Basic Variable Setting (=)

VARIABLE = "value"

The following example sets VARIABLE to "value".

Eg. MACHINE = "raspberrypi3"

This assignment occurs immediately as the statement is parsed.

It is a **"hard" assignment**.

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

```
VARIABLE = " value"
VARIABLE = "value "
```

Note: You can also use single quotes (') instead of double quotes when setting a variable's value.

Benefit:

```
VARIABLE = 'I have a " in my value'
```

How to check the value of variable?

For configuration changes, use the following:

```
$ bitbake -e
```

This command displays variable values after the configuration files (i.e. local.conf, bblayers.conf, bitbake.conf and so forth) have been parsed.

For recipe changes, use the following:

```
$ bitbake recipe -e | grep VARIABLE="
```

Line Joining

BitBake joins any line ending in a backslash character ("") with the following line before parsing statements.

The most common use for the "" character is to split variable assignments over multiple lines.

```
BBLAYERS ?= " \
  /home/linuxtrainer/Yocto_Training/source/poky/meta \
  /home/linuxtrainer/Yocto_Training/source/poky/meta-poky \
  /home/linuxtrainer/Yocto_Training/source/poky/meta-yocto-bsp
\
  "
```

To check:

```
$ bitbake -e | grep ^BBLAYERS
```

Setting a default value (?=)

?= is used for **soft assignment** of a variable.

What's the benefit?

Allows you to define a variable if it is undefined when the statement is parsed.

If the variable has a value, then the soft assignment is lost.

Eg:

```
MACHINE ?= "qemuarm"
```

If MACHINE is already set before this statement is parsed, the above value is not assigned.

If MACHINE is not set, then the above value is assigned.

Note: Assignment is immediate.

What happens if we have multiple ?=

If multiple "?=" assignments to a single variable exist, the first of those ends up getting used.

Setting a weaker default value (??=)

Weaker default value is achieved using the ??= operator.

Difference between ?= and ??=

Assignment is made at the **end** of the parsing process rather than immediately.

When multiple "??=" assignments exist, the last one is used.

Eg.

```
MACHINE ??= "qemux86"
MACHINE ??= "qemuarm"
```

If MACHINE is not set, the value of MACHINE = "qemuarm".

If MACHINE is set before the statements, then the value will not be changed.

It is called **weak assignment**, as assignment does not occur until the end of the parsing process.

Note: "=" or "?=" assignment will override the value set with "??=".

Variable Expansion

Variables can reference the contents of other variables using a syntax similar to variable expansion in Bourne shells.

```
A = "hello"
B = "${A} world"
```

Check values:

```
$ bitbake -e | grep ^A=
$ bitbake -e | grep ^B=
```

The "=" operator does not immediately expand variable references in the right-hand side.

Instead, expansion is deferred until the variable assigned to is actually used.

```
A = "${B} hello"
B = "${C} world"
C = "linux"
```

Check:

```
$ bitbake -e | grep ^A=
```

What happens if C is not defined above?

The string is kept as is.

Immediate Variable Expansion (:=)

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

```
A = "11"
B = "B:${A}"
A = "22"
C := "C:${A}"
D = "${B}"
```

```
A = "11"
B := "B:${A}"
A = "22"
C := "C:${A}"
D = "${B}"
```

Appending Operators

```
+=

A = "hello"
A += "world"

.=

A = "hello"
A .= "world"
```

Difference between += and .= is space is automatically added in +=.

These operators take immediate effect during parsing.

Prepending Operators

```
=+

A = "world"
A =+ "hello"

=.

A = "world"
A =. "hello"
```

Same as previous, =+ adds an additional space.

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.

```
A = "hello"
A_append = " world"

B = "test"
B_append = "world"

C = "full"
C_prepend = "house"
```

Removal

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.

```
FOO = "123 456 789 123456 123 456 123 456"
FOO_remove = "123"
```

Check:

```
$ bitbake -e | grep ^FOO=
```

Override Style Operation Advantages

An advantage of the override style operations "_append", "_prepend", and "_remove" as compared to the "+=" and "=+" operators is that the override style operators provide guaranteed operations.

```
IMAGE_INSTALL += "usbutils"

IMAGE_INSTALL_append = " usbutils"
```

What is a layer?

A layer is a logical collection of related recipes.

Types of Layers:

- oe-core
- BSP Layer
- Application layer

Layer name starts with `meta-`, but this is not a technical restriction.

Eg. meta-mycustom

Why create a meta layer?

Despite most of the customization being possible with the `local.conf` configuration file, it is not possible to:

- Store recipes for your own software projects
- Create your own images
- Consolidate patches/modifications to other people's recipes
- Add a new custom kernel
- Add a new machine

Most important point:

Do not edit POKY/UPSTREAM Layers, as it complicates future updates.

Advantage: This allows you to easily port from one version of Poky to another.

Depending on the type of layer, add the content:

- If the layer is adding support for a machine, add the machine configuration in `conf/machine/`.
- If the layer is adding distro policy, add the distro configuration in `conf/distro/`.

- If the layer introduces new recipes, put the recipes you need in `recipes-*` subdirectories of the layer directory.

Recipe directories inside layers

By convention, recipes are split into categories.

The most difficult part is deciding in which category your recipe will go.

`..meta/recipies.txt`

By checking what was already done in the official layers, you should get a good idea of what you should do.

Layer Priority

Each layer has a priority, which is used by BitBake to decide which layer takes precedence if there are recipe files with the same name in multiple layers.

A higher numeric value represents a higher priority.

Creating a Layer

There are two ways to create your own layer:

1. **Manually**
2. **Using script**

Manually:

Step 1: Create a directory for the layer. For example:

`meta-mylayer`

Step 2: Create a `conf/layer.conf`.

You can simply copy `meta-oe`'s one and just change `openembedded-layer` to something appropriate for your layer; you may also want to set the priority as appropriate.

Step 3: Update `bblayers.conf` file with the new layer.

Creating a layer using tool

You can create your own layer using the `bitbake-layers create-layer` command.

```
$ bitbake-layers create-layer --help
```

The tool automates layer creation by setting up a subdirectory with a `layer.conf` configuration file, a `recipes-example` subdirectory that contains an `example.bb` recipe, a licensing file, and a README.

```
$ bitbake-layers create-layer ../source/meta-mylayer
```

Default priority of the layer is 6.

```
$ bitbake-layers add-layer ../source/meta-mylayer
$ bitbake-layers show-layers
```

Layer Configuration File layer.conf

```
# The configuration and classes directory is appended to BBPATH
BBPATH .= ":${LAYERDIR}"
```

```
# The recipes for the layers are appended to
BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb
${LAYERDIR}/recipes-*/*/*.bbappend"

# The BBFILE_COLLECTIONS variable is then appended
with the layer name
BBFILE_COLLECTIONS += "skeleton"
BBFILE_PATTERN_skeleton = "^${LAYERDIR}/"

# The BBFILE_PRIORITY variable then assigns a
priority to the layer.
BBFILE_PRIORITY_skeleton = "1"

# This should only be incremented on significant
changes that will
# cause compatibility issues with other layers
LAYERVERSION_skeleton = "1"

LAYERDEPENDS_skeleton = "core"

LAYERSERIES_COMPAT_skeleton = "zeus"
```

---

yocto-check-layer

The `yocto-check-layer` script provides you a way to assess how compatible your layer is with the Yocto Project.

You should use this script if you are planning to apply for the Yocto Project Compatible Program.

```
$ source oe-init-build-env
$ yocto-check-layer your_layer_directory
```

What is an image

An image is a top-level recipe. (It inherits an `image.bbclass`)

Building an image creates an entire Linux distribution from source, including:

- Compiler, tools, libraries
- BSP: Bootloader, Kernel
- Root filesystem:
  - Base OS
  - Services
  - Applications
  - etc

Creating custom images

You often need to create your own image recipe to add new packages or functionality.

Two ways:

1. **Creating an image from scratch**
2. **Extending an existing recipe (preferable)**

Package group

A package group is a set of packages that can be included in any image.

Using the `packagegroup` name in the `IMAGE_INSTALL` variable installs all the packages defined by the package group into the root file system of your target image.

There are many package groups available, present in subdirectories named `packagegroups`.

```
$ find . -name 'packagegroups'
```

Package group files are recipes (`.bb`) and start with `packagegroup-`. For example:

```
packagegroup-core-boot: Provides the minimum set
of packages necessary to create a bootable image
with console.
```

Creating an image from scratch

The simplest way is to inherit the `core-image` bbclass, which provides a set of image features that can be used easily.

```
inherit core-image
```

This means the definition of what actually gets installed is defined in the `core-image.bbclass`.

Image recipes set `IMAGE_INSTALL` to specify the packages to install into an image through `image.bbclass`.

Steps:

```
$ mkdir -p recipes-examples/images
$ vi recipes-examples/images/lwl-image.bb
SUMMARY = "A small boot image for LWL learners"
LICENSE = "MIT"
inherit core-image
# Core files for basic console boot
IMAGE_INSTALL = "packagegroup-core-boot"
IMAGE_ROOTFS_SIZE ?= "8192"
# Add our needed applications
IMAGE_INSTALL += "usbutils"
```

Reusing an existing image

When an image mostly fits our needs and we need to do minor adjustments, it is convenient to reuse its code.

This makes code maintenance easier and highlights functional differences.

Example: To include an application (lsusb)

```
$ vim recipes-examples/images/lwl-image-reuse.bb

require recipes-core/images/core-image-minimal.bb
IMAGE_INSTALL_append = " usbutils"
```

Customizing Images Using Custom IMAGE_FEATURES and EXTRA_IMAGE_FEATURES

Another method for customizing your image is to enable or disable high-level image features by using the `IMAGE_FEATURES` and `EXTRA_IMAGE_FEATURES` variables.

`IMAGE_FEATURES`/`EXTRA_IMAGE_FEATURES` enables special features for your image, such as:

- Empty password for root
- Debug image
- Special packages
- X11
- Splash
- SSH-server

Difference between IMAGE_FEATURES and EXTRA_IMAGE_FEATURES

Best practice:

- Use `IMAGE_FEATURES` from a recipe
- Use `EXTRA_IMAGE_FEATURES` from `local.conf`

How it works?

To understand how these features work, refer to `meta/classes/core-image.bbclass`.

This class lists the available `IMAGE_FEATURES`, most of which map to package groups, while some (e.g., `debug-tweaks`, `read-only-rootfs`) resolve as general configuration settings.

`The build system automatically adds the appropriate packages or configurations to the IMAGE_INSTALL variable based on the contents of IMAGE_FEATURES.`

Example of IMAGE_FEATURES

To illustrate how you can use these variables to modify your image, consider an example that selects the SSH server.

The Yocto Project ships with two SSH servers you can use with your images:

1. **Dropbear:** Minimal SSH server suitable for resource-constrained environments.
2. **OpenSSH:** A well-known standard SSH server implementation.

By default:

- The `core-image-sato` image is configured to use Dropbear.
- The `core-image-full-cmdline` and `core-image-lsb` images include OpenSSH.
- The `core-image-minimal` image does not contain an SSH server.

---

debug-tweaks

In the default state, the `local.conf` file has `EXTRA_IMAGE_FEATURES` set to `debug-tweaks`.

Advantage:

Enables password-less login for the root user, making debugging or inspection easier during development.

Disadvantage:

Anyone can easily log in during production.

Solution:

Remove the `debug-tweaks` feature from the production image.

Read-Only Root Filesystem

Why do we need read-only rootfs?

- Reduce wear on flash memory
- Eliminate system file corruption

How to do it?

To create the read-only root filesystem, simply add the `read-only-rootfs` feature to your image.

```
IMAGE_FEATURES = "read-only-rootfs" # in your recipe
```

or

```
EXTRA_IMAGE_FEATURES += "read-only-rootfs" # in local.conf
```

Boot Splash screen

```
IMAGE_FEATURES += "splash"
```

or

```
EXTRA_IMAGE_FEATURES += "splash"
```

Some other Features

- `tools-debug`: Installs debugging tools such as strace and gdb.
- `tools-sdk`: Installs a full SDK that runs on the device.

IMAGE_LINGUAS

Specifies the list of locales to install into the image during the root filesystem construction process.

```
IMAGE_LINGUAS = "zh-cn"
```

Inside QEMU image:

```
$ locale -a
```

IMAGE_FSTYPES

The `IMAGE_FSTYPES` variable determines the root filesystem image type.

If more than one format is specified, one image per format will be generated.

Image format instructions are delivered in Poky:

```
meta/classes/image_types.bbclass
```

To check the current value of `IMAGE_FSTYPES` for an image:

```
$ bitbake -e <image_name> | grep ^IMAGE_FSTYPES=
```

Types supported

Supported filesystem types include:

- `btrfs`
- `container`
- `cpio`
- `cpio.gz`
- `cpio.lz4`
- `cpio.lzma`
- `cpio.xz`
- `cramfs`
- `elf`
- `ext2`, `ext2.bz2`, `ext2.gz`, `ext2.lzma`
- `ext3`, `ext3.gz`
- `ext4`, `ext4.gz`
- `f2fs`
- `hddimg`
- `iso`
- `jffs2`, `jffs2.sum`
- `multiubi`
- `squashfs`, `squashfs-lz4`, `squashfs-lzo`, `squashfs-xz`
- `tar`, `tar.bz2`, `tar.gz`, `tar.lz4`, `tar.xz`
- `ubi`, `ubifs`
- `wic`, `wic.bz2`, `wic.gz`, `wic.lzma`

Creating your own image type

If you have a particular layout on your storage (e.g., bootloader location on an SD card), you may want to create your own image type.

This can be achieved through a class that inherits from `image_types`.

The class must define a function named `IMAGE_CMD_<type>`.

Example:

`sdcard_image-rpi.bbclass in meta-raspberrypi`

IMAGE_NAME

The name of the output image files minus the extension.

This variable is derived using the `IMAGE_BASENAME`, `MACHINE`, and `DATETIME` variables.

```
IMAGE_NAME = "${IMAGE_BASENAME}-${MACHINE}-
${DATETIME}"
```

IMAGE_MANIFEST

The manifest file for the image.

This file lists all the installed packages that make up the image.

Each line contains package information:

`packagename packagearch version`

The image class defines the manifest file as follows:

```
IMAGE_MANIFEST =
"${DEPLOY_DIR_IMAGE}/${IMAGE_NAME}.rootfs.manifest
"
```

Recipes

Recipes are fundamental components in the Yocto Project environment.

A Yocto/OpenEmbedded recipe is a text file with the file extension `.bb`.

Each software component built by the OpenEmbedded build system requires a recipe to define the component.

A recipe contains information about a single piece of software.

What information is present in a recipe?

Recipes contain 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
- How to package the compiled output

Poky includes several classes that abstract the process for the most common development tools such as Autotools, CMake, and QMake.

Recipe File Format

The file format for a recipe is:

`<base_name>_<version>.bb`

Example:

File: `dropbear_2019.78.bb` in `poky/meta/recipes-core/dropbear`

- Base name: `dropbear`
- Version: `2019.78`

Another example:

File: `tiff_4.0.10.bb` in `poky/meta/recipes-multimedia/libtiff/`

- Base name: `tiff`
- Version: `4.0.10`

The recipe is for a C library to read and write TIFF image files.

Note: Use lowercase 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 a list of tasks to execute the build steps.

To build a recipe:

```
$ bitbake <basename>
```

Important tasks include:

- `do_fetch`: Fetches the source code
- `do_unpack`: Unpacks the source code into a working directory
- `do_patch`: Locates and applies patch files to the source code
- `do_configure`: Configures the source by enabling/disabling build-time options
- `do_compile`: Compiles the source
- `do_install`: Copies files to a holding area
- `do_package`: Splits files into subsets for packaging
- `do_package_write_rpm`: Creates RPM packages and places them in the package feed area

Generally, the only tasks the user needs to specify in a recipe are:

- `do_configure`
- `do_compile`
- `do_install`

The remaining tasks are automatically defined by the Yocto Project build system.

Task execution order:

The tasks are executed in the correct dependency order from top to bottom.

To run a specific task in a recipe:

```
$ bitbake -c compile dropbear
```

To list all tasks for a particular recipe:

```
$ bitbake <recipe_name> -c listtasks
```

Stage 1: Fetching Code (do_fetch)

The first step in a recipe is to specify how to fetch the source files. This is controlled by the `SRC_URI` variable, which must point to the location of the source files.

BitBake supports fetching source code from various protocols including:

- `git://`
- `svn://`
- `https://`
- `ftp://`
- `file://` (for local files)

Syntax:

```
SRC_URI = "scheme://url;param1;param2"
```

By default, sources are fetched in `$BUILDDIR/downloads`.

Examples:

```
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"
```

Patches should be included in `SRC_URI` to be automatically applied.

Stage 2: Unpacking (do_unpack)

All files found in `SRC_URI` are copied into the recipe's working directory: `$BUILDDIR/tmp/work/`.

If the extracted files follow the format `<application>-<version>`, the `S` variable does not need to be defined. Otherwise, it must be explicitly set:

```
S = "${WORKDIR}/git"
```

Stage 3: Patching Code (do_patch)

Patches (.patch, .diff) mentioned in `SRC_URI` are automatically applied during the build process.

Example patch inclusion:

```
SRC_URI = "file://fix-bug.patch"
```

The build system applies patches with the `-p1` option by default. If necessary, the `striplevel` parameter can be set to adjust the directory stripping level.

Licensing

Every recipe must include license details to comply with Yocto Project requirements.

```
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=<checksum>"
```

Yocto checks `LIC_FILES_CHKSUM` to detect any changes in license text and issues an error if any modifications are found.

Stage 4: Configuration (do_configure)

Software usually provides options to set build-time configurations. The configuration stage is automated for commonly used tools:

- **Autotools**: Requires `configure.ac`
- **CMake**: Requires `CMakeLists.txt`

If no standard configuration files are available, a custom `do_configure` task must be defined in the recipe.

Stage 5: Compilation (do_compile)

This task compiles the source code into executable binaries and libraries.

Stage 6: Installation (do_install)

After compilation, BitBake executes the `do_install` task to copy built files to their designated locations on the target device.

Stage 7: Packaging (do_package)

The `do_package` task splits the generated files into logical components (such as binaries, libraries, debug symbols, and documentation) to ensure efficient packaging and deployment.

Even single binary software can have multiple logical components that need to be packaged separately.

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

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

Copy `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=0835ade698e0
bcf8506ecda2f7b4f302"

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: Build the recipe

```
bitbake myhello
```

`install` keyword

The `install` command:

- Copies files while changing ownership and permissions.
- Optionally removes debugging symbols from executables.
- Combines `cp`, `chown`, `chmod`, and `strip` operations.

WORKDIR

`WORKDIR` is the location where the OpenEmbedded build system builds a recipe. It is structured as follows:

```
${TMPDIR}/work/${MULTIMACH_TARGET_SYS}/${PN}/${EXTENDPE}${PV}-${PR}
```

- `TMPDIR`: Top-level build output directory
- `MULTIMACH_TARGET_SYS`: Target system identifier
- `PN`: Recipe name
- `PV`: Recipe version
- `PR`: Recipe revision

OpenEmbedded Variables

- `S` : Contains the unpacked source files
- `D` : Destination directory (before creating the image)
- `WORKDIR`: Location where OpenEmbedded builds the recipe

- `PN` : Name of the recipe
- `PV` : Version of the recipe
- `PR` : Revision of the recipe

Recipe Explanation

1. **do_fetch:**
    - Since `SRC_URI` points to a local file, BitBake will copy it to `WORKDIR`.

```
bitbake -c fetch myhello
bitbake -c unpack myhello
bitbake -c configure myhello
```

2. **do_compile:**
    - The compiler is invoked to compile `userprog.c`.
3. **do_install:**
    - Specifies where the compiled binary should be installed within the rootfs.
4. **do_package:**
    - Splits and packages the files correctly for installation in the final image.

Adding the recipe to rootfs

```
IMAGE_INSTALL += "myhello"
```

Who defines the fetch, configure, and other tasks?

When `bitbake` is run, the `base.bbclass` file is inherited automatically by any recipe.

Located at:

```
classes/base.bbclass
```

This file contains definitions for standard tasks such as:

- Fetching
- Unpacking
- Configuring
- Compiling
- Installing
- Packaging

These classes are often extended by others such as the `autotools` or `package` classes.

Challenge

Navigate to the `meta` folder and analyze the various recipes related to Git.

Where do I find build logs?

For each individual recipe, a `temp` directory exists under the work directory. This directory is pointed to by the `T` variable within the build system.

To locate it:

```
$ bitbake -e <recipename> | grep ^T=
```

Each task for a recipe generates `log` and `run` files in `${WORKDIR}/temp`.

- Log files are named `log.<taskname>` (e.g., `log.do_configure`, `log.do_fetch`).
- Symbolic links are maintained pointing to the latest log files using `log.<task>`.
- Scripts for each task can be run using the pattern `run.<task>.<pid>`, which contain the build commands.

Logging Information during task execution

BitBake provides logging utilities to trace code execution, available for both Python and Shell scripts.

Python log levels:

- `bb.fatal` - Terminates processing.
- `bb.error` - Displays errors but does not stop the build.
- `bb.warn` - Issues warnings.
- `bb.note` - Provides informational messages.
- `bb.plain` - Outputs a simple message.
- `bb.debug` - Displays debug information.

Shell script log levels:

- `bbfatal`, `bberror`, `bbwarn`, `bbnote`, `bbplain`, `bbdebug`

Difference in behavior:

- Python logging is handled by BitBake and seen on the console.
- Shell script logging outputs messages to the respective task log file.

Example logging in a `do_compile` function:

```
do_compile() {
 bbplain "**********************************"
 bbplain "* Example recipe created by bitbake-
layers *"
 bbplain "**********************************"
}
```

Debug Output

Use the `-D` option to view debug output:

```
$ bitbake -D <recipe>
```

Increasing verbosity:

```
$ bitbake -DDD <recipe>
```

---

oe_runmake

The `do_compile` task runs `oe_runmake` by default if a Makefile is found. If no Makefile is present, it does nothing.

The `do_configure` task runs `oe_runmake clean` by default.

Advantages of oe_runmake over make:

- Passes `EXTRA_OEMAKE` settings to `make`.
- Displays the `make` command.
- Checks for errors.

Example usage:

```
oe_runmake all
```

EXTRA_OEMAKE

To pass additional make options, use the `EXTRA_OEMAKE` variable in the recipe.

```
EXTRA_OEMAKE = "-j4"
```

Handling Makefiles without a clean target

If a Makefile lacks a clean target, use:

```
CLEANBROKEN = "1"
```

This prevents the build system from attempting to run `make clean`.

Challenge

Given the version variable:

```
PV = "0.1+git${SRCPV}"
```

Explore how different recipes use Git-based source code management.

Writing a Recipe for a Git Remote Repository

Yocto supports fetching source code from remote Git repositories as part of the build process.

Step 1: Set `SRC_URI`

Specify the repository URL and protocol:

```
SRC_URI = "git://<URL>;protocol=https"
```

Step 2: Set `S` environmental variable

Define the source directory location:

```
S = "${WORKDIR}/git"
```

Step 3: Set `SRCREV` environmental variable

`SRCREV` determines the specific revision of the repository to build from.

Options for `SRCREV`:

1. **AUTOREV** (latest commit revision)
2. `SRCREV = "${AUTOREV}"`

   The build system fetches the latest version of the repository each time.

3. **Fixed revision** (specific commit SHA)

   ```
   SRCREV = "d6918c8832793b4205ed3bfede78c2f915c23385"
   ```

   Using a fixed revision prevents unnecessary queries to the remote repository.

Specifying a different branch

To fetch a specific branch, use:

```
SRC_URI = "git://server.name/repository;branch=branchname"
```

If no branch is specified, BitBake defaults to the "master" branch.

Fetching from a local Git source

For local repositories, specify:

```
SRC_URI = "git:///home/user/git/myTest/;protocol=file"
```

Working with private repositories

For private repositories that require authentication, Yocto supports SSH:

```
SRC_URI="git://git@github.com/group_name/repo_name.git;protocol=ssh"
```

Ensure SSH keys are set up on the system and added to GitHub.

Using Git tags

The `tag` parameter allows you to specify a repository tag instead of `SRCREV`:

```
SRC_URI = "git://github.com/example/repo.git;tag=v1.0"
```

In this case, `SRCREV` is not required.

Patching the source for a recipe

Yocto builds everything from source, allowing easy code modifications. During the build, OpenEmbedded (OE) creates the following work directory:

`tmp/work/<architecture>/<recipe>/<version>/`

It contains the source in a subdirectory named `<recipename>-<version>` or `git`.

Why not modify source files directly?

1. Changes can be lost when running `bitbake -c clean`.
2. The build system does not detect manual changes without forcing compilation:

`3. bitbake -c compile <recipe> -f`

Creating patches

If the source is a Git repository, patches can be created easily:

1. Make the required changes.
2. Commit the changes.
3. Generate the patch using:

```
git show HEAD > my-patch.patch
git format-patch -1
```

Store patches in a subdirectory where the recipe resides.

Example structure:

```
meta-mycustomlayer/
   ├── recipes-example/
   │    ├── myrecipe/
   │    │    ├── myrecipe.bb
   │    │    ├── files/
   │    │    │    ├── my-patch.patch
```

Yocto will automatically apply patches during the build process.