the distribution policy; by default, this is the poky.conf file.

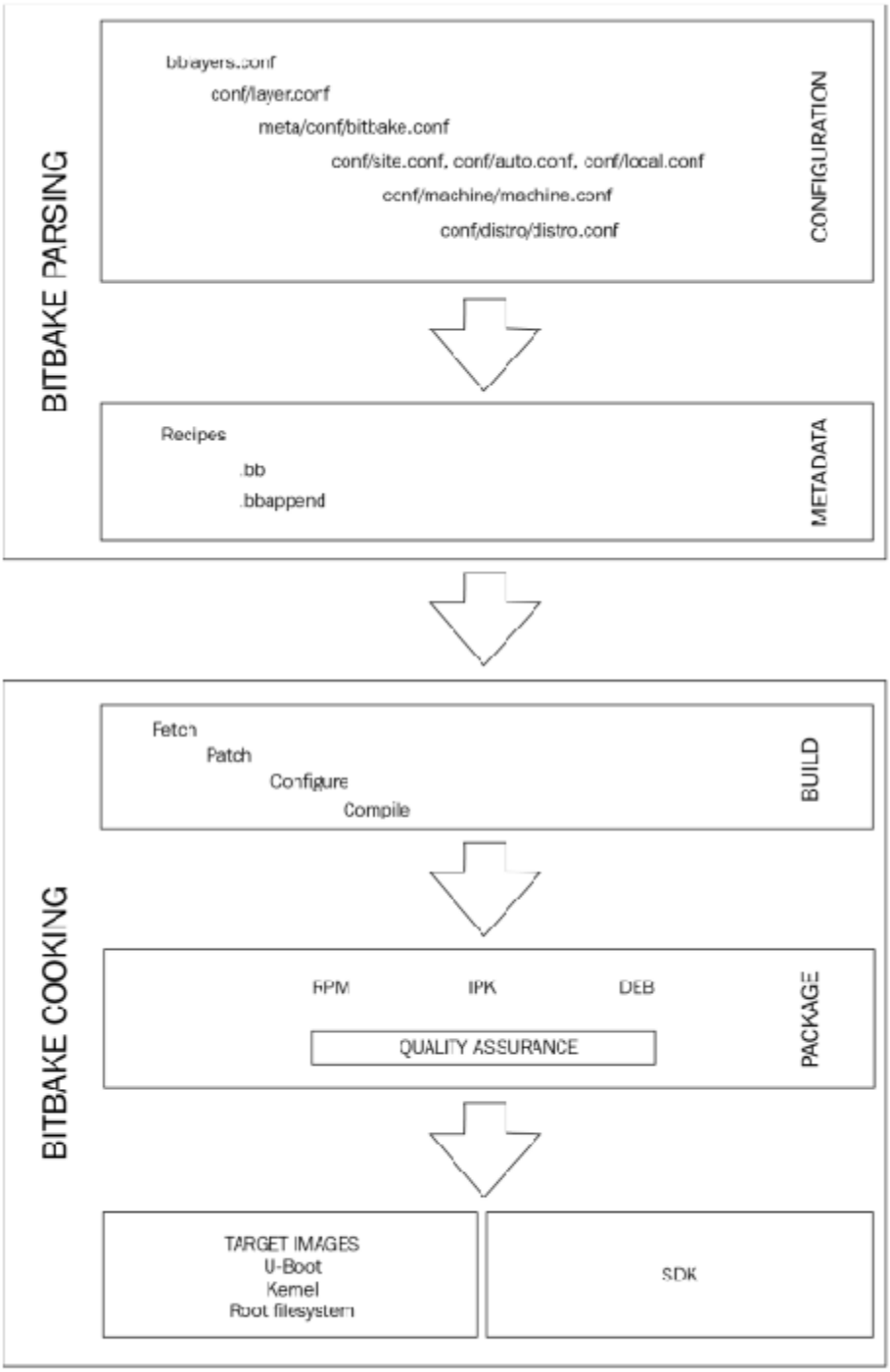
There are some other distribution variants included with Poky:

- poky-bleeding: Extension to the Poky default distribution that includes the most up-to-date versions of packages.
- Poky-lsb: LSB compile extension to Poky.

- Poky-tiny: Oriented to create headless systems with the smallest Linux kernel and BusyBox read-only for RAM-based root filesystems, using the musl C library.

And then, BitBake parses the target recipe that has been provided and its dependencies. The outcome is a set of interdependent tasks that BitBake will then execute in order.

A depiction of the BitBake build process is shown in the following diagram:



BitBake build process

Most developers won't be interested in keeping the whole build output for every package, so it is recommended to configure your project to remove it with the following configuration in your conf/local.conf file:

INHERIT += "rm_work"

But at the same time, configuring it for all packages means that you won't be able to develop or debug them.

You can add a list of packages to exclude from cleaning by adding them to the RM_WORK_EXCLUDE variable. For example, if you are going to do BSP work, a good setting might be:

RM_WORK_EXCLUDE +="linux-wandboard u-boot-fslc"

On a normal build, the –dbg packages that include debug symbols are not needed. To avoid creating –dbg packages, do this:

INHIBIT_PACKAGE_DEBUG_SPLIT = "1"

Once the build finishes, you can find the output images in the tmp/deploy/images/qemuarm directory inside your build directory.

You can test run your images on the QEMU emulator by executing this:

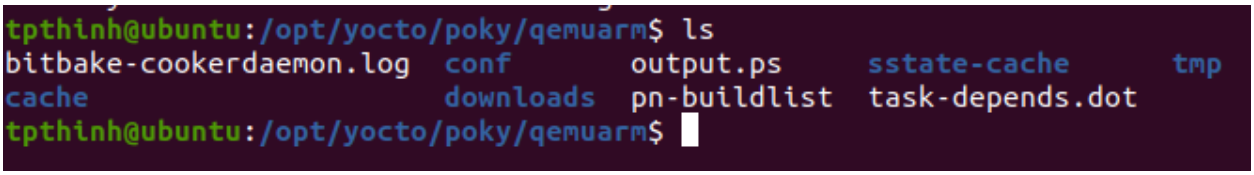
$ runqemu qemuarm core-image-minimal

The runqemu script included in Poky's scripts directory is a launch wrapper around the QEMU machine emulator to simplify its usage.

The Yocto Project also has a set of precompiled images for supported hardware platforms that can be downloaded from

http://downloads.yoctoproject.org/releases/yocto/yocto-2.4/

Check the output



```
tpthinh@ubuntu:/opt/yocto/poky/qemuarm/tmp/work/x86_64-linux$ ls
acl-native                      libassuan-native          ninja-native
alsa-lib-native                 libcap-native             openssl-native
attr-native                     libcap-ng-native          opkg-native
autoconf-archive-native         libcheck-native           opkg-utils-native
autoconf-native                 libcomps-native           patch-native
automake-native                 libdnf-native             pbzip2-native
bc-native                       libdrm-native             perl-native
binutils-cross-arm              libepoxy-native           pigz-native
binutils-native                 libffi-native             pixman-native
bison-native                    libgcrypt-native          pkgconfig-native
bzip2-native                    libgpg-error-native       popt-native
chrpath-native                  libmodulemd-native        prelink-native
cmake-native                    libmpc-native             pseudo-native
createrepo-c-native             libnsl2-native            python3-iniparse-native
cross-localedef-native          libpciaccess-native       python3-mako-native
curl-native                     libpcre2-native           python3-native
cwautomacros-native             libpcre-native            python3-setuptools-native
db-native                       libpthread-stubs-native   python3-six-native
debianutils-native              librepo-native            qemu-helper-native
dnf-native                      libsdl2-native            qemu-native
docbook-xml-dtd4-native         libsolv-native            qemu-system-native
docbook-xsl-stylesheets-native  libtirpc-native           quilt-native
dtc-native                      libtool-native            re2c-native
dwarfsrcfiles-native            libx11-native             readline-native
e2fsprogs-native                libxau-native             rpm-native
elfutils-native                 libxcb-native             rsync-native
expat-native                    libxdamage-native         shadow-native
file-native                     libxdmcp-native           shared-mime-info-native
flex-native                     libxext-native            sqlite3-native
gcc-cross-arm                   libxfixes-native          swig-native
gdbm-native                     libxml2-native            texinfo-dummy-native
gettext-minimal-native          libxrandr-native          tzcode-native
gettext-native                  libxrender-native         unifdef-native
glib-2.0-native                 libxslt-native            update-rc.d-native
gmp-native                      libxxf86vm-native         util-linux-libuuid-native
gnu-config-native               libyaml-native            util-linux-native
gobject-introspection-native    lzo-native                util-macros-native
gperf-native                    m4-native                 virglrenderer-native
gpgme-native                    makedepend-native         xcb-proto-native
gtk-doc-native                  makedevs-native           xmlto-native
itstool-native                  make-native               xorgproto-native
json-c-native                   mesa-native               xrandr-native
kern-tools-native               meson-native              xtrans-native
kmod-native                     mklibs-native             xz-native
ldconfig-native                 mpfr-native               zlib-native
libarchive-native               ncurses-native
tpthinh@ubuntu:/opt/yocto/poky/qemuarm/tmp/work/x86_64-linux$
```

```
tpthinh@ubuntu:/opt/yocto/poky/qemuarm/tmp/work/qemuarm-poky-linux-gnueabi$ ls
base-files          depmodwrapper-cross  linux-yocto             shadow-securetty
core-image-minimal  init-ifupdown        packagegroup-core-boot  sysvinit-inittab
tpthinh@ubuntu:/opt/yocto/poky/qemuarm/tmp/work/qemuarm-poky-linux-gnueabi$
```

Explaining the NXP Yocto ecosystem

As we saw, Poky metadata starts with the meta, meta-poky, and meta-yocto-bsp layers, and it can be expanded by using more layers.

An embedded product's development usually starts with hardware evaluation using a manufacturer's reference board design. Unless you are working with one of the reference boards already supported by Poky, you will need to extend Poky to support your hardware by adding extra BSP layers.

The FSL community BSP extends Poky with the following layers:

- Meta-freescale: This is the community layer that supports NXP reference designs. It has a dependency on OpenEmbedded-Core. Machines in this layer will be maintained even after NXP stops active development on them. You can download meta-freescale from its Git repository at http://git.yoctoproject.org/cgit/cgit.cgi/meta-freescale/

The meta-freescale layer includes NXP's proprietary binaries to enable some hardware features most notably its hardware graphics, multimedia, and encryption capabilities.

Reference

[1] https://a4z.gitlab.io/docs/BitBake/guide.html