→ Elements of Embedded linux,
→ Yocto, poky, metadata, Bitbake,
↓
(Recipes, Conf, classes)
→ Build directory
→ Build workflow
→ Assignment operators
→ Creating layers
→ Creating Images from scratch / existing
→ writing recipe
→ Static & Dynamic library
→ SState-cache

1. Elements of embedded linux:-

① Toolchain :- Compiler & other tools needed to create code for your target device. Everything else depends on toolchain.

② BootLoader :- The program that initializes the board & loads the linux Kernel.

③ Kernel :- This is the heart of the system, managing system resources & Interfacing with HW.

④ Root filesystem :- Contains libraries & pgms that are run once Kernel has completed its Initialization.

One more, element can be Collection of pgms specific to your embedded app which make device do what ever it is supposed to do, be it weighing groceries, display movies, control a robot, ...

2. Yocto project:-

Yocto is the smallest SI metric system prefix. [10^{-24}]

1) yocto project provides open source, high quality infrastructure and tools to help developers create their own custom linux distributions for any HW architecture.

History:-

founded in 2010 is an effort to reduce work duplication, provide resources & information catering to both new & experienced users

Collobration of

1) many Hardware manufacturers.
2) Open source operating system vendors &
3) Electronic companies.

Yocto is also a project working group of the Linux foundation.

③ Input & output of yocto project:-

Input:- Set of data that describes what we want, that is our specification. (Kernel Configuration, HW Name, Packages / Binaries to be installed)

Output:- Linux Based Embedded product (Linux Kernel, Root file system, Boot loader, Device Tree)

④ setting up build Machine.

⑤ poky:-

→ poky is a reference distribution of yocto project. The word "reference" is used to mean "example" in this context.

→ yocto project uses poky to build images (kernel, system and application software) for targeted hardware.

At the technical level, it is a combined repository of the Components →1) Bitbake
2) Open Embedded Core
3) meta - yocto - bsp
4) Documentation

Note:- poky does not contain binary files, it is a working example of how to build your own Custom Linux distribution from source.

what is the difference between poky & yocto:-

The exact difference between yocto & poky is yocto refers to the organisation (like one would refer to 'Canonical', the Company behind Ubuntu) & poky refers to the actual bits downloaded (analogous to 'ubuntu')

⑥ → We can take poky & modify, create own Linux distribution

⑥ Metadata:-

Non-yocto:- A set of data that describes & gives information about other data.

Yocto world:-

→ Metadata refers to the build instructions
→ Commands & data used to indicate what versions of SW are used.
→ where are they obtained from

→ Changes or additions to the SW itself (patches) which are used to fix bugs (or) customize the SW for use in particular situation.

Metadata is collection of

① Configuration files (.conf)
② Recipes (.bb & .bbappend)
③ classes (.bbclass)
④ Include (.inc)

⑦ openEmbedded project:-

→ It offers a best-in class cross-compile environment. It allows developers to create a complete linux distribution for embedded systems.

What is the diff b/n openEmbedded and yocto project:-

yocto project & openEmbedded share a core collection of metadata called openEmbedded-Core. However, the two organisations remain separate each with its own focus.

openEmbedded provides a comprehensive set of metadata for wide variety of architectures, features and applications.

↳ Not a reference distribution

↳ Designed to be the foundation for others.

But yocto project focuses on providing powerful, easy to use, interoperable, well-tested tools, metadata and board-support packages (BSPs) for core set of architectures and specific boards.

Open Embedded-Core (oe-core):-

yocto project & open Embedded have agreed to work together and share a common core set of metadata (recipes, classes, associated files): oe-core

⑧ Bitbake:-

→ It is core component of yocto project.
→ It basically performs the same functionality as of make.
→ It's a task scheduler that parses python & shell script mixed code.
→ The code parsed generates & runs tasks which are basically a set of steps ordered according to code's dependencies.
→ It reads recipes & follows them by fetching packages, building them & incorporating the results into bootable images.
→ It keeps track of all tasks being processed in order to

ensure completion, maximizing use of processing resources to reduce build time & being predictable.

⑨ meta-yocto-bsp :-

A Board support package (BSP) is a collection of information that defines how to support a particular HW devices, set of devices, (or) HW platform.

→ Bsp includes information about HW features present on the device & Kernel configuration information along with any additional hardware drivers required.

→ Bsp also lists any additional software components required in addition to generic linux software stack for both essential and optional platform features.

→ meta-yocto-bsp layer in poky maintains several BSPs such as Beaglebone, EdgeRouter, generic versions of both 32 bit & 64 bit-IA machines.

Machines supported :-

→ Texas Instruments Beaglebone (beaglebone)
→ Freescale MPC8315E-RDB (mpc8315e-rdb)
→ Intel X86-based pcs & devices (genericx86 & genericx86_64)
→ ubiquiti Networks EdgeRouter Lite (edgerouter)

Note :- To develop on different HW, you will need to complement poky with Hardware-specific Yocto layers.

⑩ Other poky repositories :-

-meta-poky, which is poky-specific metadata
→ Documentation which contains the yocto project source files used to make the set of user manuals.

Conclusion :- poky includes

→ Some OE Components (oe-core)
→ bitbake
→ demo Bsp's
→ helper Scripts to setup environment
→ emulator QEMU to test the image

poky is the reference distribution of yocto proj.

⑪ Hello world poky → Selecture
⑫ Run Generated image in QEMU
⑬ Build & Run QEMU ARM
⑭ Run QEMU in nographic mode
⑮ Add lsusb in yocto image :

Go to Local.conf, add IMAGE_INSTALL_append = " usbutils" (all small, space)

then again build it. Go boot it & then check

$ lsusb

⇓

It will display usb info.

16. Build & Run CoreImage Sato :-

17. Challenge → Generate & Run Image for "qemumips".

Note :- [From experience]

If you try to run "runqemu" ~~script~~ command, If Console throws "Command not found" then follow the below steps:

1. Again go to poky folder.
2. Source oe --- ../build/

Then now again go to build, run & execute runqemu command.

18. Resources

19. Quiz!

19. Metadata :-

poky = Bitbake + Metadata

Metadata is Collection of

→ Configuration files (.conf)
→ Recipes (.bb & .bbappend)
→ Classes (.bbclass)
→ Includes (.inc)

20. Recipe :

In General,

It is set of instructions that describe how to prepare or make something, especially a dish.

Yocto : A recipe is a set of instructions that describe how to prepare or

A recipe is a set of instructions that is ~~used~~ read & processed by bitbake.

Extension of Recipe : .bb

A recipe describes :

→ where you get source code
→ which patches to apply
→ Configuration options
→ Compile options (Library dependencies)
→ Install
→ License

Example of Recipes :-

dhcp-4.4.1.bb, gstreamer 1.0-1.bb

21. Configuration file :-

Files which hold

→ global definition of variables
→ user defined variables &
→ Hardware Configuration information

They tell the build system what to build & put into image to support a particular platform.

Extension: .conf

Types:-

→ Machine Configuration options (arm, mips, oremux86)
→ Distribution Configuration options (currently poky)
→ Compiler tuning options
→ General common configuration options
→ User Configuration options (local.conf)

22. classes :-

class files are used to abstract common functionality & share it amongst multiple recipe (.bb) files.

To use a class file, you simply make sure recipe inherits the class.

Eg:- inherit classname

Extension: .bbclass

They are usually placed in classes directory inside meta* directory.

Example of classes

Cmake.bbclass → Handles Cmake in recipes.
Kernel.bbclass → Handles building kernels. Contains code to build all kernel trees.
module.bbclass → provides support for building out-of-tree Linux kernel modules.

23. Layers:-

- A collection of related recipes.

(or) * Layers are recipes containers (folders)

Typical naming convention: meta-<layername>

poky has the following layers:

meta, meta-poky, meta-selftest, meta-skeleton, meta-yocto-bsp

24. Why layers :-

Layers provides a mechanism to isolate meta data according to functionality for instance BSPs, distribution configuration etc.

You could have BSP layer, a GUI layer, a distro configuration, middleware (or) an application.

Putting your entire build into one layer limits & complicates future customization & reuse.

$ cd /poky/meta
$ ls
$ vi recipetool → Have a look.

25. Find out layers used by Bitbake build system

Which layers are used by poky build system?

BB layers Variables present in build/conf/bblayers.conf files list the layers Bitbake tries to find.

If bblayers.conf is not present when you start the build, open Embedded build system creates it from bblayers.conf.

Sample when you source oe-init-build-env.script

Command to find out which layers are present

$ bitbake-layers show-layers

Note: you can include any no. of layers from octo project

26. Where to get other layers

https://layers.openembedded.org/layerindex/branch/master/layers/

27. Yocto project Compatible layers:

These layers are tested & fully compatible with yocto project

Open Embedded layer index contains more layers but the content is less universally validated.

28. Image:-

→ An image is the top level recipe, it has description, a license & inherits the core-image class.

→ It is used alongside Machine definition

→ Machine describes Hardware used & its capabilities

→ Image is architecture agnostic & defines how the root filesystem is built, with what packages.

By default, several images are provided in poky

Command to check list of available image recipes:-

$ ls meta*/recipes*/images/*.bb

29. Packages:

Non-yocto:- → Any wrapped (or) boxed object (or) group of objects.

Yocto:- A package is a binary file with name *.rpm, *.deb (or) *.ipkg

A single recipe produces many packages. All packages that a recipe generated are listed in recipe variable.

$ vi meta/recipes-multimedia/libtiff/tiff_4.0.10.bb

30. Explore poky Directories

Poky-source tree :-

bitbake → Holds all python scripts used by bitbake Command. bitbake/bin is placed into PATH environmental Variable so bitbake can be found.

Documentation → All DOCS sources for yocto project documentation Can be used to generate nice PDFs

meta → Contains Oe-core metadata

meta-poky → Holds the Configuration for poky reference distribution Local.conf.sample, bblayers.conf.sample are present here.

meta-Skeleton → Contains template recipes for BSP & Kern. Dev.

meta-yocto-bsp → Maintains several BSPs such as Beaglebone, EdgeRouter.

Scripts → Contains Scripts used to set up the Environment, der tools to flash generated images on target.

License → License under which poky is distributed.

31. Explore Build folder :-

Conf

→ when you run source poky/oe-init-build-env, it will create a "build" folder in that directory. Inside build folder, it will create "conf" folder which contains two files:

① Local.conf ② bblayers.conf

① Local.conf → Configures almost every aspect of Build System which contains local user settings

DL_DIR: Where to place downloads.

→ During a first build the system will download many different source code tarballs from various upstream projects. These are all stored in DL_DIR. The default is a downloads directory under TOPDIR which is the build directory.

TMP_DIR: Where to place build output. This option specifies where the bulk of the building work should be done & where BitBake should place its temporary files (source extraction, Compilation) & output.

Note: Local.conf file is a very convenient way to override several default Configurations over all yocto project's tools.

→ Essentially we can change (or) set any variable, for example add additional packages to an image file.

→ Though it is convenient, it should be considered as a temporary change as the build/conf/local.conf file is not usually tracked by any Source Code Management System.

32. BB_NUMBER_THREADS :-

→ Determines the number of tasks that Bitbake will perform in parallel

Note :- These tasks are related to bitbake & nothing related to compiling.

Defaults to the number of CPUs on the system.

$ bitbake -e core-image-minimal | grep ^BB_NUMBER_THREADS=

33. PARELLEL_MAKE :-

→ Corresponds to -j make option

→ specifies the no. of processes that GNU make can run in parallel on compilation task.

→ Defaults to the number of CPUs on the system

$ bitbake -e core-image-minimal | grep ^PARELLEL_MAKE=

34. Where should we place contents of local.conf ?

In general everything in your local.conf should be moved to your own distro configuration.

finally, you should set DISTRO to your own distro in local.conf

35. other directories :- [inside build folder]

downloads - download upstream tarballs/git repositories of the receipes used in build

SState-cache - shared state cache

tmp - Holds all build system output

tmp/deploy/images/machine - Images are present here

Cache - Cache used by bitbake's parser.

36. Build Workflow :-

1. Developers specify architecture, policies, patches & configuration details.
2. Build system fetches & downloads the source code from the specified location. → support downloading tarballs & source code repositories systems such as git/svn.
3. Extract the sources into local work area.
4. Patches are applied.
5. Steps for configuring & compiling the software are run.

6. Installs the software into temporary staging area depending on the User Configuration, deb(or) rpm(or) ipk binaries are generated.
7. Build system generates a binary package feed that is used to create final root file image.
8. Finally generates the file system image and Customized Extensible SDK (eSDK) for application development in parallel.

[search "yocto build workflow" image].

3. Images generated by poky build :-

The build process writes images out to Build directory inside tmp/deploy/images/machine/ folder.

1. Kernel-Image :

→ A Kernel binary file.
→ The KERNEL_IMAGETYPE variable determines the naming scheme for Kernel image file.
$bitbake -e core-image-minimal | grep ^KERNEL_IMAGETYPE=

2. root-filesystem-image :-

Root filesystems for the target device (eg:- *.ext3 or *.bz2 files).

The IMAGE_FSTYPES variable determines the root filesystem image type

$ bitbake -e core-image-minimal | grep ^IMAGE_FSTYPES=

3. Kernel modules :-

Tarballs that Contain all the modules built for the Kernel.

4. boot loaders :- If applicable to target machine, bootloaders supporting the image.

Symlinks :-

Symbolic link pointing to the most recently built file for each machine. These links might be useful for the external scripts that need to obtain latest version of each file.

3. Save disk space :-

Yocto build system can take a lot of disk space during the build. But bitbake provides options to preserve the disk space.

You can tell bitbake to delete all source code, build files after building a particular recipe by adding

the following line in local.conf file.

INHERIT += "rm_work"

Disadvantage: Difficult to debug while build fails of any recipe.

for example: If you want to exclude bitbake deleting the source code of a particular package, you can add it in

RM_WORK_EXCLUDE += "recipe-name"

E.g: RM_WORK_EXCLUDE += "core-image-minimal"

41 ⇔ 55 :- Yocto on Beaglebone → watch lectures

(or) access the resources for better understanding of create partition on SD card.

56. Yocto project releases:-

Naming Convention

Yocto project releases follow the naming convention

: Major.minor.patch number

Eg:- yocto-2.7.3

→ Major release number changes imply compatability changes with previous releases.

→ Minor release number changes imply significant changes up to, but not including compatability changes.

→ Minor rev number changes are for minor issues such as simple bugfixes, security updates, ..etc.;

Poky Releases

poky releases has code names as well as Major. Minor. patch numbering.

Yocto project / poky Release are released at the same time.

57. Why poky has codenames ?

Major Releases: Every six months in April & October

Minor Releases: No particular schedule

Driven by the accumulation of enough significant fixes (or) enhancements to the associated major release

Concept of Codenames

Branches of metadata with the same codename are compatible with eatch other.

58. Types of Releases:-

a) Milestone Releases

Major Release is divided into milestone releases.
Milestone releases are used as:
→ checkpoints for our progress
→ preview for the public community
→ To manage changes in the middle of release.

Each milestone lasts about 4-8 weeks and has one planning week, several development weeks, stabilization weeks and one release week

b) Point releases

A point release is a minor release such as 1.0.X
Point releases are necessary to address the following
→ Issues building on Current Linux distributions.
→ fix critical bugs and Security issues (CVEs) for the first 6 months after release
→ fix Security issues for the first year after release

59. Release Lifecycle:-

There are two possible lifecycles a release may follow:

Initial Release → Stable → Community → EOL
↳ End of life

(or)

Initial Release → LTS → Community → EOL

Stable Vs LTS Release Difference

Stable releases are maintained for seven months.
LTS releases are maintained initially for two years
LTS releases are pre-announced & known to be LTS in advance (usually every two years)

Community Support

Branch transitions to Community Support after the last Stable dot release identified by Stable/LTS maintainer.
A call for a Community Support maintainer is sent to the mailing lists.
→ A six week waiting period
→ If there is no new maintainer, then branch goes

to into EoL Status.

Note: Branches only have Community Support status if there is an active Community member willing to step into maintainer role for that Series.

60: Layers & Branches:-

Layers being developed simultaneously by several different parties, different flavor of yocto & subsequent layers have to be split into branches in Git.

Most Bsp layers will only work with OE-Core of the same branch.

Mixing branches among your layers will end up in Conflicts.

For example, the so called bbappends, sometimes are tied to specific version number & will break the build if those versions are not found in layers.

Which branch to choose?

you should evaluate all the layers and find a Compromise between

→ Getting the latest stable branch

→ Getting the latest branch supported by all Layers.

Note: yocto branch you decided to use, Could not be supported by your Current host Linux distribution.

Eg: I want to use layer which forces me to stick with Krogoth, but this branch is not tested with newer distributions such as Ubuntu 16.04 (or) 18.04

61. Bsp-Layer:-

→ A Collection of information (meta data) that defines how to support 1) a particular HW device

2) set of devices

3) HW platform

62. Bsp Layer Naming Convention:

meta-<bsp-name>

How to find out what all HW devices are supported

Conf/machine/*.conf

63. Meta-ti Layer:-

↳ Bsp Layer for Texas Instrument HW

64. meta-ti vs meta-yocto-bsp :-

meta-yocto-bsp:

→ provides "reference" BSPs for each of supported arch.

→ it is based on mainline Kernel/bootloader.

meta-ti:

→ offical TI BSP that provides latest WIP "staging" Kernel & bootloader.

65. Add layer [meta-ti]

$ bitbake-layers show-layers

By default, only three layers are included.

① meta ② meta-poky ③ meta-yocto-bsp

How to add meta-ti layer?

① manual approach:-

open bblayer.conf file & add it

BBlayers ... Source/meta-ti

② Automatic:

$ bitbake-layers add-layer <path-to-new-layer>

$ bitbake-layers add-layer ~/yocto-training/source/meta-ti/

Then go & check bblayers.conf file, it will be automatically updated.

66. Build yocto Image using meta-ti:-

In (65), we have added meta-ti layer.

Build for beaglebone

① Source the env. Script

$ Source poky/oe-init-build-env

② open local.conf & set Machine to beaglebone

Machine = 'beaglebone'.

③ Add INHERIT = "rm_work" to save diskspace.

④ $ bitbake core-image-minimal

⑤ Build o/p:- $Builddir/tmp/deploy/images/beaglebone

(67) Flashing Image on SD Card using WIC utility :-

→ WIC images are SD Card images & Can be directly written into Sd-Card.

→ Core-image-minimal-beaglebone.wic.xz is the Compressed wic image.

→ It Can be uncompressed using unxz utility.

$ unxz Core-image-minimal-beaglebone.wic.xz

$ ls -lh Core-image-minimal-beaglebone.wic

Flash it to sd Card

$ lsblk

$ sudo dd (space) if=Core-image-minimal-beaglebone.wic (space) of=/dev/sdb (space) bs=4096 (space) status=progress && sync (space)

section → Raspberry pi3

(5) - learn yocto - 2

① Variable Assignment - Hard :-

VARIABLE = "value".

The following example sets VARIABLE to "value".

Eg. MACHINE = "raspberrypi3"

This assignment occurs immediately as statement is parsed.

It is hard assignment.

If you include leading (or) trailing spaces as part of an assignment, 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'

② Find value of variable :-

→ For Configuration changes in file, use the following :-

$ bitbake -e

This Command displays variable values after Conf files (ie. local.conf, bblayers.conf, bitbake.conf ...) have been parsed.

→ For recipes changes, use the following

$ bitbake recipe -e | grep VARIABLE="

$ cat Conf/bblayers.conf

Check `BBPATH` variable. it will be : BBPATH = "${TOPDIR}"

Now how we will check variable?

$ bitbake -e | grep ^BBPATH=

o/p: BBPATH = "/home/.----"

Anything that defined in conf files => global

" " in recipes => local

Now how will we check variable inside recipes?

$ bitbake core-image-minimal -e | grep ^LICENSE=

o/p:
LICENSE=MIT

↓ before that atleast do

$ bitbake core-image-min -e

③ split variable assignment over multiple lines:-

Line joining

Bitbake joins any line ending in backslash character ("\") with the following line before parsing statement.

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

```
BBLAYERS ?= " \
  /home/.---- \
  /home/.---- \
"
```

④ Variable Assignment - Soft

Setting default value (?=)

?= is used for soft-assignment for variable.

What's benefit?

→ Allows you to define a variable if it is undefined when the statement is parsed, If variable has value, then soft assignment is lost.

Eg:-

MACHINE ?= "genuarm"

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

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

Note: Assignment is immediate.

What happens if we have multiple ?=

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

How to check?
$ bitbake -e | grep ^MACHINE

Eg:-

$ vi Conf/local.conf

① Case 1:-

#MACHINE ?= "genuarm"

....

If we come & add MACHINE = genux86 ← Hard assignment.

=> value of variable MACHINE is updated with genux86.

Case 2:-

Now,

MACHINE = "genuxx86" → hard assignment always win

MACHINE ?= "genuarm"

. . .

Now value of MACHINE is genuxx86.

Case 3:- [Both soft assignments]

Now,

MACHINE =? "genuarm" → This will win [First soft assign.]

MACHINE =? "genuxx86"

⑤ Variable Assignment - Weaker default value:-

Setting a weaker default value (??=)

Weaker default value is achieved using (??=) operator.

Difference b/n (?=) and (??=)

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

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

Eg:

MACHINE ??= "genuxx86"

MACHINE ??= "genuarm"

If MACHINE is not set, value of MACHINE = "genuarm".

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

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

Note: ":=" (or) "?=" assignment will override value set with "??="

Case 1:-

MACHINE ??= "genuxx86"

MACHINE ??= "genuarm"

MACHINE = "genuarm64" → This will win.

⑥ Variable Expansion - Assigning value of other variables:-

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

Eg:- like in shells

$ export VARIABLE = value

$ echo $VARIABLE

o/p: value → is used for variable

A = "hello"
B = "${A} World" } in local.conf file

$ 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"

$ bitbake -e | grep ^A=

What happens if C is not defined in above?

o/p: string is kept as is [bitbake -e | grep ^B=]
B="\ ${C} world hello"

bitbake -e | grep ^A=

o/p "\ ${C} world hello"

⑦ Immediate Variable Expansion (:=)

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

$ bitbake -e | grep ^B,

①
A="1"
B="B:${A}"
A="22"
C := "C:${A}"
D = "${B}"

last one hard assign

o/p:- A=22
B=B:22 → See difference with Case ②
C=C:22
D=B:22

Case ②
A=""
B := "B:${A}"
A=22
C := "C:${A}"
D = "${B}

o/p:
A=22
B=B:"" → This is the difference
C=C:22

⑧ Appending operators

+= :- A = "hello"
A += "world"
o/p: bitbake -e | grep ^A=
A="hello world" → space

.= :- A = "hello"
A .= "world"
o/p: A="helloworld" → no space

Difference b/n += & .= is space is automatically added in +=
These operators take immediate effect during parsing.

⑨ prepending operators

=+ :- A = "world"
A =+ "hello" (space)
o/p: A="hello world"

=. :- A = "world"
A =. "hello" (space)
o/p: A="helloworld"

Now we have =, ?=, ??=, /, :=, +=, .=, =+, =., $

⑩ overriding style syntax :-

Appending & prepending (override style syntax)

you can also append & prepend a variable's value using an override style syntax..

When you use this syntax, no spaces are inserted..

Eg: A = "hello"
A_append = " world" → manually adding space
B = "text"
B_append = "world")
C = "full"
C_append = "house".

o/p: bitbake -e | grep ^A=
A = "hello world"
B = "textworld".
C = "housefull".

(11). Removal 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.

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

$ bitbake -e | grep ^FOO=
o/p: 456 789 123456 456 456"

(12) Advantage of override style operations

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

Eg: $ vi ../source/poky/meta/recipes-core/images/core-image-minimal.bb

=> IMAGE_INSTALL = "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}

Now vi/conf/local.conf → hard assignment
in → add in local.conf

IMAGE_INSTALL += "usbutils"

$ bitbake -e core-image-minimal | grep ^IMAGE_INSTALL=

o/p: IMAGE_INSTALL = "packagegroup-core-boot"

But value usbutils is not appended, only previous value is there.

Now modify in local.conf,

IMAGE_INSTALL_append = "usbutils"

old syntax -> N.T working
new syntax
IMAGE_INSTALL : append =

Now check value.

o/p:- IMAGE_INSTALL = "packagegroup-core-boot usbutils"

(iy override style operations provide guaranteed operations.

Why IMAGE_INSTALL += "usbutils" not happen?

Bcoz, order of parsing may be differ. Let's it come to

local.conf file, it will choose/consider "usbutils". Then it go to recipe, it will see IMAGE_INSTALL = , it will remove "usbutils".

But if we use append, usbutils also been considered & added.

Just for understanding the parsing order :-

Now instead of local.conf, go & change in recipe like below

IMAGE_INSTALL = "packagegroup-core-boot ${CORE_IMAGE_EXTRA_INSTALL}"

IMAGE_INSTALL += "usbutils"

↳ append operator,

:

Now result, o/p :- "packagegroup-core-boot usbutils"

Here we did not use "append keyword", bcoz we edited & arranged directly in recipe file

(13) Quiz.

(14) What is 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 technical restriction.

Eg :- meta-mycustom

(15) Why create a meta layer?

Depite most of the customization can be done with 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

(16) ~~Why do we need to create layer?~~ [meta-ti, meta-raspberry]

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.

(15) Layers in detail :

Depending on type of layer, add the content :

→ If layer is adding support for Machine, add the machine configuration in conf/machine

→ If layer is adding distro policy, add the distro configuration in conf/distro

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

Recipe directories inside layers:-

By Convention, recipes are splitted into Categories
The most difficult part is deciding in which Category your recipe will go.
By checking what was already done in official layers should give you 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 same name in multiple layers.
A higher numeric value represents a higher priority.

Go & read, $ cd ../source/poky/meta/recipes.txt

$ bitbake-layers show-layers

O/P:
layer	path	priority
meta		5
meta-poky		5

(16) Creating Layer:-

There are two ways to create your own layer.
1. Manually
2. Using Script

Manually:
Step1:- Create directory for layer. Eg:- 'meta-mylayer'
Step2:- Create conf/layer.conf
→ you can simply copy meta-oe's one & just change "openembedded-layer" to something appropriate for your layer; you may also want to set the priority as appropriate.
Step3:- update bblayer.conf in build folder with new layer.

$ mkdir meta-mylayer
$ cd meta-mylayer
$ ls
$ mkdir conf
$ cd conf
$ cp ../../poky/meta-skeleton/conf/layer.conf
$ vi layer.conf
⇒ Rename BBFILE_COLLECTIONS += "skeleton"
↓
skeleton with meta-mylayer in all occurrences in that file. Replace all.

If we want to priority change,

BBFILE_PRIORITY_mylayer = "1"

then

$ cd build/

$ vi Conf/bblayers.conf

$ add as

BBLAYERS ?=" \

/home/linux ---- /meta

/home/ - - - - /meta-poky

/home/ - - - - - /meta-mylayer

"

17. Creating layers using bitbake layer Command:-

you can create your own layer using bitbake-layers create-layer command

$ bitbake-layers create-layer --help

→ Tool automates the layer creation by setting up a subdirectory with layer.conf configuration file, a recipes-example subdirectory that contains an example .bb recipe, a licensing file, a README

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

Default priority of layer is 6.

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

$ bitbake-layers show-layers.

18. Layer Configuration (layer.conf) → just explore variables in layer.conf.

19. Script to check layer compatibility:-

yocto-check-layer.

→ This script provides you a way to access how compatible your layer is with yocto project.

→ you should use this script if you are planning to apply for Yocto project Compatible program.

~~cd~~ ⇒ go to meta-mylayer path.

$ yocto-check-layer meta-mylayer

Note:-
layer should not be in /build/bblayers.conf

20. challenge → find out in meta folder which one is used a lot : += (or) _append.

(21) Image :-

Image is a top level recipe. (It inherits an image.bb class)

Building an image creates an entire Linux distribution from source

→ Compiler, tools, libraries
→ Bsp: Bootloader, kernel
→ Root filesystem : → Base OS, services, Applications, etc..

(22) Creating Custom images :-

you often need to create your own Image recipe in order to add new packages (or) functionality.

Two ways:

a) Creating an image from scratch
b) extend an existing recipe (preferable)

(23) package-group :-

→ A package group is a set of packages that can be included on any image.
→ A package group can contain a set of packages.
→ using package group name in IMAGE_INSTALL variable install all the packages defined by the package group into root filesystem of target image.

There are many package groups. There are present in subdirectories named "packagegroups"

$ find . recipe-*/ -name 'packagegroups'

(24) Creating an image from scratch :-

The simplest way is to inherit the core-image.bb class, as it provides a set of image features that can be used very easily.

Check:- [in side meta]

$ cd vi classes/core-image.bb class

we can see "inherit image" in that file. so this file has already inherit 'image'. so no need of inherit 'image' in our recipe. So we can directly use core-image.bb class

→ inherit core-image

which tells us that the definition of what actually gets installed is defined in the core-image.bb class.

→ Image recipes set IMAGE_INSTALL to specify the packages to install into an image through image.bb class.

Creating:

(1) Check already our meta-mylayer is there
(2) $ cd meta-mylayer
(3) $ cd recipes-examples

④ $mkdir images

⑤ $ cd images/

⑥ $ vi vicky-image.bb

⑦ add below statement in that file,

inherit core-image , save it.

⑧ $cd build ⇒ just to check packages

⑨ $ bitbake ~~vicky~~ -e vicky_image | grep ^IMAGE_INSTALL=

o/p:- IMAGE_INSTALL = " packagegroup-core-boot packagegroup-...

how it was there? bcoz we inherited core-image.

Check → $ vi /meta/classes/recipes core-image.bb ⇒ in build infra

We can see IMAGE_INSTALL ? = "${CORE_IMAGE_BASE_INSTALL}"

CORE_IMAGE_BASE_INSTALL = "

packagegroup-core-boot \

packagegroup-base-extended \

"

And one more thing that is soft assignment ?

Now try to do different,

$ cd meta-mylayer/recipes-examples/images

$ vi vicky-image.bb

add → IMAGE_INSTALL = "packagegroup-core-boot"

↓ hard assignment

Now same step ⑨, this time it will show only

o/p:- IMAGE_INSTALL = "packagegroup-core-boot"

⑩ $ cd build

$ bitbake vicky-image ⇒ Generating our image

⑪ once generated, explore cd build/tmp/deploy/images/qemu-arm/..

⑫ Adding package into existing image :-

Now inside our imag bb file, only packagegroup-core-boot there.

$ cd build/tmp/deploy/images/qemu-arm

$ runqemu

→ login as root

Now requirement is "we need usb related stuffs in our developed imag". ☺

Inside imag [Inside the same login terminal]

$root@qemuarm: ~# lsusb

o/p:- -sh: lsusb: not found

Now how do we add lsusb ?

$come to poky folder.

$ cd meta-mylayer/recipes-examples/images

$ vi vicky-image.bb

add ~ IMAGE_INSTALL += "usbutils"

$ cd build

Confirm IMAGE variable is updated,

$ bitbake -e vicky-image | grep ^IMAGE_INSTALL=

o/p:- IMAGE_INSTALL="packagegroup-core-boot usbutils"

↑ Note.

$ bitbake vicky-image => Generate new image

Then go & check root@qemuarm:~# lsusb.

o/p:- => It will be there.

(26) Reusing an existing image:-

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

→ This makes code maintenance easier & highlights the functional differences

Instead of develop from scratch,

→ delete meta-mylayer/recipes-examples/vicky-image.bb

→ again create (ie) $ vi meta-mylayer/recipes-examples/vicky-image.bb (new)

add → require recipes-core/images/core-image-minimal.bb

Now go & check $ bitbake -e vicky-image | grep ^IMAGE_INSTALL=

o/p:-

IMAGE_INSTALL = "packagegroup-core-boot".

Now modify (add for lsusb)

$ vi meta-layers/recipes-examples/vicky-image.bb

$ add ~> IMAGE_INSTALL += "usbutils"

$ bitbake vicky-image => Generating image

(27) Customizing Images using custom IMAGE_FEATURES and EXTRA_IMAGE_FEATURES

→ Another method of customizing your image to enable (or) disable high-level images features by using IMAGE_FEATURES & EXTRA_IMAGE_FEATURES variables.

→ Both are made to enable special features for image, such as empty password for root, debug-image, special packages, x11, splash, ssh-server.

Best practice is to,

Use IMAGE_FEATURES from recipe

use EXTRA_IMAGE_FEATURES from local.conf

(28) How Image feature actually work?

TO understand how these features work, best reference is meta/classes/core-image.bb class.

→ This class lists out the variable IMAGE_FEATURES of which most map to package groups while some, such as debug-tweaks and read only-rootfs, resolve as general Conf. settings.

→ In summary, file looks at the Content of IMAGE_FEATURES variable & then maps (or) Configures feature accordingly.

→ Based on the information, build system automatically adds the app. packages (or) Conf. to IMAGE_INSTALL variable

(29) Example of IMAGE_FEATURES:-

→ To illustrate how you can use these variables to modify our image, Consider an example that selects SSH server.

→ yocto project ships with 2 SSH servers we can use with our images: Dropbear & openSSH.

→ Open SSH is a well known standard SSH server implementation.

→ Dropbear is a minimal SSH server appropriate for resource Constrained environments.

→ By default, Core-image-sato image is Configured to use Dropbear. Core-image-full-cmdline & Core-image-lsb images both include openSSH.

→ Core-image-minimal image does not Contain SSH server.

→ Edit & modify vicky.bb

"Instead of "inherit core-image",
"require recipes-core/images/core-image-minimal.bb"

add: → IMAGE_INSTALL += "usbutils"

IMAGE_FEATURES = "ssh-server-dropbear debug-tweaks"

ssh Connection

for without password.

(30) debug-tweaks

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

→ debug-tweaks features enable password-less login for root user

Advantage: makes logging in for debugging (or) Inspection easy during development.

Disadvantage: Anyone can easily log in during production. So you need to remove 'debug-tweaks' feature from production img.

(31) Read only Root filesystem :- → (Not working in latest linux)

Why do we need read-only rootfs:-

→ Reduce wear on flash memory

→ Eliminate system file Corruption

How to do it?

To create read only filesystem, simply add "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 file

(32) splash screen:-

IMAGE_FEATURES += "splash"

(or)

EXTRA_IMAGE_FEATURES += "splash"

Note:- should not run in "nographic" (i.e) runqemu qemuarm

(33) Some other features:-

tools-debug: Installs debugging tools such as strace & gdb

tools-sdk: Installs a full SDK that runs on the device

(34) Other language support :-

IMAGE_LINGUAS

specifies the list of locales to install into image during root filesystem Construction process.

IMAGE_LINGUAS = "zh-cn".

Inside qemu, [sign as root]
Inside booting terminal

$ locale -a

(35) IMAGE_FSTYPES :-

→ This variable determines the root filesystem image type

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

Image formats instructions are delivered in poky:

meta/classes/image_types.bbclass

$ bitbake -e <image-name> | grep ^IMAGE_FSTYPES=

Types supported

btrfs

Containers

cpio, cpio.gz, cpio.lz4, cpio.lzma, ...ext2...

(36) Create own image type [For Raspberry pi]

(37) Different Image Name:-

Check in tmp/deploy/... /qemuarm/

$ ls

you can see

core-image-vicky-qemuarm-.conf

vicky-qemuarm-.ext4

If I would like to change the image name

IMAGE_NAME:-

The name of the output image files minus the extension.

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

Eg: In tmp/deploy/images:-

vicky-genuarm-20230220120949.rootfs.ext4.

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

Simply add IMAGE_NAME = "myimage" in vicky.bb

(38) IMAGE_MANIFEST:-

→ The manifest file for the image.

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

→ This file contains package information on line-per-package basis as follows:

packagename packagearch version

The image class defines the manifest file as follows:

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

Eg: In tmp/deploy/images ...

$ vi vicky-genuarm.manifest

(40) Recipes:

→ They are fundamental components in yocto.

→ It is a text file with filename *.bb

→ Each SW component build by OE build system require a recipe to define component.

→ A recipe contains info about single piece of SW.

What information is present in recipe?

Information such as

→ Location from which to download unaltered source.

→ Any patches to be applied to that source.

→ Special conf. to apply.

→ How to compile the source files

→ How to package the compiled o/p.

41. Recipe file format:-

File format: <base_name>_<version>.bb

Eg: dropbear_2019.78.bb in poky/meta/recipes-core/dropbear has

basename: dropbear

version: 2019.78

Note: use lower-cased characters & do not include the reserved suffixes -native, -cross, -initial (or) -dev

42. How to build recipes:-

→ Yocto's build tool bitbake parses a recipe & generates list of tasks that it can execute to perform build steps.

$ bitbake basename

The most important tasks are

do_fetch → Fetches the source code

do_unpack → unpacks the source code into working directory

do_patch → Locates patch files & applies them to the source code.

do_configure → Configures the source by enabling & disabling any build-time & configuration options for sw being built.

do_compile → Compiles the source in compilation directory.

do_install → Copies the files from compilation directory to the holding area

do_package → Analyzes the content of holding area & splits it into subsets based on available packages & files.

do_package_write_rpm Creates the actual RPM packages & places them in the package feed area.

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

do_configure,
do_compile and
do_install ones.

The remaining tasks are automatically defined by YP build system.

→ Above task list is in the correct dependency order. They are executed from top to bottom.

→ you can use -c argument to execute specific task of recipe.

$ bitbake -c compile dropbear

To list all tasks of particular recipe,

$ bitbake <recipe name> -c listtasks

(43) Recipe fetch stage:-

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 SRC_URI variable.

→ your recipe must have SRC_URI variable that points to where source is located.

→ SRC_URI variable must define each unique location for your source files.

Bitbake supports fetching source code from git, SVN, https etc

URI scheme syntax: scheme://url;param1;param2

Scheme can describe a local file using "file://" (or) remote location with https://, git://, ...

By default, sources are fetched in $BUILDDIR/downloads

Examples of SRC_URI,

→ busybox-1.31.0.bb : SRC_URI = "https://busybox.net/downloads/"

The do_fetch task uses the prefix of each entry in the SRC_URI variable value to determine how to fetch source code.

Note: Any patch files present needs to be specified in SRC_URI

(#4) Stage 2 : Unpacking (do_unpack)

→ All local files found in SRC_URI are copied into recipe's working directory, in ${BUILDDIR}/tmp/work/

→ When extracting a tarball, bitbake expects to find the extracted files in directory named <application>-<version>. This is controlled by S variable.

→ If you are fetching from SCM like git (or) SVN (or) your file is local to your machine, you need to define S.

If scheme is git, S = ${WORKDIR}/git

(#5) 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 (Eg: diff.gz) are treated as patches.

→ do_patch task automatically applies these patches.

→ Build system should be able to apply patches with "-p1" option

(#6) Recipe Licensing :-

→ your recipe needs to have both LICENSE & LIC_FILES_CHKSUM variables.

LICENSE → This variable specifies license for SW.

Eg: GNU,

LICENSE = "GPLv2"

For standard licenses, use the names of files in meta/files/common-licenses

LIC_FILES_CHKSUM :-

→ OE build system use this variable to make sure license text has not changed.

→ LIC_FILES_CHKSUM = "file://COPYING;md5=xxx"

(#7) Recipe Configure Stage :-

Stage 4 : Configuration (do_configure)

Most SW provides some means of setting build-time config options before compilation.

(48) Recipe Compile, Install & package stage:-

Stage5: do-compile

do_compile task happens after source is fetched, unpacked & configured.

Stage6: Installation (do_install)

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

Stage 7: packaging (do-package)

→ do_package task split the files produced by recipe into logical components.

→ It will ensure that files are split up & packaged correctly.

49. Recipe for C pgm:

```
$ cat userproc.c
#include <stdio.h>
:   → Helloworld pgm
```

now want to include in rootfs?

Ans:- Any sw component needs to be part of rootfs, should need recipe.

Step1:- Create a file userprog.c

Step2:- Create a folder in layer recipes-example 'myhello'. (→ [meta-mylayer])

$ mkdir myhello → should be same name of recipe that we are going to create.

Step3: create 'files' folder inside 'myhello' folder & copy userprog.c inside this folder.

Step4: Create a file called 'myhello-0.1.bb' (→ version)

(In general, create files folder in -example folder. There should be our recipe & image-recipe)

```
DESCRIPTION = "Simple helloworld application"
LICENSE = MIT
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;
                   md5=083..."

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}
}
```

(→ got from
$ cd meta/files/common-licenses
$ md5sum MIT)

(If files are local, it will be automatically copied to WORKDIR.
If we have tar file, it will be extracted to name of tar file.)

Step5:- bitbake myhello

Explanation:

install -d ${D}${bindir} → permission.

install -m 0755 userprog ${D}${bindir}

→ first Create bindir. In yocto, it points to user bin directory

→ We want to Copy "userprog" to userbin.

→ D ⇒ actually points to 'image' folder

⇒ Creating bindir in image folder

50. Install Keyword:-

→ install not only copies files but also changes its ownership & permissions & optionally removes debugging symbols from executables.

→ It Combines cp with chown, chmod & strip.

51. WORKDIR:-

→ Location of work directory in which OE - build system builds a recipe.

→ This directory is located with TMPDIR directory structure & is specific to the recipe being built & system for which it is being built.

WORKDIR is defined as follows:

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

TMPDIR: Top-level build o/p directory

MULTIMACH_TARGET_SYS: target system identifier

PN: The recipe name

EXTENDPE: Mostly blank

PV: recipe version

PR: recipe revision

52. OpenEmbedded Variable:-

S: Contains unpacked source files for given recipe

D: Destination directory (root directory of where files are installed, before creating img)

WORKDIR: Location where OpenEmbedded build system builds a recipe (ie: does work to create package).

once myhello recipe is added, [ensure meta_mylayer is added]

$ bitbake -e myhello | grep ^WORKDIR=

"

^S=

^D=

^PN=

^PR=

^PV=

$ bitbake -c listtasks myhello

$ bitbake myhello

53. Exploring WORKDIR:

$ cd /tmp/work/core2-64-poky-linux/myhello/0.1-r0

54. Recipe build in deep:-

$ bitbake -c cleanall myhello

Now apart from tmp folder, nothing will be there.

Step-by-step :- [rm -rf tmp]

① $ bitbake -c fetch myhello

⇒ Now tmp folder is created with lot of files.

Eg:- vi log.task_order

vi log.do_fetch

② $bitbake -c unpack myhello

go & check build/tmp/work/core2-64-poky-linux/myhello/0.1-r0

There will be userprog.c.

③ $ bitbake -c configure myhello

there will be recipe-sysroot, recipe-sysroot-native

what is sysroot?

→ Contain needed headers & libraries for generating binaries that run on target architecture.

recipe-sysroot-native:-

Includes the build dependencies used in host system during build process.

It is critical to cross-compilation process because it encompasses the compiler, linker, build script tools & more.

recipe-sysroot:

→ libraries & headers used in target code

④ $ bitbake -c compile myhello

there will be 'userprog' obj file created

⑤ $ bitbake -c install myhello

there will be image folder. Inside that usr/bin ⇒ directory is created. [that's we defined in hellovicky-0.1.bb file]

Then check permission for helloworld 'userprog' binary. [we already provided in .bb file].

⑥ $ bitbake -c package myhello

there will be lot of package folders available.

⑦ Just checking for inserting to rootfs :-

$ bitbake vicky → [image] → Note already vicky.bb is there. [(i.e) inside (or) require recipes-core/images/core-image-minimal.bb inherit core-image]

build will be started.

→ $ runqemu nographic

⇒ Linux will be booted.

Inside check /usr/bin & search for 'userprog'.

It will be not there.

8. Now inserting our recipe to image

add
↳ IMAGE_INSTALL:append = "myhello";

9. $bitbake vicky
→ check /usr/bin/userprog.

10. Execute userprog in booting OS

Who defines the fetch, configure & other tasks?

(↓ when bitbake is run to build a recipe, base.bbclass file gets inherited automatically by any recipe.
→ we can find it in oe-core/meta/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) & packaging (empty by default).

→ These classes are often overridden (or) extended by other classes as autotools class (or) the package class.

OS challenge → write a recipe for c code which uses header file (two files: .c/.h)

① create file vi hello.h & define some macro in that.
② add hello.h in userprog.c & print value of macro in c file.
③ both hello.h & userprog.c file should be in same folder.
④ tar -cvf userprog.tar hello.h userprog.c
⑤ Now userprog.tar will be created in 'files' folder.
⑥ gzip userprog.tar
⑦ Now userprog.tar.gz will be created in 'files' folder.
⑧ Edit in myhello_0.1.bb file

(i) SRC_URI = "file://userprog.tar.gz"

(ii) install -m 0755 userprog ${D}${bindir}

⑨ $ bitbake -c cleanall myhello.
⑩ $ bitbake -c fetch myhello
⑪ $ bitbake -c unpack myhello
⑫ $ bitbake -c configure myhello
⑬ $ bitbake -c compile myhello
⑭ $ bitbake -c install myhello
⑮ $ bitbake vicky
⑯ Go & check /usr/bin/userprog in booted linux console & execute it.

58. Introduction:-

Where do I find build Logs?

→ Every build produces lots of log o/p for diagnostics and error chasing.

→ o/p of bitbake gets logged to tmp/log/Cooker/<machine>/

$ cat tmp/log/cooker/<machine>/<timestamp.log | grep

'NOTE:.*task,*started'.

$ cd build/tmp/log/cooker/qemux86-64

$ ls

20230223135619.log . . . .

- - -

* There will be "Console-latest.log" (i.e) symbolic link which links to the latest timestamp log file.

59. Log & Run files for recipes:-

→ For each individual recipe, there is a "temp" directory under the work directory.

→ within the build system this directory is pointed to by T variable, so if you need to you can find it by using bitbake -e

$ bitbake -e <recipe-name> | grep ^T=

Each task that runs for recipe produces "log" & "run" files in ${WORKDIR}/temp:

→ you can find log files for each task in recipe's temp directory.

→ log files are named log.taskname (E.g. log.do_configure, log.do_fetch, ...)

→ for convenience; symbolic links are kept updated by Bitbake, pointing to last log files using pattern log.<task>

→ we can run scripts for every task with pattern run.<task>.<pid>

→ These files contain commands which produce build results.

60. Logging functions during task execution:-

→ Logging utilities provided by Bitbake are very useful to trace the code execution path.

→ Bitbake provides logging functions for use in Python & Shell script code as described

Python: For use within python functions, Bitbake supports log levels which are bb.fatal, bb.error, bb.warn, ...

Shell_script: For use in shell script functions, same set of log levels exists & accessed with syntax bbfatal, bberror, ...

Difference between them:-

→ Logging functions in python are directly handled by BitBake, seen on the Console & stored in execution log that Can be build/tmp/log/cooker/<machine>.

→ When logging functions are used in Shell Script, information is outputted on task's respective task log file, which is available in build/tmp/work/<arch>/<recipename>/<Software version>/temp

61. Python Logging functions:-

you can add in bb file like

```
do_compile(){
        bberror "fatal error"
        bbplain "plain Msg"
        bbdebug "Debug Msg"
        ${cc} userprog.c ${LDFLAGS} -o userprog
        bbnote "Compilation Complete"
}
```

Then build

→ $ bitbake -f -c compile myhello

Logging

bb.fatal and bbfatal: → Highest priority of logging messages as they print the message & terminate the processing. They cause the build to be interrupted.

bb.error & bberror:- they are used to display an error but do not force the build to stop.

bb.warn & bbwarn: They are used to warn users about the something.

bb.note & bbnote: Those add a note to user. They are only informative.

bb.plain & bbplain: These output a message.

bb.debug & bbdebug: These add debugging information that is shown depending on debug level used.

62. Enable debug output:-

Debug output:-

→ you can see debug o/p from BitBake by using -D option.

→ debug o/p gives more information about BitBake is doing & reason behind it.

→ Each -D option you can increase Logging level.

→ Most common usage is -DDD

$ bitbake -c cleanall myhello

$ bitbake -D myhello → Enable 1st level debug

OR

$ bitbake -c cleanall myhello

$ bitbake -DD myhello → Enable 2nd level debug

OR DebugMSG1

63. Recipe for C code span across multiple files :-

```
SRC_URI = "file://userprog.c \
           file://function.c"

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

64. Add preprocessing directive to Recipe :-

Let's say, int main() → userprog.c

```
{
    #ifdef USE_SYSCALL   * any name for macro
        write (1, "Hello System Call", 17);
    #else
        printf ("Hello world");
    #endif
    return 0;
}
```

⇒ $ gcc -DUSE_SYSCALL userprog.c -o userprog

Same logic in recipe

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

65. Recipe for Makefile :-

$ vi Makefile

In Makefile

```
CFLAGS = -g -DUSE_SYSCALL

TARGET = userprog

all : $(TARGET)

$(TARGET) : $(TARGET).c
    ${CC} $(CFLAGS) -o $(TARGET) $(LDFLAGS) $(TARGET).c

Clean:
    rm -f $(TARGET)
```

Go inside file folder,

Makefile, userprog.c, function.c files should be there.

In recipe,

```
SRC_URI = "file://userprog.c \
           file://Makefile,
```

```
do_compile() {
    make
}
```

66. oe_runmake :-

→ default behavior of do_compile task is to run oe_runmake function if a makefile (Makefile, makefile) is found. If no such file is found, do_compile task does nothing.

→ Default behavior of do_configure task is to run oe_runmake clean if Makefile is found.

oe_runmake vs make

oe_runmake function is used to run make.

oe_runmake :-

→ passes EXTRA_OEMAKE settings to make

→ displays the make command

→ checks for error generated via the call.

In OE environment you should not call make directly, rather use oe_runmake when you need to run make.

oe_runmake is one of many helper functions defined by base class

67. EXTRA_OEMAKE :-

Let's say in linux terminal, we have Makefile & simple c file.

$ make (V=1) ← this we can give as argument

↳ enabling verbose.

Like that we can add in Recipe,

↳ EXTRA_OEMAKE += "V=1"

68. Add install task to Makefile :-

In normal Linux, we have Makefile user prog.c

```
$ Cat Makefile
CFLAGS = -g -Wall -DUSE_SYSCALL
$ TARGET = userprog
all: ${TARGET}
:
:
install:
    install -d $(DESTDIR)
    install -m 0755 $(TARGET) $(DESTDIR)
Uninstall:
    rm $(DESTDIR) $(TARGET)
```

Now in terminal,

$ sudo make install DESTDIR=/usr/bin/

$ we can run `hello` (userprog) in any place.

$ make uninstall DESTDIR=/usr/bin/.

Now like in recipe:-

```
do_install () {
        oe_runmake install DESTDIR=${D}/usr/bin/
}
```

69. Makefile without clean target → [ignore clean:]

What if i have a Makefile which don't have clean target?

CLEANBROKEN = "1"

If set to "1" within a recipe, CLEANBROKEN specifies that the make clean command does not work for software being built. Consequently, OE build system will not try to run make clean during do_configure task, which is default behavior.

70. Init related.

Splitting files:-

do_package task splits the files produced by recipe during do_install into logical components.

Even sw that produces a single binary might still have

→ debug symbols
→ documentation and
Other logical components that should be split out.

→ This separation exists because not all of those installed files are useful in every image.

→ For example, you probably don't need any of the documentation installed in a production image.

→ do_package task ensures that files are split up & packaged correctly.

→ PACKAGES and FILES variables Controls splitting

PACKAGES → lists all of packages to be produced

FILES → specifies which files to include in each package by using an override to specify package.

2. PACKAGES Variable

PACKAGES → The list of packages the recipe creates.

default value : ${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-locale ${${PACKAGE_BEFORE_PN}} ${PN}

- FILES : - List of files & directories that are placed in package.

To use FILES variable, provide a package name, override that identifies resulting package.

Eg: FILES_${PN} specifies the files to go into main package. Then provide a space-separated list of files (or) paths that identify files you want includes as part of resulting package.

Eg:- FILES_${PN} += " ${bindir}/mydir1 ${bindir}/mydir2/myfile"

Good practice: use ${sysconfdir} rather than /etc (or) ${bindir} rather than /usr/bin

List of variables can be found in meta/conf/bitbake.conf

What are default values of various FILES_* variables?

Consequently you might find you do not even need to set these variables in your recipe unless SW recipe is building installs files into non standard locations.

PACKAGES & FILES_* variables in meta/conf/bitbake.conf

Configuration file define how file installed by do_install task are packaged.

③ Examples of FILES and PACKAGES variable:-

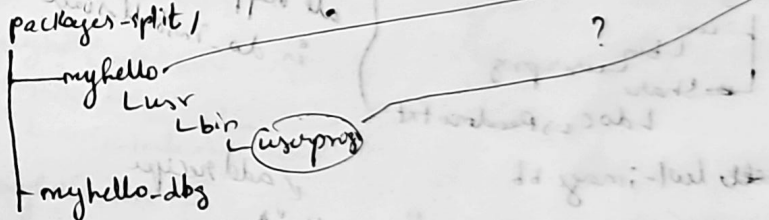
In our usrprog.c, we didn't tell usrbin (or) usrprog → go to FILE_myhello package, then how it split?.

$ Cro & check bitbake -e myhello | grep ^PACKAGES=

$ Cro & to, cd build/tmp/work/core2-64-poky-linux/myhello/0.1-r0

then in that folder, $ tree packages-split.

o/p:



then $ bitbake -e myhello | grep ^FILES_myhello=

o/p: FILES_myhello = "/usr/bin/* ....

⇒ That's why it is splitted.

④ Adding Readme.txt file to recipe:-

SRC_URI = "file://usrprog.c \
file://Readme.txt"

```
do_install() {
    install -d ${D}${bindir}
    install -m 0755 userprog ${D}${bindir}
    install -d ${D}${docdir}
    install -m 0644 Readme.txt ${D}${docdir}
}
```

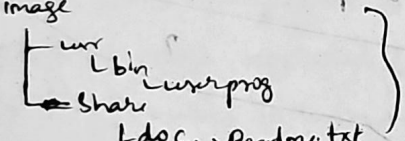
$ bitbake myhello

$ Inside build/tmp/work/core2-64-poky-linux/myhello/0.1-r0

Readme.txt file is there.

$ tree image

o/p:



all happening in do_install stage

$ vi local.conf

→ IMAGE_INSTALL:append = " myhello"

add recipe

Now check /boot.

but usr/share/doc is not there.

BCOZ, we need to add package also to IMAGE_INSTALL.

(ii) IMAGE_INSTALL:append = "myhello-doc myhello"

(iii) Now Generated image will have

doc → Readme.txt file will be there.

5. What happens when we copy Readme to includedir?

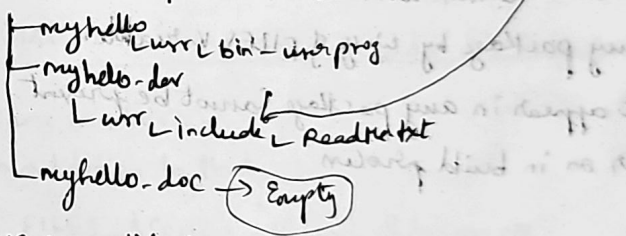
```
: do_install() {
      install -d ...
      install -d ${D}{includedir}
      install -m 0644 Readme.txt ${D}${includedir}
  }
```

Now after build, it will throw error.

"No match for myhello-doc"

Now if you go to work folder.

$ tree image packages-split/



Now, modify image recipe local.conf

IMAGE_INSTALL:append = "myhello" ← this is enough.

6. What happens if we have same binary in two packages?

→ for each installed file, first package whose FILES value matches the file is the package into which file goes.

→ If a file matches FILES variable for more than one package

in PACKAGES, it will be assigned to earliest (left most) package.

FILES_${PN}_dbg += "${bindir}/userprog"

FILES_${PN} += "${bindir}/userprog"

(i.e) myhello

(i.e) myhello-dbg

7. Installed vs shipped error:-

- Files/directories were installed but not shipped in any package [Installed-vs-shipped]

→ Files have been installed within do_install task but have not been included in any package by way of FILES variable.

→ Files that do not appear in any package cannot be present in an image later on in build process

Solution:

① → Add files to FILES for the package you want them to appear (ie. FILES_${PN} for main package).

(or)

② → Delete files at end of the do_install task if files are not needed in any package.

Eg: do_install() {

install -d ${D}${bindir}

install -m 0755 userprog ${D}${bindir}

install -d ${D}${includedir}

install -m 0644 ReadMe.txt ${D}${includedir}

}

FILES_${PN} += "${bindir}/userprog"

FILES_${PN}-dbg += "${bindir}/userprog"

PACKAGES = "${PN}-dbg ${PN}"

not added.

(i.e) example.

⑧ Create our own package & add a file into package :-

Same as above.

In addition to that,

FILES_${PN} += "${bindir}/userprog"

FILES_${PN}-dbg += "${bindir}/userprog ~~${includedir}/ReadMe.txt~~"

FILES_${PN}-lwl = "${includedir}/Re

—

9. Introduction to static library :-

→ A static library is basically a set of object files that were copied into single file with suffix .a

→ Basic tool used to create static libraries is a program called ar (archiver)

→ program can be used to
Create static libraries (also known as archive files),
modify the object files in static library,
list the names of object files in the library etc..,

In order to create static library, we have to perform two steps :-

1. Create object files from source files of the project.
2. Create static library (archive file) from object files.

10. Example of creating static libraries :-

$ mkdir 'static_lib' & 'ls'

$ arithm.c, mylib.h, print.c → files in folder

print.c :-

```
#include <stdio.h>
#include "mylib.h"
void print (char *str, int times)
{
  while (times--)
  {
    printf ("%s", str);
  }
}
```

arith.c :-

```
#include "mylib.h"
int add (int quant1, int quant2)
{
  return (quant1 + quant2)
}
int div ( )
  .
  .
int subtraction )
  .
  .
int mul ( )
  :
```

lib.h:

```
#ifndef __LIB_H
#define __LIB_H
int add (int quant1, int quant2);
int sub ( )
  .
  .
```

Step1:- Create object files

```
$ gcc -c arith.c
$ gcc -c print.c
```

Create static library

```
$ ar rcs liblwl.a arith.o print.o
```

c → Create archive if it doesn't exist

r → replace older object files in library, with new object files

s → write an object-file index into archive

This index will be used by Compiler to speed up symbol -lookup inside the library.

```
$ ls -l liblwl.a
```

o/p:- -rw-r--r-- .. liblwl.a

```
$ file liblwl.a
```

o/p:- liblwl.a : Current ar archive

```
$ mkdir temp
```

Inside temp, Create application

```
$ vi userprog.c
#include <stdio.h>
#include "mylib.h"
int main()
{
  print ("Helloworld\n", 5);
}
```

→ which is available outside of temp folder.

try to Compile,

```
$ gcc userprog.c -o userprog
```

o/p:- fatal error "mylib.h" not such file (or) directory

Reason → bcoz, it is not in usr/include (i.e) non standard location

try

```
$ gcc userprog.c -o userprog -I../
```

→ now able to find mylib.h

o/p:- undefined reference to 'print'

reason → we need to link static library

try:

```
$ gcc userprog.c -o userprog -I../ -llwl
```

→ st. library name

o/p:- Cannot find llwl

reason → bcoz, it is ~~not~~ in non-standard location.

try:

```
$ gcc userprog.c -o userprog -I../ -llwl -L..
```

Now Successfully,

```
$ ./userprog
```

o/p:- Helloworld
Helloworld
:
} 5 times.

other simple Example:

```
$ gcc -c lib_add.c -o lib_add.o
$ gcc -c lib_Sub.c -o lib_Sub.o
$ ar rcs lib_Calc.a lib_add.o lib_Sub.o
$ gcc -c main.c -o main.o
$ gcc -o main main.o -L. lib_Calc.a
```

telling linker my library is present in current dir.

```
$ ./main
```

(11) writing static library recipe

```
DESCRIPTION.
License.
SRC_URI = "file://print.c \
           file://arith.c \
           file://mylib.h"
S = "${WORKDIR}"

do_compile() {
    ${CC} -c print.c
    ${CC} -c arith.c
    ${AR} rcs liblwl.a print.o arith.o
}

do_install() {
    install -d ${D}${libdir}
    install -m 0755 liblwl.a print.o arith.o
    install -d ${D}${includedir}
    install -m 0644 mylib.h ${D}${includedir}
}
ALLOW_EMPTY_${PN} = "1"   → To solve dependency problm.
```

(12) Dynamic library:

→ Also called Shared library
→ They are linked at run time.
→ Every program can access this library at runtime & can avoid creation of multiple copies for every pgm.

```
$ ls
calc.h  lib_add.c  lib_Sub.c  main.c
$ gcc -c lib_add.c lib_Sub.c -fpic   → one of characteristic needed for dynamic/shared library
$ ls
calc.h  lib_add.c  lib_add.o  lib_Sub.c  lib_Sub.o
$ gcc *.o -shared -o lib_Calc.so
$ gcc -c main.c -o main.o
$ gcc -o main main.o -L. -l_Calc   → (or) lib_Calc.so
$ ./main
```

o/p:

./main : error while loading shared libraries : lib_calc.so : Cannot open shared object file : No such file (or) directory

Why? check dependency

$ ldd main

o/p lib_calc.so => not found

Two Solutions:

① Copy library

$ sudo cp lib_calc.so /usr/lib

$ ./main

o/p: addition

② providing path

$ pwd

$ export LD_LIBRARY_PATH=://

(13) Shared library names:-

-> Dynamic libraries follow certain naming conventions on running systems so that multiple versions can co-exist

Linux shared library can have three names. which are:

1) Linker name (eg:- libexample.so)
2) Soname (eg: libexample.so.1.2)

Real Name (eg: libexample.so.1.2.3)

Linker Name:-

-> It is the name that is requested by linker when another code is linked with your library (with -lexample linker opt)

Linker Name typically starts with

-> prefix lib
-> name of library.
-> the phrase .so

Soname:-

Every shared library has special name called the 'soname'.

Soname has

-> prefix lib
-> name of library
-> phrase '.so'.
-> followed by a period
-> version number that is incremented whenever the interface changes

Eg: libCalc.so.1

Real Name:

Real Name is the actual name of the shared library file.

Real Name = Soname + minor version number

Eg:- liblwl.so.1.0

It can also be liblwl.so.1.0.1.3

During shared library installation:-

Soname is a symbolic link to the real name.

Linkername is symbolic link to soname.

In this way both soname & linkernames ultimately point to the real name of library (i.e. actual library file)

Command to read Soname

```
$ readelf -d liblwl.so
```

(14) Dynamic library recipe:

Same,

```
do_compile(){
    ${CC} -c -fpic print.c
    ${CC} -c -fpic arith.c
    ${CC} ${LDFLAGS} -shared -Wl,-soname,liblwl.so.1
                        -o liblwl.so.1.0.*.0
}
do_install() {
    install -d ${D}${libdir}
    install -m 0755 liblwl.so.1.0 ${D}${libdir}
    ln -s liblwl.so.1.0 ${D}${libdir}/liblwl.so.1
    ln -s liblwl.so.1 ${D}${libdir}/liblwl.so
    install -d ${D}${
```

After booting,

why is liblwl.so not present in the image?

-> unversioned symbolic link is only used at development time

(15) Packaging unversioned library:-

(16) Challenge

(17) Dependencies:-

-> Most software packages have short list of other packages that they require which are called dependencies.

-> make menuConfig need ncurses library

-> These dependencies fall into two main categories

(i) build-time dependencies: required when software is built

(ii) Run-time dependencies: which are required to be installed on target in order for SW to execute.

Eg:

Build Time Dependencies:-

-> your SW uses particular library; eg: pthread / openssl

Application cannot build without pthread / openssl library

Run Time Dependencies:-

-> Eg your SW internally calls particular command for example (lspci)

App/SW can be build but need them during execution.

yocto variables:-

① DEPENDS → specifies build-time dependencies via list of bitbake recipes to build prior to build recipe.

② RDEPENDS → specifies run-time dependencies via list of packages to install prior to installing current package

⑱ DEPENDS :-

→ Within a recipe, you can specify build time dependencies using DEPENDS variable.

→ It is important that you specify all build-time dependencies explicitly.

→ When a recipe 'A' is DEPENDS on recipe 'B'. In this case, Bitbake first builds recipe 'B' & recipe 'A'.

Example:- Adding recipe1 to recipe2 as build dependency.

DEPENDS = "recipe1"

The line above means that before do_configure task of recipe2 can be run, task do_populate_sysroot from recipe1 will have completed.

do_populate_sysroot:

→ Copies a subset of files installed by do_install task into appropriate sysroot.

→ check of log of do_populate_sysroot in recipe2

→ sysroot-destdir/ : Contains a subset of files installed within do_install that have been put into shared sysroot.

⑲ Sharing files b/n Recipes:-

→ Recipes often need to use files provided by other recipes on the build host.

→ For example, an app link to common library needs access to libr. itself and its associated headers.

→ The way this access is accomplished is through sysroot.

one sysroot exists per "machine":-

→ a sysroot exists for target m/c

→ a sysroot exists for build host.

Subset of files installed into standard locations during do_install task within ${D} directory automatically goes into sysroot

do_prepare_recipe_sysroot:-

This task sets up the two sysroots in ${WORK_DIR}.

(1-a) recipe-sysroot & recipe-sysroot-native

(20) updating static lib to helloreciqe

(21) " " dyn. lib " "

(22) Introduction to RDEPENDS

→ Within a recipe, runtime dependencies can be specified using RDEPENDS variable.

→ If your recipe says T RDEPENDS on p, that tells bitbake that it must deploy p to target system if it deploys T, because T can't be used without p'.

Difference b/n DEPENDS and RDEPENDS

→ DEPENDS List of the recipe build-time dependencies.

→ RDEPENDS List of package runtime dependencies.
Must be package specific (e.g. with _${PN})
· RDEPENDS_${PN} = "package_name"

If T RDEPENDS on p, then T's do_build task is made to depend on p's do_package_write task.

(23) Examples of Recipe using RDEPENDS

```
int main()
{
    printf("Hello world");
    system("lspci");
}
```
→ System calls

In Recipe,

add,

↓

RDEPENDS_${PN} = "pciutils"

$ bitbake build recipe

→ build last-image

(24) Dependency on specific version :-

Sometimes a recipe have dependencies on specific versions of another recipe.

Bitbake allows to reflect this by using:

DEPENDS = "recipe-b (>=1.2)"

RDEPENDS_${PN} = "recipe-b (>=1.2)"

following operators are supported : =, >, <, >= and so on.

(25) Recipe..

(26) DoT..

(27) Exo exe

(28) problems with Makefile

Make allows people to generate easily based on source code.

Adv:

→ allows for faster builds

→ keeps track of source files that haven't changed since last time a build was run.

→ Skips unnecessary steps

Disadvantage:
→ Difficult to make SW pgm portable.
→ If you need to build same pgm on different platform, Makefile needs to be tweaked to work on that platform.
→ C Compiler differs from system to system.
→ Certain lib fns are missing on some systems.
→ header files may have different names.

One way to handle this to write Conditional code, with Code blocks selected by means of preprocessor directives (#ifdef)
but because of wide variety of build environments this approach quickly becomes unmanageable.
-Autotools is designed to address this problm more manageably

(29) Autotools:

GNU autotools family is a Collection of utilities designed to assist application & library developers in making their source code portable across variety of unix like systems.

It can be used both for
→ building native pgms on the build machine
→ also for Cross-Compiling to other architectures.

Autotools is frequently the brains behind the typical

$./configure
$make
$make install

It is made up of many tools,
primary tools: → automake, autoConf, make
GNU projects require a minimum set of files to be included with project: Readme, INSTALL, COPYING, THANKS, NEWS, AUTHORS.

Autotools:

Step1: Configuration (./configure)
→ Configure script is responsible for getting ready to build SW on your specific system.
→ It makes sure all of dependencies for rest of build & install process are available. Programs often written in C, usually need a C Compiler to build them.
→ In these cases, the Configure script will establish that your system does indeed have C Compiler & find out what's

Called & where to find it.

Step2 : Build (make)

→ Once configure has done its job, we can invoke make to build SW.

→ This runs a series of tasks defined in Makefile to build finished program from its source code.

→ Makefile comes with template called Makefile.in & then configure script produces a customised Makefile specific to your system.

Step3 : Install (make install)

→ make Install command will copy built program & its libraries and documentation to correct locations.

→ program's binary will be copied to directory on your PATH.

Watch lectures

CMake

L) It is cross-platform free and open source soft tool for managing build process of SW using compiler-independent method.

CMake website:

It is an extensible, open source system that manages build process in an operating system and in compiler.

CMake Vs Autotools:-

CMake is cross platform build system that can work not only with GNU make, but also Microsoft ware Visual Studio and.

$ sudo apt-get install cmake

Helloworld for cmake :-

Step1:- create helloworld c pgm (usrprog.c)

Step2: Create file CMakeLists.txt in same folder.

→ File CMakeLists.txt is the input to cMake build system.

→ It contains set of directives & instructions describing the project's source files & targets.

add_executable(usrprog usrprog.c)

Above directive/instruction will build usrprog executable from usrprog.c file.

Step3: CMake generates bunch of files in place where we run it, so its good to practice to have directory:

$ ls

CMakelists.txt usrprog.c

$ mkdir tmp

$ cd tmp

$ cmake ..

error → does not appear to contain CMakelists.txt.

so,

$ cmake ..

then do $ make

then do $ ./usrprog

Devshell - Development shell

→ Yocto build system handles all steps needed to build SW from scratch by following yocto recipes.

→ When editing package(or) debugging build failures, a development shell can be useful tool.

What is devshell?

→ devshell is terminal shell that runs in same context as Bitbake task engine.

→ In new terminal, all environment variables needed for the build are still defined, so we can use commands such as configure & make.

→ Commands execute just as if build system was executing them.

Command:

$ bitbake -c devshell <recipename>

$ bitbake -c devshell myhello

What happens when you run command?

→ starts a shell whose env is set up for development, debugging.

→ All tasks up to and including do_patch are run for target.

→ Then a new terminal is opened & you are placed in ${S}, source dir.

→ When you are finished using devshell, exit the shell (or) close terminal window.

understanding file searching paths :-

→ When a file (a patch (or) generic file) is included in SRC_URI, bitbake searches for FILESPATH and FILESEXTRAPATH variables.

In SRI_URI,

we SRC_URI = "file://usrprog.c"

↳ How when we put single file, it detects file in "files" folder?

→ If we rename "files" to myfiles (or some other) it will not work.

How?

The default setting is to look in following locations:

<recipe>-<version>
<recipe>
<files>
→ (that's why!)

} you can give as folder names

4 - Learn yocto

① Basics

What is clean build?

→ Building codebase from scratch.
→ It ensures that all parts are built fresh & no possibility of state data exists that can cause problems.

Disadv → Building from scratch takes much longer as it generally means rebuilding things that don't necessarily need to rebuilt.

Incremental build

→ Way of (i) Identifying things that do not necessarily need to be rebuilt.
(ii) Building only things which have been modified since last successful build.

Adv → It takes very less time

② Incremental build in yocto

→ yocto builds everything from source
→ when we create new build, only conf. files are created.
→ Build process then compiles everything from scratch including cross-compilation toolchain & some native tools important.
→ process can take long time.

How is incremental build supported by yocto?

Shared State Cache

→ yocto project implements a shared state cache mechanism that is used for incremental builds with aim to build only strictly necessary components for given change.
→ for this to work, build system calculates a checksum of given i/p data to task.
→ If i/p data changes, task needs to be rebuilt.
→ If all i/ps remained the same, build artifacts will be copied from sstate-cache to destination

③ Shared state cache?

Qn1: How does pieces of system have changed and what pieces have not changed detected?

Answ1: the build system detects changes in i/ps to given task by creating checksum of task's i/ps

→ If checksum changes, system assumes i/ps have changed and task needs to be rerun.

Qn 2: How are changed pieces of SW removed & replaced by sstate-cache?

Ans: → Shared State (sstate) Code tracks which tasks add which o/p to build process. This means o/p from given task can be removed, upgraded.

Qn 3: How are pre-built components that do not need to be rebuilt from scratch used they are available?

Ans: - Build system can fetch sstate objects from remote locations & install them if they are deemed to be valid.

④ Build performance

To illustrate how useful sstate-cache can be:

→ Build, no external sstate-cache can be: ≃ 1·5 hrs
time it takes to parse all of files, execute all of tasks.

→ Rebuild with no changes ≃ 10 sec
time it takes to check all hashsums & figure out there's nothing to do, parsing is already cached.

→ Removing tmp/ and rerun build ≃ 1:5 min

$cd build — one folder maintain outside of build
$ ls ../sstate-cache
$ Now delete build folder
then again source it.
$ now vi conf/local.conf
update SSTATE_DIR folder in local.conf
(1st) SSTATE_DIR ?= "${TOPDIR}/../sstate-cache"
→ build directory
$ do → $ time bitbake test-image
→ Build happens in 4 mins.

⑤ Tasks

As yocto developer, we tend to think in terms of recipe, but bitbake's fundamental unit of work is a task. Bitbake works on a "per-task" basis rather than "per-recipe" basis to determine what parts of system need to be built.

What is task?

A task is a shell (or) python script that gets executed to accomplish something, usually generating some form of o/p.

Eg: do_install, do_package.

What's benefit of per task over per-recipe approach?

Consider a case where IPK packaging backend enabled and then switching to DEB, do_install and do_package task o/p's are still valid.

With per-recipe approach, you would have to invalidate whole build & re-run it.

Rerunning everything is not the best solution.

⑥ How does shared state cache work?

Bitbake uses

a) checksums (or signatures)

b) setscene

to determine if task needs to be run (or) not.

⑦ checksum

→ It is unique signature of task's i/p's,

→ Configuration (local.conf/distro.conf)

→ recipe (.bb/.bbappend)

→ files (src_uri)

How are checksums/signature calculated?

Since a task is just a shell, taking hash of the script should tell us when it changes.

so task hash is the combination of

→ hash of the script +

→ hashes of all dependencies.

If any i/p of task change, hash will change.

⑧ setscene

Bitbake support skipping tasks if prebuilt objects are available.

What is setscene?

→ It process enables Bitbake to handle "pre-built" artifacts.

→ Signature provide an ideal way of representing whether an artifact is compatible.

→ If signature is same, an object can be reused.

⑨ How setscene works?

When Bitbake is asked to build a given target

a) Before building anything, it first asks whether cached information is available for any of targets, it's building.

b) If cached information is available, Bitbake uses this info instead of running main tasks.

Setscene Tasks (do_taskname_setscene)

It is a version of task where instead of building something bitbake can skip to end result and simply place a set of files into specific location as needed.

Not all tasks have setscene variant.

Eg: do_patch, do_unpack don't have setscene variant
do_package, do_..... has setscene variant.

Build system has knowledge of relationship b/n tasks and other preceding tasks.

Eg: If bitbake runs do_populate_sysroot_setscene for something,
→ it does not make sense to run any of do_fetch, do_patch, do_configure ...
→ If do_package needs to be run, Bitbake needs to run those other tasks.

(10) How Setscene works?

→ Bitbake runs in two phases.
a) setscene stage
b) actual build

1) Bitbake first calls function defined by BB_HASHCHECK_FUNCTION variable with a list of tasks & corresponding hashes it want to build.
$ bitbake -e | grep BB_HASHCHECK

2) This fn is designed to fast & returns list of tasks for which believes it can obtain artifacts.

3) Next for each of tasks that were returned as possibilities, Bitbake executes a setscene version of task that possible artifact covers.

4) Setscene version of task executes & provides necessary artifacts returning either success (or) failure

5) After all setscene tasks have executed, Bitbake calls the remaining tasks which have no artifacts.

(11) Stamps

Bitbake uses checksums (or signatures) along with the setscene to determine if task needs to be run.
Signatures are present in STAMPS_DIR directory.
$ bitbake -e | grep ^STAMPS_DIR=
This directory holds information that Bitbake uses for accounting → what tasks have run,
when they have run.

Stamp files

→ For each task that completes successfully, Bitbake writes a stamp file into STAMPS_DIR directory.

How does bitbake determine if task is need to be rerun?

Bitbake checks if stamp file with matching i/p checksum exists for task.

→ If such stamp file exists, task's output is assumed to exist & still be valid.

→ If file does not exist, task is rerun.

$ cd build/tmp/STAMPS ...

(12), (13),

(14) Bitbake clean tasks

① $ bitbake -c clean <recipename>

↳ Removes all o/p files for target from do_unpack task onwards (i.e. do_unpack, do_configure, do_compile, do_install, do_package)

Running this task does not remove sstate cache files.

② $ bitbake -c cleanstate <recipename>

↳ Removes all o/p files & shared state cache for target.

③ bitbake -c cleanall <recipename>

Removes all o/p files & shared state cache & downloaded source files for target.

(15) PROVIDERS

→ How does bitbake myhello picks up myhello_0.1.bb recipe?

Ans PROVIDERS

What happens when you say bitbake <target>?

1. Bitbake parses all recipes.
2. Bitbake looks through PROVIDES list for each of recipe.
3. When target matches PROVIDES list, it will build that recipe.

What is PROVIDES list?

A PROVIDES list is the list of names by which recipe can be known.

By dfault, PN is added automatically into PROVIDES list.

$ bitbake -e myhello | grep ^PROVIDES=

What's benefit of recipe adding names to PROVIDES list?

When a recipe uses PROVIDES, recipe functionality can be found under an alternative name (or) names other than implicit PN name.

Eg: Suppose a recipe named keyboard_1.0.bb contained

The following:
PROVIDES += "fullkeyboard"
PROVIDES list for this recipe becomes "keyboard" which is implicit and "fullkeyboard" which is explicit.

(16) Example of providers :-
Let's say recipe name "myhello-0.1.bb".
So, usually we can do $ bitbake myhello -> It will build.
Let's ~~say~~ try, $ bitbake hello1 -> It throws error "Nothing PROVIDES hello1".
Now inside "myhello-0.1.bb" recipe,
add, PROVIDES += "hwl"
$ bitbake -c cleansstate myhello
$ bitbake hwl -> build -> myhello-0.1.bb recipie only.

(17) Introduction to preferences
What happens when there are two recipes with same name and you say bitbake <target>?
Eg: myhello_0.1.bb myhello_0.2.bb
PREFERRED_VERSION -> this variable determines which recipe should be given preference.
Eg:- PREFERRED_VERSION_(myhello) = "myhello_0.1.bb"
(myhello -> target image)
So, while build "myhello", "myhello_0.1.bb" is used.

(18) Compatibility :-
$ bitbake -e myhello | grep ^COMPATIBLE_MACHINE =
We can use variable to stop recipes from being built for machines with which recipes are not compatible.
Eg: COMPATIBLE_MACHINE = "qemux86 | "
Try :- modify myhello recipe to support only qemuarm
COMPATIBLE_MACHINE = "qemuarm" -> add in recipies.

(19) Yocto Kernel development (E)(K) [Kindly focus on yocto kernel upcoming session]
When you are doing development with kernel, you need the following:
-> Kernel source, Kernel config (.config), patches (if any)
-> Each y.p release has set of yocto linux kernel recipes (linux-yocto).
-> Each release of y.p every six months, includes two (or) three versions of Linux kernel with broad range of HW support.
Eg: y.p -> 3.0 -> LTS kernel version: linux-yocto-4.19; released kernel version: linux-yocto-5.2
$ vi source/poky/meta/recipes-kernel/linux
-> linux-yocto-5.2.bb linux-yocto-4.19.bb . . .
Let's boot image & check
$ uname -r
o/p: ~~linux~~ 5.2.32-yocto-standard

What is command to check kernel version?

Try

Find out recipe of kernel?

$ bitbake -e lwl-image | grep PREFERRED_PROVIDER_virtual/kernel

To find which version?

$ bitbake -e lwl-image | grep PREFERRED_VERSION_linux-yocto

o/p 5.2

Change preferred version

$ vi local.conf

PREFERRED_VERSION_linux-yocto = "4.19"

(20) Linux kernel recipes in yocto

→ Two ways of Compiling kernel in yocto,

→ (1) By using linux-yocto recipe provided in poky

→ (2) By using a fully Custom kernel recipe

How do we know which recipe is used?

Machine file specifies which kernel is used using

PREFERRED_PROVIDER_virtual/kernel

How do we know which version of recipe is used?

PREFERRED_VERSION_<kernel_provider>

you can check some machine conf files

Eg: vi poky/meta/conf/machine/qemux86.conf

(21) creating a new kernel recipe [all in meta-mylayer]

$ mkdir recipes-kernel → $ cd recipes-kernel

$ mkdir linux → lwl-linux_0.1.bb

Copy any recipe & paste here & edit

DESCRIPTION = "LWL kernel recipe"

LICENSE = "MIT"

LIC_FILES_CHKSUM = "file://{}----"

inherit kernel

SRC_URI = " https://cdn-----;name=kernel \

file://defconfig"

S = "${WORKDIR}/linux-5-7"

SRC_URI[kernel.md5sum] = " "

SRC_URI[kernel.sha--] = " "

(1) SRC_URI → kernel link. Go to kernel.org. click latest kernel tar file link & paste it.

(2) once try to build, it will throw error as need to add md5sum & sha---. then copy md5sum & sha- from the error which is thrown in terminal & add in recipe file.

(3) defconfig → kernel config file download from link. and put in files folder.

Then build it. $ bitbake lwl-linux → build kernel for 5-7-0

(22) WORKDIR of Kernel recipe

$ bitbake -e kwl-linux | grep ^WORKDIR=

o/p ---- ↖
Then $cd to).

$ cd linux-5-7
$ cd image & $ ls
o/p boot etc lib
$ cd image/boot, $ cd image/etc/ , $ cd image/lib
build kwl-image & check Kernel version in booting. (uname -r)

(23) Kernel customizations:

We can enable/disable some options.

Eg:- CONFIG_LOCALVERSION → o/p: linux-5-7-0
instead of "linux-5-7-0-kwl"

↳ This allows you to append an extra string to end of your Kernel version.
→ This will show up when you enter uname command.

Q $ bitbake -c menuconfig virtual/kernel

Note:- This requires an existing configurations to start the process.

To get an initial configuration, execute
$ bitbake -c kernel_configme virtual/kernel"

This modifies .config in working source directory
→ Make sure to capture these changes as subsequent changes will overwrite your changes.

After executing menuconfig, which modifies .config directly workdir, you could use '-c' variant for compile step.
$ bitbake -c compile virtual/kernel.

Now,
run, → $ bitbake -c menuconfig virtual/kernel
It will open GUI, select "Local version - ..."
& type "-kwl".
Then build kwl-image.
Now check in booting. uname -r → linux-5.7.0-kwl

Generate new defconfig
save defConfig
$ bitbake -c savedefConfig virtual/kernel -vDDD
↳ When invoked by user, creates a defConfig file that can be used instead of default defConfig.
It is in location, build/tmp/work/genericx86-64-poky-linux/kwl-linux/0.1-r0/build
Then go & check vi defConfig & search
"CONFIG_LOCALVERSION" = "-kwl"

(24) Kernel recipe for git repository. (X)

(25) Patching Kernel (X)

Static library:

```
$ add.c sub.c fun.h main.c
$ gcc -c add.c sub.c
$ ar rcs new.a add.o sub.o
$ gcc -I. -c main.c
$ gcc -o main main.o new.a
```

Dynamic library

```
$ add.c sub.c fun.h main.c
$ gcc -fPIC -c add.c sub.c
$ gcc *.o -shared -o lib new.so
$ export LD_LIBRARY_PATH=:// $(pwd); $ ldconfig
$ gcc -c main.c
$ gcc -o main main.o lib new.so
```