# Recipes

## Recipes

- Recipes are fundamental components in the Yocto Project environment.
- A Yocto/OpenEmbedded recipe is a text file with file extension .bb
- Each software component built by the OpenEmbedded build system requires a recipe to define the component
- A recipe contains information about single piece of software.

## What information is present in a recipe?

- Information such as:
    - Location from which to download the unaltered source
    - any patches to be applied to that source (if needed)
    - special configuration options to apply
    - how to compile the source files and
    - how to package the compiled output
- Poky includes several classes that abstract the process for the most common development tools as projects based on Autotools, CMake, and QMake.

## Recipe File Format

- File Format: <base_name>_<version>.bb
- For example the file dropbear_2019.78.bb in poky/meta/recipes-core/dropbear has
    - base name : dropbear
    - version : 2019.78
- Another Example:
    - file tiff_4.0.10.bb in poky/meta/recipes-multimedia/libtiff/ has
    - base name : tiff
    - version : 4.0.10
    - The recipe is for a C library to read and write tiff image files
- **Note: Use lower-cased characters and do not include the reserved suffixes -native, -cross, -initial, or -dev**

## Bitbake

- Yocto/OpenEmbedded's build tool bitbake parses a recipe and generates list of tasks that it can execute to perform the build steps
- $ bitbake basename
    - The most important tasks are
    - do_fetch Fetches the source code
    - do_unpack Unpacks the source code into a working directory
    - do_patch Locates patch files and applies them to the source code
    - do_configure Configures the source by enabling and disabling any build-time and
        - configuration options for the software being built
    - do_compile Compiles the source in the compilation directory

    - do_install Copies files from the compilation directory to a holding area
    - do_package Analyzes the content of the holding area and splits it into subsets
        - based on available packages and files
    - do_package_write_rpm Creates the actual RPM packages and places them in the Package Feed area
- Generally, the only tasks that the user needs to specify in a recipe are
    - do_configure,
    - do_compile and
    - do_install ones.
- The remaining tasks are automatically defined by the YP build system
- The above task list is in the correct dependency order. They are executed from top to bottom.
- You can use the -c argument to execute the specific task of a recipe.
    - $ bitbake -c compile dropbear
- To list all tasks of a particular recipe
    - $ bitbake <recipe name> -c listtasks

## Stage 1: Fetching Code (do_fetch)

- The first thing your recipe must do is specify how to fetch the source files.
- Fetching is controlled mainly through the SRC_URI variable
- Your recipe must have a SRC_URI variable that points to where the source is located.
- The SRC_URI variable in your recipe must define each unique location for your source files.
- Bitbake supports fetching source code from git, svn, https, ftp, etc
- URI scheme syntax: scheme://url;param1;param2
- scheme can describe a local file using file:// or remote locations with https://, git://, svn://, hg://, ftp://
- By default, sources are fetched in $BUILDDIR/downloads

### Examples of SRC_URI

- busybox_1.31.0.bb : SRC_URI = "https://busybox.net/downloads/busybox-${PV}.tar.bz2"
- linux-yocto_5.2.bb : SRC_URI = "git://git.yoctoproject.org/linux-yocto.git"
- weston-init.bb : SRC_URI = "file://init"
- The do_fetch task uses the prefix of each entry in the SRC_URI variable value to determine how to fetch the source code.
- **Note: Any patch files present, needs to be specified in SRC_URI**

## Stage 2: Unpacking (do_unpack)

- All local files found in SRC_URI are copied into the recipe's working directory, in $BUILDDIR/tmp/work/
- When extracting a tarball,BitBake expects to find the extracted files in a directory named <application>-<version>. This is controlled by the S variable.
- If the tarball follows the above format, then you need not define S variable
    - Eg. SRC_URI = "https://busybox.net/downloads/busybox-${PV}.tar.bz2;name=tarball
- If the directory has another name, you must explicitly define S
- If you are fetching from SCM like git or SVN, or your file is local to your machine, you need to define S
- If the scheme is git, S = ${WORKDIR}/git

## Stage 3: Patching Code (do_patch)

- Sometimes it is necessary to patch code after it has been fetched.
- Any files mentioned in SRC_URI whose names end in .patch or .diff or compressed versions of these suffixes (e.g. diff.gz) are treated as patches
- The do_patch task automatically applies these patches.
- The build system should be able to apply patches with the "-p1" option (i.e. one directory level in the path will be stripped off).
- If your patch needs to have more directory levels stripped off, specify the number of levels using the "striplevel" option in the SRC_URI entry for the patch

### Licensing

- Your recipe needs to have both the LICENSE and LIC_FILES_CHKSUM variables:

### LICENSE:

- This variable specifies the license for the software.
- If you do not know the license under which the software you are building is distributed, you should go to the source code and look for that information.
- Typical files containing this information include COPYING, LICENSE, and README files.
- You could also find the information near the top of a source file.
- For example, given a piece of software licensed under the GNU General Public License version 2, you would set LICENSE as follows:
- LICENSE = "GPLv2"
- For standard licenses, use the names of the files in meta/files/common-licenses/

### LIC_FILES_CHKSUM:

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

- If it has, the build produces an error and it affords you the chance to figure it out and correct the problem.
- Example that assumes the software has a COPYING file:
    - LIC_FILES_CHKSUM = file://COPYING;md5=xxx

## Stage 4: Configuration (do_configure)

- Most software provides some means of setting build-time configuration options before compilation
- Typically, setting these options is accomplished by running a configure script with options, or by modifying a build configuration file
- Autotools: If your source files have a configure.ac file, then your software is built using Autotools.
- CMake: If your source files have a CMakeLists.txt file, then your software is built using CMake
- If your source files do not have a configure.ac or CMakeLists.txt file,you normally need to provide a do_configure task in your recipe unless there is nothing to configure.

## Stage 5: Compilation (do_compile)

- do_compile task happens after source is fetched, unpacked, and configured.

## Stage 6: Installation (do_install)

- After compilation completes, BitBake executes the do_install task
- During do_install, the task copies the built files along with their hierarchy to locations that would mirror their locations on the target device.

## Stage 7: Packaging (do_package)

- The do_package task splits the files produced by the recipe into logical components.
- Even software that produces a single binary might still have debug symbols, documentation, and other logical components that should be split out.
- The do_package task ensures that files are split up and packaged correctly.

## Challenge

- Write a recipe for a C code which also uses a header file (Two files: .c/.h)

## Simple Helloworld recipe

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

```
#include <stdio.h>

int main()
{
        printf("Hello World\n");
        return 0;
}
```

Step 2: Create a folder in the layer recipes-example 'myhello'

```
mkdir -p recipes-examples/myhello
```

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

```
mkdir -p recipes-examples/myhello/files
```

Copy the userprog.c into the above location

Step 4: Create a file called 'myhello_0.1.bb' with the following content:

```
DESCRIPTION = "Simple helloworld application"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://userprog.c"

S = "${WORKDIR}"
```

```
do_compile() {
        ${CC} userprog.c ${LDFLAGS} -o userprog
}

do_install() {
        install -d ${D}${bindir}
        install -m 0755 userprog ${D}${bindir}
}
```

Step 5: bitbake myhello

## install keyword

- install not only copies files but also changes its ownership and permissions and optionally removes debugging symbols from executables.
- It combines cp with chown, chmod and strip

## WORKDIR

- The location of the work directory in which the OpenEmbedded build system builds a recipe.
- This directory is located within the TMPDIR directory structure and is specific to the recipe being built and the system for which it is being built.
- The WORKDIR directory is defined as follows:
    - ${TMPDIR}/work/${MULTIMACH_TARGET_SYS}/${PN}/${EXTENDPE}${PV}-${PR}
- TMPDIR: The top-level build output directory
    - MULTIMACH_TARGET_SYS: The target system identifier
    - PN: The recipe name
    - EXTENDPE: Mostly blank
    - PV: The recipe version
    - PR: The recipe revision

## Recipe Explanation

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

$ bitbake -c cleanall myhello

### 1. do_fetch:

- in this case, since the specified SRC_URI variable points to a local file, BitBake will simply copy the file in the recipe WORKDIR.
- This is why the S environment variable (which represents the source code location) is set to WORKDIR.
    - $ bitbake -c fetch myhello
    - $ bitbake -c unpack myhello
    - $ bitbake -c configure myhello

## What is sysroot?

- contain needed headers and libraries for generating binaries that run on the target architecture
- recipe-sysroot-native:
    - includes the build dependencies used in the host system during the build process.
    - It is critical to the cross-compilation process because it encompasses the compiler, linker, build script tools, and more,
- recipe-sysroot:
    - the libraries and headers used in the target code

## 2. do_compile:

- when executing this task, BB will invoke the C cross-compiler for compiling the myhello.c source file.
- The results of the compilation will be in the folder pointed by the B environment variable (that, in most of the cases, is the same as the S folder).

## 3. do_install:

- this task specifies where the helloworld binary should be installed into the rootfs.
- It must be noticed that this installation will only happen within a temporary rootfs folder within the recipe WORKDIR (pointed by the variable D)
- image: This contains the files installed by the recipe (pointed to D variable).

## 4. do_package:

- in this phase the file installed in the directory D will be packaged in a package named myhello.
- This package will be used later from BitBake when eventually building a rootfs image containing the helloworld recipe package
- packages: The extracted contents of packages are stored here
- packages-split: The contents of packages, extracted and split, are stored here. This has a sub-directory for each package

## OpenEmbedded Variables

### S :

- Contains the unpacked source files for a given recipe

### D :

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

### WORKDIR:

- The location where the OpenEmbedded build system builds a recipe (i.e. does the work to create the package).

### PN :

- The name of the recipe used to build the package

### PV :

- The version of the recipe used to build the package

### PR :

- The revision of the recipe used to build the package.

## Add the recipe to rootfs

- IMAGE_INSTALL += "myhello"

## Who defines the fetch, configure and other tasks

- When bitbake is run to build a recipe, base.bbclass file gets inherited automatically by any recipe
- You can find it in classes/base.bbclass
- This class contains definitions for standard basic tasks such as fetching, unpacking, configuring (empty by default), compiling (runs any Makefile present), installing (empty by default) and packaging (empty by default)
- These classes are often overridden or extended by other classes such as the autotools class or the package class.

## Challenge

- Write a recipe for a C code which also uses a header file (Two files: .c/.h)