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.

To check:

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

Types supported:

btrfs, container, cpio, ext2, ext3, ext4, iso, squashfs, ubi, wic, etc.

Creating your own image type

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

This is done through a class that inherits from `image_types`.

It has to 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.

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

Challenge

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

Where do I find build logs?

Every build produces lots of log output for diagnostics and error tracking.

General logs:

Output of `bitbake` is logged to:

```
tmp/log/cooker/<machine>/
```

To view log details:

```
$ cat tmp/log/cooker/<machine>/<timestamp>.log |
grep 'NOTE:.*task.*Started'
```

Recipe-specific logs:

Each recipe has a `temp` directory under the work directory.

To locate the temp directory:

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

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

- Log files: `log.<taskname>` (e.g., `log.do_configure`)

- Run files: `run.<task>.<pid>` (contain commands executed during the build)

Symbolic links are maintained pointing to the last log file using `log.<task>` pattern.

Logging Information during task execution

BitBake provides logging functions for use in Python and Shell scripts:

Python logging levels:

- `bb.fatal`: Terminates processing
- `bb.error`: Displays error without stopping build
- `bb.warn`: Issues warnings
- `bb.note`: Provides information
- `bb.plain`: Prints plain messages
- `bb.debug`: Provides debug output based on the debug level

Shell script logging levels:

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

Example of logging in do_compile:

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

Debug Output

To view BitBake's debug output:

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

Increasing verbosity:

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

---

oe_runmake

The default behavior of `do_compile` is to run `oe_runmake` if a Makefile is found. If no Makefile is found, `do_compile` does nothing.

The default behavior of `do_configure` is to run `oe_runmake clean` if a Makefile is found.

Why use oe_runmake instead of make?

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

Example:

```
SRC_URI =
"git://github.com/example/repo.git;protocol=https"
EXTRA_OEMAKE = "-j4"
```

Handling Makefiles without clean target

If a Makefile does not have a `clean` target, use:

```
CLEANBROKEN = "1"
```

Git Repository in Yocto Recipe

To write a recipe for a remote Git repository:

1. **Set SRC_URI**

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

2. **Set S environmental variable**

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

3. **Set SRCREV environmental variable**

Options for SRCREV:

- **AUTOREV**: Fetches the latest commit automatically
- `SRCREV = "${AUTOREV}"`
- **Specific revision (SHA1 hash):**
- `SRCREV = "d6918c8832793b4205ed3bfede78c2f915c23385"`

Specifying branch:

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

Fetching from local Git sources

For local repositories, use:

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

Using private repositories

For private repositories, use SSH protocol:

```
SRC_URI="git://git@github.com/group/repo.git;proto
col=ssh"
```

Ensure SSH keys are configured.

Fetching specific Git tags

Specify a tag using:

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

`SRCREV` is not required in this case.

Patching the source for a recipe

One of the key advantages of Yocto is that everything is built from source, making it easy to modify any component during the build process.

During the build process, OpenEmbedded (OE) creates a `tmp/work/<architecture>/<recipe>/<version>` directory, known as the **work directory**. This directory contains all the work done to build a recipe.

A subdirectory inside the work directory holds the source of the recipe, typically named `<recipename>-<version>` or `git`, depending on the source provided.

Why not modify source code directly in the work directory?

Although it might seem convenient to directly edit source files and recompile, there are several reasons why this approach is not recommended:

1. **Loss of changes:**
   - Running `bitbake -c clean <recipe>` will erase any manual changes made in the work directory.
2. **Build system tracking:**
   - The Yocto build system does not automatically detect manual changes, requiring a forced recompilation using:

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

Patches

Applying patches to source code is the preferred way to make changes persistently and reproducibly.

Creating patches using Git

If the third-party source code is available as a Git repository, creating patches using Git is the easiest and most efficient method.

Steps to create a patch:

1. Clone or download the repository.
2. Make the required code changes.
3. Commit the changes.
4. Generate a patch using the following command:

```
$ git show HEAD > my-patch.patch
```

Including patches in a Yocto recipe

- The generated patch files should be placed in a sub-directory where the recipe is located.
- The Yocto build system automatically applies these patches when building the recipe.

Example recipe structure:

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

By following this structure, the patches will be applied automatically when the recipe is built.