System Setup

Setup requirements

- Beagle Bone Black Kit (BBB+USB Cable)
- USB2TTL Cable
- 4GB or more uSD
- uSD Reader
- Linux Machine (Preferred) or VM with Ubuntu 18.04 or higher

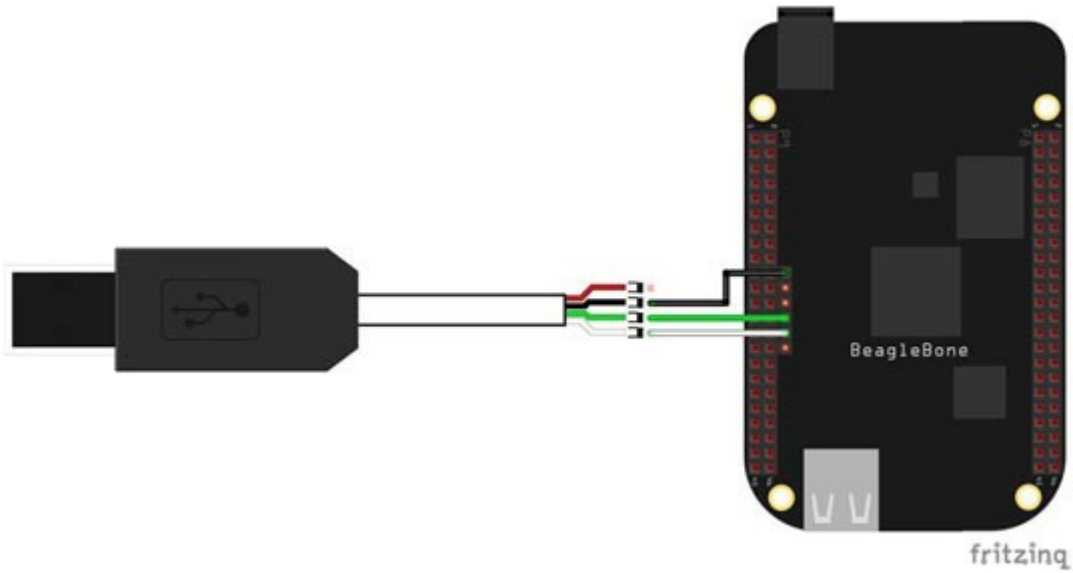
Package Installation

- Install the packages with following command:
  $ sudo apt install sed make binutils gcc g++ bash patch gzip bzip2 perl tar cpio python unzip rsync wget libncurses-dev git minicom libssl-dev flex bison

Default Bootup Setup (One time)

Connect USB2TTL cable

- Connect USB2TTL cable to BBB and system as per below steps. Do not power up BBB
- Connect the USB side of the TTL cable to your computer
- Connect the wires to the J1 header on your BeagleBone Black as shown:
  - Black wire to pin 1
  - Green wire to pin 4
  - White wire to pin 5



Getting the serial console messages

On Linux system (install the minicom package if not already there):

- sudo minicom -s and do the setup for baud 115200 bits 8n1 hw & sw flow control off

- sudo minicom -o
- Power on BBB to boot into Linux
- Login into BBB as root
- uname -r # Verify your original kernel version
    - df -h /boot/uboot | tail -1 | awk '{print $6}' # Determine / or /boot/uboot
    - poweroff

PS If you don't see any boot up messages or login prompt on the serial console, try swapping the green & white wire

Buildroot Exercises

Customizing the build-root for Beaglebone Black

Follow the below steps:

1) Create the required directories
   $ cd
   $ mkdir Exercises
   $ git clone https://github.com/buildroot/buildroot
   $ cd buildroot
   $ git checkout -b br-2-2025 2025.02.6

   OR

   If both of above doesn't work, download the tar ball from https://buildroot.org/downloads/buildroot-2025.02.6.tar.gz and untar with

   $ tar -xvf buildroot-2025.02.6.tar.gz
   $ cd buildroot

2) Configure the buildroot by using 'menuconfig' utility
   $ make menuconfig

3) Select Target options menu. Under that
   - Select "Target architecture" as ARM (little endian)
   - Select "Target Architecture Variant" as cortex-A8
   - Select "EABIHF" as Target ABI

4) Select Toolchain Menu
   - Select Toolchain Type as External Toolchain
   - Select Toolchain as ARM14.2 rel1

5) Select the System Configuration Menu. Under that
   Put some custom values for System hostname, System banner and Root password
6) Select the Kernel Menu
   - Select Custom version as "Kernel Version" and enter 6.12.54 in "Kernel Version" text field
   - For kernel configuration, enter 'omap2plus' in the 'defconfig name' option
   - Select 'Build a Device Tree Blob' option and type
   - ti/omap/am335x-boneblack as the 'In-tree Device Tree Source file names'
   - Select "Needs host OpenSSL"

-

7) Leave Target Packages and Filesystem Images as is.
8) Select “Bootloaders” menu and under that
   - Select ‘U-Boot’
   - Select Kconfig as “Build system”
   - Select ‘Custom Version’ for ‘U-Boot Version’ and below that select 2024.04 as custom version
   - Use ‘am335x_evm’ as ‘Board defconfig’
   - Select u-boot.img as the ‘U-Boot binary format’
   - Select ‘Install U-Boot SPL binary image’ and Use ‘MLO’ as ‘U-Boot SPL/TPL binary image name(s)’
   - DEVICE_TREE=am335x-boneblack as “Custom make options”

9) Exit & save the configuration
10) Build the images using
    $ make 2>&1 | tee build.log

Preparing the SD Card

1. Connect the SD Card to the host machine and figure out the corresponding device file (usually /dev/sdb or /dev/mmcblk0). Use dmesg to figure out the device file for the SD card.
2. Unmount all partitions of your SD card (they are generally automatically mounted by Ubuntu).
   $ umount /dev/sdb* (Considering the sd card device file is /dev/sdb)
3. Erase the beginning of the SD card to ensure that the existing partitions are not going to be mistakenly detected:
   $ sudo dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=16

   In general, if you use the internal SD card reader of a laptop, it will be mmcblk0, while if you use an external USB SD card reader, it will be sdX (i.e. sdb, sdc, etc.). Be careful /dev/sda and /dev/sdb are typically the harddisks as well

4. SD Card needs to be split in 2 partitions:
   - First Partition of type FAT32 would hold Bootloaders (MLO and u-boot.img), kernel image (zImage) and Device Tree (am335x-boneblack.dtb)
   - Second partition for the root filesystem

5. Start the cfdisk tool for that:
   $ sudo cfdisk <device file for SD card (eg. /dev/mmcblk0)
6. Choose the dos partition table
7. Create the first partition of size 256MB, primary, with type e (W95 FAT16) and mark it bootable
8. Create a second partition, also primary, of size 2GB, with type 83 (Linux)
9. Create a third partition, also primary, with the rest of the available space, with type 83 (Linux)
10. Write the partition table
11. Exit cfdisk
12. Format the first partition
    $ sudo mkfs.vfat -F 32 -n boot /dev/mmcblk0p1 **(Would be /dev/sdb1 if device file is /dev/sdb)**
13. Format the Second partition

$ sudo mkfs.ext4 -L FirstRootfs /dev/mmcblk0p2
14. Format the Third partition
    $ sudo mkfs.ext4 -L SecondRootfs /dev/mmcblk0p3
15. Remove the SD card and insert it again, the two partitions should be mounted automatically, in /media/$USER/boot and /media/$USER/FirstRootfs
16. Copy MLO, u-boot.img, am335x-boneblack.dtb and zImage to the boot partition of the SD card. The images can be found under <buildroot>/output/images directory
17. Extract the rootfs.tar file to the Second partition
    $ cd <Path to Buildroot>/output
    $ sudo tar -C /media/$USER/FirstRootfs/ -xf images/rootfs.tar (Need to make sure where the mountpoint is)
18. Next create the directory by name extlinux in the boot (first) partition and copy the extlinux.conf under it (extlinux.conf could be found under git repo or shared point. Typically under Images directory)
19. Unmount the paritions, remove the SD Card, put it into the card and power up the board.
20. If board doesn't boot and shows 'CCCC', then raw dump the bootloaders using the following steps.
21. umount the SD Card using 'umount /dev/sdb*'
22. Next, dummp MLO using disk dump utility as below:
    $ cd <path to buildroot>/output/images
    $ sudo dd if=MLO of=<device file for sd card> seek=256 bs=512 conv=notrunc
    $ sudo dd if=u-boot.img of=<device file for sd card> seek=768 bs=512 conv=notrunc

RootFs Related

Adding an init script to the rootfs

The idea over here is to add the script which automatically gets executed during the boot up. The script would set up the network over usb

1. Get into the buidroot directory
   $ cd Exercises/buildroot
   Create the directory with name rootfs-overlay
   $ mkdir -p board/test/bbb/rootfs-overlay/
2. Since the file needs to be created under /etc/init.d, let's create the similar directory structure and add the script
   $ mkdir -p board/test/bbb/rootfs-overlay/etc/init.d/
   $ cp <path to repo>/Images/S30usbgadget board/test/bbb/rootfs-overlay/etc/init.d/ (S30usbgadget is available in repo (under Scripts) or sharepoint provided during the training)
3. Add executable permission for S30usbgadget
   $ chmod a+x board/test/bbb/rootfs-overlay/etc/init.d/S30usbgadget
4. Configure the path for rootfs-overlay in buildroot
   $ cd board/test/bbb/rootfs-overlay
   $ pwd (copy the path)
   $ cd -
   $ make menuconfig
   Locate Root filesystem overlay directories (ROOTFS_OVERLAY) under System configuration and paste the path for the overlay directory

5. Next thing is to add the script for configuring the network

$ mkdir -p board/test/bbb/rootfs-overlay/etc/network/
Copy the interfaces script (From the Scripts folder of the repo or sharepoint)
$ cp <path to interfaces> board/test/bbb/rootfs-overlay/etc/network/

6. $ make
7. Plug the SD Card into Laptop
8. Untar the rootfs.tar to the SD card
   $ cd <path to buildroot>
   $ sudo rm -rf /media/$USER/FirstRootfs/* (Make sure that its mounted)
   $ sudo tar -C /media/$USER/FirstRootfs/ -xvf output/images/rootfs.tar
   (Rootfs for SD card would be under /media/$USER/FirstRootfs/)
9. Unmount the SD card, plug it into the board and boot up the board
10. Configure the host network address using:
    nmcli con add type ethernet ifname enxf8dc7a000001 ip4 192.168.7.3/24
    (To be executed On PC/Laptop)
11. Verify if you are able to ping the system using 'ping 192.168.7.2'

Adding dropbear as an ssh server

The idea over here is to add the ssh and scp capability into the Root Filesystem:

1. Get into the buidroot directory
   $ cd Exercises/buildroot
1. $make menuconfig
2. Search for dropbear by pressing '/' and then enter DROPBEAR
   It will give you a list of results, and each result is
   associated with a number between parenthesis, like (1). Then simply press 1, and menuconfig will jump to the right option.
3. Select the dropbear, exit the menuconfig and initiate the build
   $ make
4. Insert the SD card in PC & update the rootfs in SD Card
   $ cd <path to buildroot>/
   $ sudo rm -rf /media/$USER/FirstRootfs/* (Make sure that the FirstRootfs is mounted)
   $ sudo tar -C /media/$USER/FirstRootfs/ -xvf output/images/rootfs.tar
5. Unmount the SD card partitions and test it on the board
6. Try to connect with the board using
   $ ssh root@192.168.7.2
   if prompted for password, set the password at beaglebone black with 'passwd'. Try to set the small password to avoid typing it all the time

U-Boot

Adding the command in u-boot

The idea over here is to add the custom command in the uboot

1. Get into the u-boot directory
   $ cd <path to buildroot>/output/build/uboot-2024.04
2. Make a copy of already existing command say led.c under cmd
   $ cd cmd
   $ cp led.c myprint.c
3. Modify the myprint.c as follows:
   - Remove everything except do_led and U_BOOT_CMD
   - Change the name of the function do_led to do_myprint
   - Deleting everything in do_led, except the 'return 0'
   - Include the 'printf' statement above 'return 0'
4. Modify the U_BOOT_CMD as below:

```
U_BOOT_CMD(
        myprint, 1, 1, do_myprint,
        "My printf",
        "My Test command"
);
```

5. Next thing is to modify the Kconfig to give out the menu option for our command. For this, modify the Kconfig to add the menu option as below:
   ```
   config CMD_MYPRINT
        bool "MY Test Print"
        help
          Enable the Test printf
   ```
6. Next thing is to modify the Makefile to add the below line:
   obj-$(CONFIG_CMD_MYPRINT) += myprint.o
7. Get into the buildroot directory and execute the following
   $ make uboot-menuconfig
   Search for MYPRINT and select this option
8. Rebuild the uboot
   $ make uboot-rebuild
9. Check if myprint.o is generated under buildroot/output/build/uboot-2021.04/cmd/
10. Update the u-boot in the SD Card's first partition (applicable only if MLO & u-boot.img are not raw-dumped. If its raw-dumped, skip to the next step)
    This can be done in 2 ways.
    Using the scp command:
    $ mount /dev/mmcblk1p0 /mnt (on the board)
    $ scp buildroot/output/images/u-boot.img root@192.168.7.2:/mnt/ (On host)
    $ sudo umount /mnt (on board)
    $ reboot & stop at the uboot prompt by pressing 'Enter' during boot up
    Another way is to Transfer by inserting the SD card in PC
    $ cp buildroot/output/images/u-boot.img /media/$USER/boot/
    $ Unmount the SD card
11. if u-boot is raw-dumped, then follow the below steps.
    This can be done in 2 ways.
    (i) Using the scp command:
    $ scp buildroot/output/images/u-boot.img root@192.168.7.2:(On host)
    $ dd if=u-boot.img of=<device file for sd card> seek=768 bs=512 conv=notrunc
    $ reboot & stop at the uboot prompt by pressing 'Enter' during boot up
    Or (ii) Transfer by inserting the SD card in PC:
    $ sudo dd if=u-boot.img of=<device file for sd card> seek=768 bs=512 conv=notrunc
    $ Unmount the SD card and plug it into the board
12. While booting up, press 'Enter' or 'Space' key to get the uboot prompt
13. Test if myprint command is available in the prompt & execute the same

Creating & Using the uboot patch

The idea over here is to make the uboot changes persistent across the build. This can be achieved by creating the patches and saving it in appropriate board directory

1. Get into the uboot build directory
   $ cd <path to buildroot>/output/build/u-boot-2024.04
2. Temporarily mv the changes to some other place and Intialize the git for uboot and commit the initial changes

$ git init
$ git add .
$ git commit (Add the commit message & exit)

3. Copy back the relevant files (myprint.c, Makefile and Kconfi) to respective directory (i.e at uboot-2024.04/cmd)
   $ git add .
   $ git commit (Add the comment & exit)
   $ git add cmd/Kconfig
   $ git commit (Add the comment & exit)
4. Create the patches
   $ git format-patch -2
   This would create 2 patches
5. Next step is to create the patches directory in buildroot and copy the patches
   $ cd buildroot
   $ mkdir board/test/patches/uboot/
   $ cp ../u-boot-2024.04/*.patch board/test/patches/uboot/
6. Next, update patches path in menuconfig
   $ make menuconfig
   Navigate to Bootloaders and update the path in 'Custom U-boot Patches' to point to above patch directory
7. Next step is to create the defconfig for uboot
   $ make uboot-menuconfig
   Search for MYPRINT, select the same & exit the menuconfig
8. Generate the defconfig for these changes
   $ make uboot-savedefconfig
   This would generate the defconfig at u-boot build directory. Verify if CONFIG_MYPRINT is set. Copy the above configuration file at board/test/bbb/uboot.config
9. Run the menuconfig for the buildroot
   $ make menuconfig
   In uboot, instead of using a defconfig, chose using a custom config file
   In the configuration file path, enter the path board/test/bbb/uboot.config
   Exit the menuconfig
10. For testing, let's first dir-clean the uboot and perform menuconfig again
    $ make uboot-dirclean
    $ make uboot-menuconfig
    Check if MYPRINT option is already set and also the patch shoulod been applied as well
11. $ make uboot-rebuild
    Make sure that myprint.o is generated. This means the patch is applied correctly and the uboot is compiled with the same

Kernel

Using custom defconfig for the kernel

The idea is to modify the kernel configuration as per the requirement, test it and make it permanent

1 Start the Kernel menuconfig tool
  $ make linux-menuconfig
  Update the Local Version option in General Setup Menu
2 Exit the menuconfig and this would generate the updated .config file under output/build/linux-<version>

This is the temporary change and would be deleted at the next make clean. So, the idea is to make this change persistent

3. Run buildroot menuconfig
   $ make menuconfig
   In the Kernel menu, instead of Using a defconfig, chose Using a custom config file. This will allow us to use our own custom kernel configuration file, instead of a pre-defined defconfig that comes with the kernel sources.
   In the Configuration file path, enter board/test/bbb/linux.config
4. Exit menuconfig
5. Run ‘make linux-update-defconfig’
   This will generate the configuration file in board/test/bbb/linux.config. It will be a minimal configuration file.
6. Now, you may restart the build of the kernel

Code Changes & Patching the Kernel

The Idea over here is to modify the kernel code, test the changes, create the patches and apply the same

1. Get into <buildroot>/output/build/linux-<version>/ and modify drivers/char/misc.c by adding the printk statement
2. Rebuild the kernel
   make linux-rebuild
   This would generate the updated image. Test the same
3. Next step is to generate the patch for the above changes and place it under the directory – board/test/bbb/patches/linux
4. The buildroot needs to be configured to apply these patches before building the kernel. To do so, run menuconfig, go the to the Build options menu, and adjust the Global patch directories option to board/test/bbb/patches/
   $ make linux-menuconfig
5. Clean up the linux package completely so that its sources will be re-extracted and our patches get applied the next time
6. $ make linux-dirclean
   In output/build/, the linux-<version> directory will have disappeared
7. Next, run ‘make linux-menuconfig’
   This will Extract the Linux kernel sources, apply the patch, load the kenrel config and starts the kernel menuconfig tool.
8. Build the kernel with linux-rebuild

Yocto Exercises

Bitbake Hands-On

Bitbake setup to build the simple recipe

The objective of this exercise is to familiarize ourselves with bitbake and understand the basic minimal setup required to build the simple recipe

1. Set the locale
   $ sudo locale-gen en_US.UTF-8

2 Clone the bitbake and verify the same
$ mkdir ${HOME}/Test
$ cd ${HOME}/Test
$ git clone https://github.com/openembedded/bitbake
$ cd bitbake/bin
$ ./bitbake --version

If there is an error such as 'locale.Error: unsupported locale setting', then export the following:
$ export LC_ALL="en_US.UTF-8"

3 Next, update the path with following
$ export PATH=${HOME}/Test/bitbake/bin/:$PATH

4 Next, create the build directory. Make sure to create it outside the bitbake repo
$ cd ../../Test/
$ mkdir build

5 Now, execute 'bitbake' command and observe the errors. To fix this, search the corresponding variable in 'Variable Glossary' section of bitbake documentation and update it accordingly. Bitbake documentation could be found at https://docs.yoctoproject.org/ under ‘Manuals’ Sections

6 Execute the 'bitbake' again & observe the error. Fix it by creating corresponding file at the expected path

7 Execute the 'bitbake' and observe the error. Fix it by copying the corresponding thing from bitbake repo cloned earlier

8 Now the bitbake should print the following:
Nothing to do. Use 'bitbake world' to build everything, or run 'bitbake --help' for usage information.

9 Next, execute 'bitbake world' and it should give out an error message:
ERROR: no recipe files to build, check your BBPATH and BBFILES?

10 To fix this, create the recipe file by name hello.bb in build directory with following contents:

```
DESCRIPTION = "Test Recipe"

python do_build() {
  print("Hello World")
}
```

11 Execute the 'bitbake world' and fix environment variable issues by understanding the error messages.

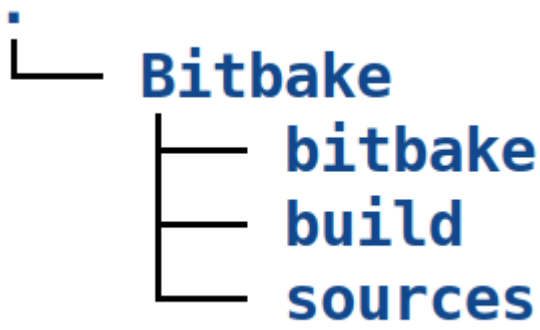
12 Here is the final output

```
pradeep@pradeep-lp3:~/Downloads/Bitbake/build$ bitbake world
Loading cache: 100% |################################################################################################| Time: 0:00:00
Loaded 1 entries from dependency cache.
WARNING: /home/pradeep/Downloads/Bitbake/build/hello.bb: python should use 4 spaces indentation, but found tabs in hello.bb, line 4 --:--:--
Parsing recipes: 100% |##############################################################################################| Time: 0:00:00
Parsing of 1 .bb files complete (0 cached, 1 parsed). 1 targets, 0 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |###########################################################################################| Time: 0:00:00
NOTE: No setscene tasks
NOTE: Executing Tasks
```

Restructuring the Directory Structure & Introducing Layer

The objective here is to have a organized metadata by using the layers

1. Create the directory by name 'Bitbake' and under that create directories by name 'build' and sources. Next, clone the bitbake repo under 'Bitbake'. The directory structure should look as below:



2. Next, create the directory by name 'classes' under build and copy base.bblclass from bitbake repo and create the directory by name conf and copy bitbake.conf from bitbake repo under conf. Further, resolve all the bitbake related issues as per the earlier assignments. Recipes would be created in the next step. Make sure to update the BBPATH to point to new build directory
3. Next step is to create the directory by name 'meta' under sources. The objective is to have this directory listed as layer.
   For this, bitbake needs the bblayers.conf file. Create the file by name 'bblayers.conf' under build/conf with following content:
   BBLAYERS ?= "<path to meta layer>"
4. Next step is to add the layer.conf file under meta/conf with following contents:
   BBPATH .= ":${LAYERDIR}"
   BBFILE_COLLECTIONS += "meta"
   BBFILE_PATTERN_meta := "^${LAYERDIR_RE}"
   Next execute 'bitbake-layers show-layers' and the output should be as below:

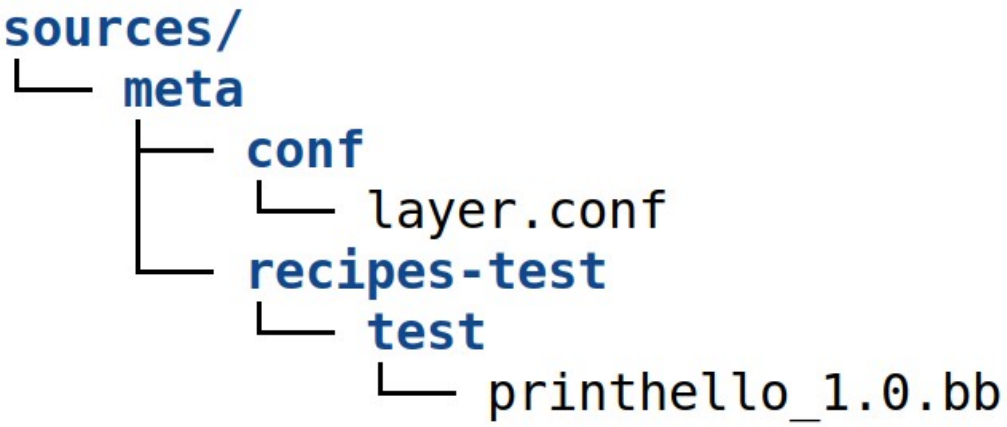
```
pradeep@pradeep-lp3:~/Trainings/Yocto/Bitbake/build$ bitbake-layers show-layers
NOTE: Starting bitbake server...
WARNING: Layer meta should set LAYERSERIES_COMPAT_meta in its conf/layer.conf file to list the core layer
layer                path                                                                priority
==========================================================================================================
meta                 /home/pradeep/Trainings/Yocto/Bitbake/sources/meta                  1
```

5 Next step is to add the recipe. For this, create the directories recipes-test/test under meta and under test create the file by name printhello_1.0.bb with following contents:

```
DESCRIPTION = "Test Recipe"

python do_build() {
        bb.plain("Hello World")
}
```

The Directory structure looks as below:



6 Expected output is

```
pradeep@pradeep-lp3:/media/pradeep/Data3/Trainings/Yocto/Bitbake/build$ bitbake printhello
WARNING: Layer meta should set LAYERSERIES_COMPAT_meta in its conf/layer.conf file to list the c
ore layer names it is compatible with.
Loading cache: 100% |#########################################################| Time: 0:00:00
Loaded 1 entries from dependency cache.
WARNING: /media/pradeep/Data3/Trainings/Yocto/Bitbake/sources/meta/recipes-test/test/printhello_
1.0.bb: python should use 4 spaces indentation, but found tabs in printhello_1.0.bb, line 3
Parsing recipes: 100% |#######################################################| Time: 0:00:00
Parsing of 1 .bb files complete (0 cached, 1 parsed). 1 targets, 0 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |####################################################| Time: 0:00:00
NOTE: No setscene tasks
NOTE: Executing Tasks
WARNING: python should use 4 spaces indentation, but found tabs in printhello_1.0.bb, line 3
Hello World
NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded.

Summary: There were 3 WARNING messages.
```

Additional Layer

The objective here is to use the classes for reusability

1. Continuing with the previous setup, create the layer with name meta-example under sources. Further to this, create the recipe by name first_0.1.bb under meta-example/recipes-example/first/. Create the recipe - first_0.1.bb with following contents:
   Description = "First Recipe"
   do_build () {
   	echo "first: Running the build shell script"
   }

2. Next execute the command 'bitbake-layers show-layers' and this should list both the layers as below:

```
pradeep@pradeep-lp3:~/Trainings/Yocto/Bitbake/build$ b
NOTE: Starting bitbake server...
WARNING: Layer meta-example should set LAYERSERIES_COM
ble with.
WARNING: Layer meta should set LAYERSERIES_COMPAT_meta
layer                 path
==========================================================
meta-example          /home/pradeep/Trainings/Yocto/Bi
meta                  /home/pradeep/Trainings/Yocto/Bi
```

3. ‘bitbake -s’ should list 2 recipes

```
pradeep@pradeep-lp3:/media/pradeep/Data3/Trainings/Yocto/Bitbake/build$ bitbake -s
WARNING: Layer meta should set LAYERSERIES_COMPAT_meta in its conf/layer.conf file to list the c
ore layer names it is compatible with.
WARNING: Layer meta-example should set LAYERSERIES_COMPAT_meta-example in its conf/layer.conf fi
le to list the core layer names it is compatible with.
Loading cache: 100% |                                                              | ETA:  --:--:--
Loaded 0 entries from dependency cache.
WARNING: /media/pradeep/Data3/Trainings/Yocto/Bitbake/sources/meta/recipes-test/test/printhello_
1.0.bb: python should use 4 spaces indentation, but found tabs in printhello_1.0.bb, line 3
Parsing recipes: 100% |##########################################################| Time: 0:00:00
Parsing of 2 .bb files complete (0 cached, 2 parsed). 2 targets, 0 skipped, 0 masked, 0 errors.
Recipe Name                                   Latest Version         Preferred Version
 Required Version
===========                                   ==============         =================
 ================

first                                                :1.0-r0

printhello                                           :1.0-r0
```

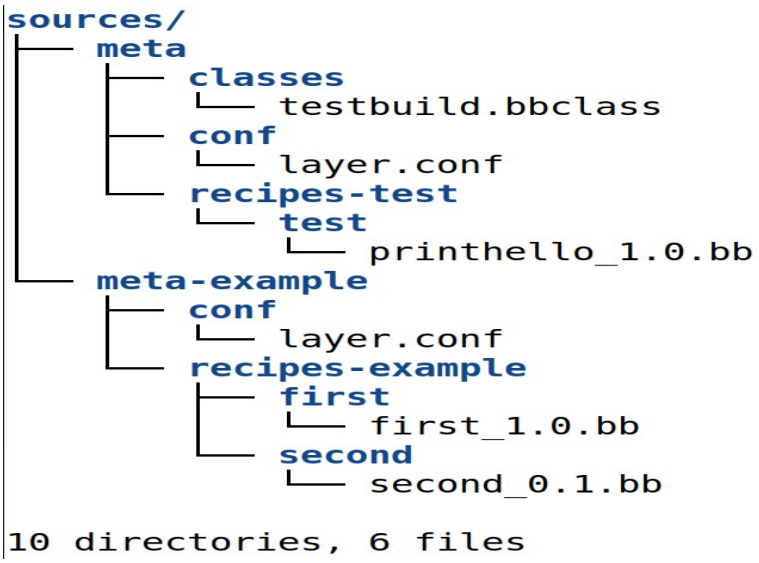
4. Next, try bitbake first and this should execute the recipe and corresponding logs should be available under tmp/work/first-1.0-r0/temp/log.do_build

Using Classes

The objective here is to use the classes for reusability

1. Create the directory by name classes under meta and create the class file by name testbuild.bbclass with following contents:
   do_build() {
   echo "Executing testbuild_do_build."
   }
   Promote this function as task

2. Further, create the folder by name second under recipes-example and add the recipe by name second_0.1.bb under that. The directory structure under sources should look as below:



3. Following should be the content for second_0.1.bb
   (i) Should inherit the testbuild class
   (ii) Should define the bitbake style python function name do_testpatch
   (iii) The do_testpatch should be promoted as task and added as dependency for build
4. ‘bitbake -s’ should output the following:

```
pradeep@pradeep-lp3:/media/pradeep/Data3/Trainings/Yocto/Bitbake/build$ bitbake -s
WARNING: Layer meta should set LAYERSERIES_COMPAT_meta in its conf/layer.conf file to list the core layer names it
 is compatible with.
WARNING: Layer meta-example should set LAYERSERIES_COMPAT_meta-example in its conf/layer.conf file to list the cor
e layer names it is compatible with.
Loading cache: 100% |############################################################| Time: 0:00:00
Loaded 1 entries from dependency cache.
WARNING: /media/pradeep/Data3/Trainings/Yocto/Bitbake/sources/meta-example/recipes-example/second/second_0.1.bb: p
ython should use 4 spaces indentation, but found tabs in second_0.1.bb, line 6
WARNING: /media/pradeep/Data3/Trainings/Yocto/Bitbake/sources/meta/recipes-test/test/printhello_1.0.bb: python sho
uld use 4 spaces indentation, but found tabs in printhello_1.0.bb, line 3
Parsing recipes: 100% |##########################################################| Time: 0:00:00
Parsing of 3 .bb files complete (1 cached, 2 parsed). 3 targets, 0 skipped, 0 masked, 0 errors.
Recipe Name                                    Latest Version         Preferred Version          Required Version
===========                                    ==============         =================          ================

first                                                :1.0-r0
printhello                                           :1.0-r0
second                                               :0.1-r0

Summary: There were 4 WARNING messages.
```

5 Execute bitbake -c listtasks second and should list do_listtasks, do_showdata, do_testpatch and do_build tasks.

6 Next execute bitbake second and check tmp/work/second-0.1-r0/temp/log.task_order and it should list log.do_testpatch followed by log.do_build

7 Next thing to confirm is which do_build (from base.bbclass or testbuild.bbclass) is getting executed. For this, check the log. Make sure that do_build from testbuild.bbclass gets executed.

8 Food for thought
(i) Why is testpatch getting executed only once and do_build every time?
(ii) How can we make testpatch to execute?
(iii) How can we make sure testpatch executes everytime?

Adding an additional Layer & reusing the class

The object here is to add an additional layer and reuse the existing class

1 Create the directory by name meta-mylayer under sources and add all the necessary stuff to get it listed as layer. 'bitbake-layers show-layers' should output below:

```
pradeep@pradeep-lp3:/media/pradeep/Data3/Trainings/Yocto/Bitbake/sources$ bitbake-layers show-layers
NOTE: Starting bitbake server...
WARNING: Layer meta should set LAYERSERIES_COMPAT_meta in its conf/layer.conf file to list the core layer names it
 is compatible with.
WARNING: Layer meta-example should set LAYERSERIES_COMPAT_meta-example in its conf/layer.conf file to list the cor
e layer names it is compatible with.
WARNING: Layer meta-mylayer should set LAYERSERIES_COMPAT_meta-mylayer in its conf/layer.conf file to list the cor
e layer names it is compatible with.
layer                path                                                                                priority
==========================================================================================================================
meta                 /media/pradeep/Data3/Trainings/Yocto/Bitbake/sources/meta                           1
meta-example         /media/pradeep/Data3/Trainings/Yocto/Bitbake/sources/meta-example                   1
meta-mylayer         /media/pradeep/Data3/Trainings/Yocto/Bitbake/sources/meta-mylayer                   1
```

2 Next, create the directory by name classes under meta-mylayer and create the testconf.bbclass under the same. Next, inherit the testbuild class in that and add the following:

```
do_configure () {
        echo "running confibuild_do_configure"
```

}
Promote this function as a task and add it as dependency for do_build

3 Next, step is to create the directories recipes-base/third/ under meta-mylayer and create the recipe by name 'third_0.1.bb'. Inherit the testconf class in third_0.1.bb recipe. List the recipes with 'bitbake -s'.

4 Execute 'bitbake third' and verify the following:
(i) Check the task order where do_configure is executed before do_build
(ii) do_build should be executed from testbuild class

Extending an existing recipe

The objective is to reuse the 'first' recipe in meta-mylayer by extending the same.

1 The requirement is to run the 'patch' task before build task. For this, create the directory by name 'first' under recipes-base. Under first, create the extended recipe by name first_1.0.bbappend with following contents:
python do_patch () {
        bb.note ("first: do_patch")
}
Promote this as a task and make this as a dependency for do_build
2 Next list the tasks for first recipe with bitbake -c listtasks first and this should list patch as one of the tasks
3 Execute 'bitbake first' and patch task should be executed

Using Variables

Global Variables

1 Variables helps in writing the configurable recipes. The users of the such recipes can give those variables the desired values, which in turn can then be used by the recipes. First step is to define the global variable in local.conf
$ echo MYVAR="hello from MYVAR" > build/local.conf
2 Next step is to access the global variable. For this, let's create the new recipe group recipe-vars and a recipe myvar in it:
Add following to meta-mylayer/recipes-vars/myvar/myvar_0.1.bb
DESCRIPTION = "Show access to global MYVAR"
PR = "r1"
do_build () {
    echo "myvar_sh: ${MYVAR}"
}
python do_myvar_py() {
  print ("myvar:" +d.getVar('MYVAR', True))
}
addtask myvar_py before do_build
3 If we now run bitbake myvar and check the log output in the tmp directory, we will see that we indeed have access to the global MYVAR variable. If you are looking for

the log file, search for a file like this:
build/tmp/work/myvar-0.1-r1/temp/log.do_myvar_py.

Local Variables

1 Create the class by name varbuild.bbclass under meta-mylayer/classes

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

2 Use the above class in the recipe.

```
$ vi  meta-mylayer/recipes-vars/varbuild/varbuild_0.1.bb
  DESCRIPTION = "Demonstrate variable usage \
  for setting up a class task"
  PR = "r1"
  BUILDARGS = "my build arguments"
  inherit varbuild
```

3 Running bitbake varbuild will produce log files that shows that the build task respects the variable value which the recipe has set.
This is a very typical way of using BitBake. The general task is defined in a class, like for example download source, configure, make and others, and the recipe sets the needed variables for the task.

Adding the Custom BSP layer

- Add the new layer
  $ cd build_bbb
  $ bitbake-layers show-layers
  $ bitbake-layers create-layer ../sources/meta-mymachine
  $ bitbake-layers add-layer ../sources/meta-mymachine
  $ bitbake-layers show-layers
- Add the support for the machine
  $ cd ../sources/meta-mymachine
  $ mkdir -p conf/machine
  $ cp ../poky/meta-yocto-bsp/conf/machine/beaglebone-yocto.conf machine/demo-board.conf
- Change the few things here:
  $ vi machine/demo-board.conf
  Change the @Name and @Description to something suitable
- Modify MACHINE ?= demo-board in build_bbb/conf/local.conf
- Build the image for the machine
  $ bitbake core-image-minimal

Linux Kernel

Adding the support for the custom kernel

1) $ cd meta-mymachine
2) $ mkdir -p recipes-kernel/linux
3) Add the following in recipes-kernel/linux/linux-yocto_6.6.bbappend
   COMPATIBLE_MACHINE:demo-board = "demo-board"
4) Next, refer ../poky/meta-yocto-bsp/recipes-kernel/linux/linux-yocto_6.6.bbappend and as per that add the following:
   KBRANCH:demo-board = "v6.6/standard/beaglebone"
   KMACHINE:demo-board ?= "beaglebone"
   SRCREV_machine:demo-board ?= "06644f0d7193d7ec39d7fe41939a21953e7a0c65"
   LINUX_VERSION:demo-board = "6.6.21"
5) Next step is to prepare the image
   $ bitbake core-image-minimal
6) The kernel image would be generated under tmp/deploy/images/demo-board/zImage

Adding the custom defconfig for the Kernel

The idea over here is to configure the kernel to add the support for usb ethernet.

1) Execute the menuconfig task for the kernel using the following commaand
   $ bitbake -c menuconfig virtual/kernel
   Enable Device Drivers -> USB Support -> USB Gadget Support -> USB Gadget precomposed configurations (CDC Composite (Ethernet & ACM)
2) Save the config as myconfig & exit the menuconfig
3) The configuration ‘myconfig’ would be found under kernel recipe’s build directory
4) Next step is to add the myconfig as a defconfig for the kernel. For this, first let’s create the directory by name linux-yocto & copy the myconfig under that:
   $ cd meta-mymachine
   $ mkdir recipes-kernel/linux/linux-yocto
   Copy ‘myconfig’ with a name ‘defconfig’ under this folder
5) Next step is modify the kernel recipe to enable the support for defconfig. For this, add the following in recipes-kernel/linux/linux-yocto_6.6.bbappend
   FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"
   SRC_URI += "file://defconfig"
6) Remove KMACHINE:demo-board = “beaglebone”
7) Next, build the kernel as follows:
   $ bitbake -c compile virtual/kernel
8) Next, copy the kernel image from <build_dir>arch/arm/boot/zImage to the uSD card
9) Boot up the target board & execute ‘ifconfig -a’. This should show ‘usb0’ interface

Creating & applying the configuration fragments to the kernel

- Launch the menuconfig for kernel
  $ bitbake -c menuconfig virtual/kernel
  Make the configuration changes such as modifying the host & kernel log buffer under general setup. Save and exit the menuconfig
- Create the configuration fragment

$ bitbake -c diffconfig virtual/kernel
This would generate fragment.cfg under kernel recipe's Work Directory

- Copy the configuration fragment as myconfig.cfg under recipes-kernel/linux/linux-yocto/
- Modify recipes-kernel/linux/linux-yocto_6.6.bbappend to add the following:
  SRC_URI += "file://myconfig.cfg"
- Build the kernel
  $ bitbake -c compile virtual/kernel

Patching the Kernel

- Open the devshell
  $ bitbake -c devshell virtual/kernel
- Open drivers/char/misc.c and add a printk in the init_module
- Close the devshell
- Test the changes by compiling the Kernel
- Next step is to create the patch. For this, open the devshell
  $ bitbake -c devshell virtual/kernel
- git status
- git add drivers/char/misc.c
- Commit the changes with git commit
- Create the patch
  $ git format-patch -1
- Exit the devshell
- Copy the patch to recipes-kernel/linux/files/ with the name test.patch
- Modify recipes-kernel/linux/linux-yocto_6.6.bbappend to add the following:
  SRC_URI += "file://test.patch"
- Compile the kernel
  $ bitbake -c virtual/kernel
- compile the core-image-minimal

Applying the patches using the SCC scripts

1. Open Kernel devshell
   $ bitbake -c devshell virtual/kernel
2. Copy Kconfig, first_driver.c & Makefile from <repo>Yocto/KernelChanges into the <kernel-source>/drivers/char/
3. Next, commit the changes & create the patch
   $ git format-patch -1 -o <path to meta-mt>/recipes-kernel/linux/files/test.patch
   Exit the devshell
4. Next step is to run the menuconfig task
   $ bitbake -c menuconfig virtual/kernel
   Select the 'My Test Driver' under Drivers->Character Driver
   Save & exit the menuconfig
5. Create the configuration fragment
   $ bitbake -c diffconfig virtual/kernel
   This would generate fragment.cfg under work Directory

6 Copy the configuration fragment as add_driver.cfg under recipes-kernel/linux/linux-yocto/
7 Next step is create the .SCC file. For this, refer the test.scc file under <repo>/Yocto/KernelChanges. And on the similar lines, create the corresponding test.scc file
8 Modify the linux/linux-yocto_6.6.bbappend to add the following:
SRC_URI += “https://test.scc”
9 Compile the kernel
$ bitbake -c virtual/kernel

Building the out of tree Module

9 $ cd meta-mymachine
10 $ mkdir -p recipes-kernel/hello-world/files
11 Copy the recipe and driver code
$ cp <repo>/Yocto/KernelModule/hello-world.bb recipes-kernel/hello-world/
$ cp <repo>/Yocto/KernelModule/hello_world.c recipes-kernel/hello-world/files
$ $ cp <repo>/Yocto/KernelModule/Makefile recipes-kernel/hello-world/files
12 Get into the build_bbb directoy
$ cd build_bbb
$ bitbake hello-world
This generates hello_world.ko under Build Directory of Kernel’s recipe

Uboot

Customize the u-boot

1. Open the devshell
$ bitbake -c devshell virtual/bootloader
2. Get into the cmd directory
$ cd cmd
3. $ cp <repo>/Uboot>my_print.c .
4. Close the devshell
5. Test the changes by compiling the uboot
$ bitbake -f -c virtual/bootloader
6. $ bitbake core-image-minimal

Add the support for custom uboot

1. Navigate to meta-mt
$ cd meta-mymachine
2. Create the respective directories for uboot
$ mkdir -p recipes-bsp/u-boot/files
3. Create the file ‘u-boot_%.bbappend’
$ vi recipes-bsp/u-boot/
Add FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
4. Open the devshell
$ bitbake -c devshell virtual/bootloader
5. git status
6. git add my_print.c Makefile

7. Commit the changes
8. Create the patch
   $ git format-patch -1 -o <build_bbb>/sources/meta-mt/recipes-bsp/u-boot/files/
9. Exit the devshell
10. Rename the patch
    $ cd recipes-bsp/u-boot/files
    $ mv <patch_name> cmd_test.patch
11. Open the u-boot_%.bbappend file and add
    SRC_URI += "file://cmd_test.patch"
12. Build the u-boot
    $ bitbake -c virtual/bootloader
13. Build the the core-image-minimal

Application Integration

Building the C Program

1. Create the recipe with the name helloworld_1.0.bb under recipe-examples/helloworld and include the following contents:
   DESCRIPTION, SECTION and SRC_URI
2. Next, create the file by name helloworld.c under recipe-examples/helloworld/helloworld/
3. Build the application:
   $ bitbake helloworld
   This would generate the executable under Build Directory for this recipe
4. Next step is to modify the program & create the patch for the same. For this, open the devshell:
   $ bitbake -c devshell helloworld
   Next, create the git repo & add the helloworld.c
   $ git init
   $ git add helloworld.c
   $ git commit -s -m "Original revision"
   Open helloworld.c & modify the printf message and commit the changes
   $ git add helloworld.c
   $ git commit -s -m "Change print message"
5. Next step is to create the patch
   $ git format-patch -1 -o
   <path_to_bbb>/sources/meta-mt/recipes-example/helloworld/helloworld/
6. Update the recipe to add the patch file to SRC_URI and build the helloworld package
   $ bitbake -c clean helloworld
   $ bitbake helloworld
   This should generate the executable under WORKDIR

Adding the Packages to the Image

1. Create a bbappend file for the image (webos-image) in recipes-core
2. Install the packages with IMAGE_INSTALL
3. Regenerate the image

Creating the Packagegroup and adding the same to the Image

1. Add the folder by name packagegroups under recipes-core
2. Create the packagegroup by name packagegroup-testapp and add the packages such as ethtool, evtest, fbset, i2c-tools and memtester
3. Modify core-image-minimal.bbappend to add the packagegroup

Rootfs Post Processing Command

Add the following in core-image-minimal

```
create_status_dir() {
   touch ${IMAGE_ROOTFS}/home/status
   chmod 777 ${IMAGE_ROOTFS}/home/status
}
ROOTFS_POSTPROCESS_COMMAND += "create_status_dir; "
```

Devtool

Using devtool to create the new recipe

1 Source the build environment & get the source code from Git
$ devtool add testdev https://github.com/embitude/testdev.git
If it gives out an error - “Fetcher failure: Unable to resolve 'master' in upstream git repository in git”, then use the following the main branches
$ devtool add testdev https://github.com/embitude/testdev.git –srcbranch=main
This would create the workspace for the recipe under <BUILD_DIR>/workspace/recipes/testdev/testdev_git.bb and sources would be fetched under <BUILD_DIR>/workspace/sources/testdev/

2 Build the Recipe
$ devtool build testdev

3 Resolve all the errors such as /usr/bin permission denied, file format not recognized
Hint: Use EXTRA_OEMAKE to pass the configuration for Makefile

4 Before finishing the work with devtool, try to build the same with bitbake as it involves QA tests before packaging. Fix the errors reported by the bitbake.

5 Next step is to finalize the build with devtool and move the recipe into the layer
$ devtool finish testdev ../sources/meta-mylayer/recipes-example/
If this gives out an Error – ‘Source Tree not clean’, then we may use -f option for forcing
$ devtool finish -f testdev ../sources/meta-mylayer/recipes-example/
This would create the recipe by name testdev_git .bb under meta-mylayer/recipes-example/

Using devtool to modify the existing application

1 Use the ‘devtool modify’ command to locate, extract and prepare the package files. The command locates the source based on the information in the existing recipe, unpacks the source into the ‘workspace’, applies patches and parses all related recipes
$ devtool modify virtual/bootloader

This will create the boot_2024.01.bbappend under workspace/appends and source for the uboot will be fetched under workspace/sources/

2. Modify the uboot source to added the command by following the same steps as in uboot exercise above
3. Rebuild u-boot using
   $ devtool build u-boot
4. Next step is to confirm if myprint.c is built by getting into the build directory. The build dir can be found under workspace/sources/u-boot/oe-workdir/u-boot-2024.01. Check if myprint.o is present in cmd/ folder of build dir for u-boot.
5. Next it needs to be tested and by u-boot.img into the board
6. Once the testing is successful, commit the changes for the u-boot with git, so that devtool can create the patch:
   $ cd <build_dir>/workspace/sources/u-boot
   $ git add .
   $ git commit -m "Add the Test Command"
7. Finally, use finish to transfer the changes to the layer
   $ devtool finish u-boot <path to Layer>/meta-demo/
8. Clean up the devtool workspace by deleting it with rm -rf