Yocto Exercises

Prerequisite

- Ubuntu 18.04 or higher based host PC or VM with a minimum of 50GB free disk space & better to have atleast 8GB of RAM
- Stable Internet connection to download the packages

Host Environment Setup

- Install the required packages with below command:
- sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat cpio python python3 python3-pip python3-pexpect xz-utils debianutils iputils-ping libsdl1.2-dev xterm minicom

Yocto Setup

Setting up the Yocto for Beaglebone Black

1. $ mkdir sources
2. $ cd sources
3. Get the required layers
   $ git clone -b dunfell git://git.yoctoproject.org/poky.git
   $ git clone -b dunfell git://git.openembedded.org/meta-openembedded
   $ cd ..
   $ source sources/poky/oe-init-build-env build_bbb
4. Modify the local.conf to add the support for the machine
   $ MACHINE ??= "beaglebone-yocto"
5. Build the image
   $ bitbake core-image-minimal
   This would generate the image under build_bbb/tmp/deploy/images/beaglebone-yocto/core-image-minimal-beaglebone-yocto-xxx.rootfs.wic (where xxx would be image generation date & time)

Flashing the image

1. Insert the uSD card in PC/laptop & flash the image:
   $ cd build_bbb/tmp/deploy/images/beaglebone-yocto
   $ dd if=core-image-minimal-beaglebone-yocto-xxx.rootfs.wic of=/dev/<uSD device file> bs=1M
   This would flash the image on the uSD
2. Umount & remove the uSD for the PC. With USB to ttl connected, insert it into the target board & power up the target. As board comes up, the boot up message should appear on the serial terminal i.e minicom

Linux Device Drivers
Setting up the build environment
The idea over here is to prepare the SDK, so as to use it for building out of tree kernel modules.

1. Source the yocto environment & start the SDK build
   $ source sources/poky/oe-init-build-env build_bbb
   $ bitbake meta-toolchain
2. Once the build is complete, the SDK installation script would be placed under build_bbb/tmp/deploy/sdk/ with the name poky-glibc-x86_64-meta-toolchain-cortexa8hf-neon-beaglebone-yocto-toolchain-3.1.14.sh
3. Next step is to install the toolchain. For this, create the directory with name Toolchain (May be under home dir) & execute the script
   $ mkdir Toolchain
   $ cd Toolchain
   $ <path_to_buid_bbb>/tmp/deploy/sdk/poky-glibc-x86_64-meta-toolchain-cortexa8hf-neon-beaglebone-yocto-toolchain-3.1.14.sh
   This would extract the yocto toolchain under Toolchain directory
4. Further to this, for compiling a driver or applicaton, source the environment using:
   $ source <SDK_installation_path>/environment-setup-cortexa8hf-neon-poky-linux-gnueabi

Driver Fundamentals
Write a simple Linux Kernel Module
1. Navigate to the FirstDriver directory under Drivers folder
   $ cd <path_to_training_dir>/Drivers/FirstDriver
2. Update the KERNEL_SOURCE variable in the Makefile to point to the compiled kernel source (Compiled in the above assignment)
3. Build the kernel module with make. This would generate the kernel module by name first_driver.ko
4. Transfer the kernel module to the board.
   $ scp first_driver.ko root@<board_ip>:
   The board_ip for bbb is 192.168.7.2
5. Load the kernel module
   $ insmod first_driver.ko
   This should display "driver registered" on the console
6. Unload the kenel module
   $ rmmod first_driver
   This should display "driver deregistered" on the console

Statically building the driver into the Kernel
This activity intends to describe the steps required to integrate any driver into the Kernel build system.

1. Open the devshell for the kernel
   $ bitbake -c devshell virtual/kernel

2   Copy the file *first_driver.c* from the Drivers/FirstDriver/ to kernel-source/drivers/char/
$ cp Drivers/FirstDriver/first_driver.c /drivers/char

3   Edit the Kconfig file under drivers/char and add the following:
config MY_DRIVER
    tristate "My Driver"
    help
    Adding this small driver to the kernel

4   Edit the makefile under drivers/char to add the following entry:
obj-$(CONFIG_MY_DRIVER) += first_driver.o

5   Once the modifications are done, configure the kernel:
$ make menuconfig
Under the Driver Driver->Character Devices, there will be a menu option *MY Driver*. Just select it, exit and save the config.

6   This will add the *CONFIG_MY_DRIVER=y* entry in the .config, which in turn would be used by Makefile

7   Compile the kernel
$ make zImage

8   Transfer the newly build kernel to the target board
$ mount /dev/mmcblk0p1 /mnt (On board)
$ scp arch/arm/boot/zImage root@<board_ip>:/mnt/

9   Unmount & reboot the board to boot up with updated kernel
$ umount /mnt
$ reboot (On board)

10   Verify if the driver is registered during the boot up
$ dmesg | grep "driver registered"
Reference: https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system

Character Driver
Character Driver Registration

The idea over here is to write a basic character driver to demonstrate the character driver registration and de-registration. Below are the steps:

1. Navigate to the CharDriver directory under Drivers folder
$ cd Drivers/CharDriver

2. Make sure that the KERNEL_SOURCE variable in the Makefile is updated to point to the compiled kernel source

3. Complete all the TODOs (1 and 2) in first_driver.c (Refer apis.txt file in CharDriver for the corresponding APIs)

4. Build the kernel module with make. This would generate the kernel module by name first_char_driver.ko

5. Transfer the kernel module to the board.
$ scp first_char_driver.ko root@<board_ip>:
The board_ip for bbb is 192.168.7.2

6. Load the kernel module
$ insmod first_char_driver.ko

7. This should display "<Major, Minor>: <240, 0>"

NB: The major/minor number may vary in your case

8. Verify if the driver is registered for corresponding major number
   $ cat /proc/devices (first_char_driver should be displayed against Major number 240)

Register the file operations

The idea over is to register the call back handlers with VFS. Below are the steps:

1. Complete all the TODOs (1, 2 & 3) in null.c (Refer apis.txt file in CharDriver for the corresponding APIs)
2. Build the kernel module with make. This would generate the kernel module by name null.ko
3. Transfer the kernel module to the board.
   $ scp null.ko root@<board_ip>:
   The board_ip for bbb is 192.168.7.2
4. Load the kernel module
   $ insmod null.ko
5. This should display "<Major, Minor>: <240, 0>"
   NB: The major/minor number may vary in your case
6. Create the device file as per the major number allocated above
   $ mknod /dev/abc c 240 0
7. Test the write operation
   $ echo "Welcome" > /dev/abc
   (Should display the ascii value of the above string in user space)
8. Test the read operation
   $ cat /dev/abc

Enhancing the driver to send data to user space

In the previous example, invocation of read had no effect except printing the messages from the my_open and my_read. Below are the steps to enhance read operation:

1. Complete TODO 4 in null.c (Refer apis.txt file in CharDriver for the corresponding APIs)
2. Build the kernel module with make. This would generate the kernel module by name null.ko
3. Transfer the kernel module to the board.
   $ scp null.ko root@<board_ip>:
   The board_ip for bbb is 192.168.7.2
4. Load the kernel module
   $ insmod null.ko
   his should display "<Major, Minor>: <240, 0>"
   NB: The major/minor number may vary in your case
5. Create the device file as per the major number allocated above
   $ mknod /dev/abc c 240 0
6. Test the write operation
   $ echo "Welcome" > /dev/abc
   (Should display the ascii value of the above string)
7. Test the read operation

$ cat /dev/abc (Should display 'e', which is last character of string provided during write operation)

Automatic device file creation

1 Complete TODOs (1 to 4) in final_char_driver.c (Refer apis.txt file in CharDriver for the corresponding APIs)
2 Build the kernel module with make. This would generate the kernel module by name final_char_driver.ko
3 Transfer the kernel module to the board.
   $ scp final_char_driver.ko root@<board_ip>:
   The board_ip for bbb is 192.168.7.2
4 Load the kernel module
   $ insmod final_char_driver.ko
   This should create the device file by name say *finalchar0* under /dev and there would corresponding entry /sys/class/char/finalchar0/dev
   NB Class name *char* and device file name *finalchar0* depend on the corresponding strings using in class_create and device_create respectively
5 Test the read operation
   $ cat /dev/finalchar0 (Should display 'A')
6 Test the write operation
   $ echo "1" > /dev/finalchar0 (Nothing is displayed)
7 Unload the driver
   $ rmmod final_char_driver
   The enteries /dev/finalchar0 and /sys/class/char/finalchar0 should disappear

Autoloading the driver

The intention of this activity is to demonstrate the mdev rules

1 Unpack the autoload.tgz. This would create the folder with the name Autoload
   $ tar -xvf autoload.tgz
2 Register mdev as hotplug manager on the board. Each time Linux kernel detects a new device, it will call /sbin/mdev with specific parameters
   $ echo "/sbin/mdev" > /proc/sys/kernel/hotplug (To be execute on board)
3 Create the directory for mdev scripts on the board
   $ mkdir /lib/mdev/ (To be execute on the board)
4 Transfer the mdev configuration file to the board
    $ scp AutoLoad/mdev.conf root@192.168.7.2:/etc/
5 Transfer the autoloader script to the board
   $ scp AutoLoad/autoloader.sh root@192.168.7.2:/lib/mdev/
6 Transfer the driver to the board
   $ scp Templates/CharDriver/final_char_driver.ko root@192.168.7.2:
7 Change the permissions for autoloader.sh on the board
    $ chmod a+x /lib/mdev/autoloader.sh (To be executed on the board)
8 Next step is to plug the pen drive or either carefully remove & re-insert the uSD. This should load the final_char_driver.ko
9 Unplug the pen drive or carefully remove the uSD. This should unload the driver

Controlling the on-board Led

The idea over is to demonstrate controlling the hardware with character driver.

 Below are the steps:

1   Complete all the TODOs (1 & 2) in CharDriver/led.c (Refer apis.txt file in CharDriver for the corresponding APIs).
2   Use pin no. 56 as an argument to gpio_set_value and gpio_get_value. There is macro called GPIO_NUMBER for this.
3   Build the kernel module with make. This would generate the kernel module by name led.ko
4   Transfer the kernel module to the board.
    $ scp led.ko root@192.168.7.2:
5   Compile the application. It can be found under Drivers/Apps
    $ make
    This would generate the executable with name led_ops
6   Transfer the application to the board
    $ scp Apps/led_ops root@192.168.7.2:
7   Load the kernel module
    $ insmod led.ko
8   Execute the application
    ./led_ops /dev/gpio_drv0
    Select open, Try write & read to play around with led

Controlling the on-board Leds using ioctl

The idea over is to demonstrate controlling the hardware by using the ioctl.
Below are the steps:

1   Complete all the TODOs in led_ioctl.c & led_ioctl.h under CharDriver/ (Refer apis.txt file in CharDriver for the corresponding APIs)
2   Copy led_ioctl.h to Apps/
3   Build the kernel module with make. This would generate the kernel module by name led_ioctl.ko
4   Transfer the kernel module to the board.
    $ scp led_ioctl.ko root@192.168.7.2:
5   Compile the application. It can be found under Dirvers/Apps
    $ make
    This would generate the executable with name led_ioctl
6   Transfer the application to the board
    $ scp Apps/led_ioctl root@192.168.7.2:
7   Load the kernel module
    $ insmod led_ioctl.ko
8   Execute the application
    ./led_ioctl /dev/gpio_drv0
    Select open and then try playing around with ioctls for selecting the leds & switching on/off the leds

Platform Drivers & Device Tree

Platform Driver & Device Registration

The idea over here is to demonstrate the usage of platform driver

1. Navigate to Drivers/PlatformDrivers. Execute 'make' & this would generate platfrom_driver.ko and platform_device.ko
2. Transfer platform_driver.ko & platform_device.ko
3. Load the drivers
   $ insmod platform_driver.ko
   $ insmod platform_device.ko
   This should invoke the probe for the Platform Driver

Enhance led driver to enable the platform driver support

The idea over here is to remove the hardcoding of the gpio number in led driver and provide the same through the led device.

1. Navigate to Drivers/PlatformDrivers. Copy CharDriver/led.c as platform_led.c and platform_device.c as led_device.c
2. Modify platform_led.c to add the support for platform driver.
3. Update led_device.c to pass led number as platform data.
4. Build the drivers with make. This would generate the platform_led.ko and led_device.ko.
5. Transfer both the above files to the board and load the same.
   $ insmod platform_led.ko
   $ insmod led_device.ko
   This should create the device file /dev/gpio_drv0
6. Transfer & execute the application
   ./led_ioctl /dev/gpio_drv0
   Select open and then try playing around with ioctls for selecting the leds & switching on/off the leds

Enchance the platform driver to use Device Tree

The objective here is to enhance the platform drivers to enable the support for device tree

1. Navigate to Drivers/PlatformDrivers. Copy platform_led_driver.c as platform_led_dtb.c
2. Modify platform_led.c to add the support for dtb node. Refer, gpio-led.c for the same.
3. Next step is to update the dtb to add the corresponding device tree node. Refer 'am335x-boneblack-common.dtsi' for the same. Compile the dtb by executing the following command:
   $ make dtbs
4. Next step is to transfer the dtb to the board & update it accordingly
   $ mount /dev/mmcblk0p1 /mnt (on board)
   $ scp am335x-boneblack.dtb root@192.168.7.2:/mnt/
   $ umount /mnt (On board)
   $ reboot (Reboot the board)
5. Build the drivers with make. This would generate the platform_led_dtb.ko. Transfer it to the board and load the same.

```
$ insmod platform_led_dtb.ko
```
This should invoke the driver's probe, which in turn would create the device file.

Bitbake Hands-On

Bitbake Setup

1. Clone the bitbake
   ```
   $ cd ${HOME}/Bitbake
   $ git clone https://github.com/openembedded/bitbake
   ```
2. Export the required paths
   ```
   $ export PATH=${HOME}/Bitbake/bitbake/bin/:$PATH
   $ export PYTHONPATH=${HOME}/Bitbake/bitbake/lib:$PYTHONPATH
   ```
3. Verify
   ```
   $ bitbake --version
   ```
4. Create a bitbake project. For this create the directory structure as below:

```
BbTutorial
├── build
│   └── conf
└── meta-tutorial
    ├── classes
    └── conf

5 directories, 0 files
```

5. Add the files as per below:

```
BbTutorial
├── build
│   └── conf
│       └── bblayers.conf
└── meta-tutorial
    ├── classes
    │   └── base.bbclass
    └── conf
        ├── bitbake.conf
        └── layer.conf

5 directories, 4 files
```

6. Create the bblayers.conf file with following contents:
```
BBPATH := "${TOPDIR}"
BBFILES ?= ""
BBLAYERS = "${TOPDIR}/../meta-tutorial"
```

7. Create meta-tutorial/conf/layer.conf with following contents:
```
BBPATH .= ":${LAYERDIR}"
BBFILES += ""
```

8. Copy base.bbclass from bitbake/classes to meta-tutorial/classes & bitbake.conf from btibake/conf/ to meta-tutorial/conf/
9. Naviage to build directory & execute the bitbake
   $ cd BbTutorial/build
   $ bitbake
   This should report:
   Nothing to do. Use 'bitbake world' to build everything or run 'bitbake –help' for usage information

Adding the recipe

1. Execute the bitbake as below:
   $ bitbake -s
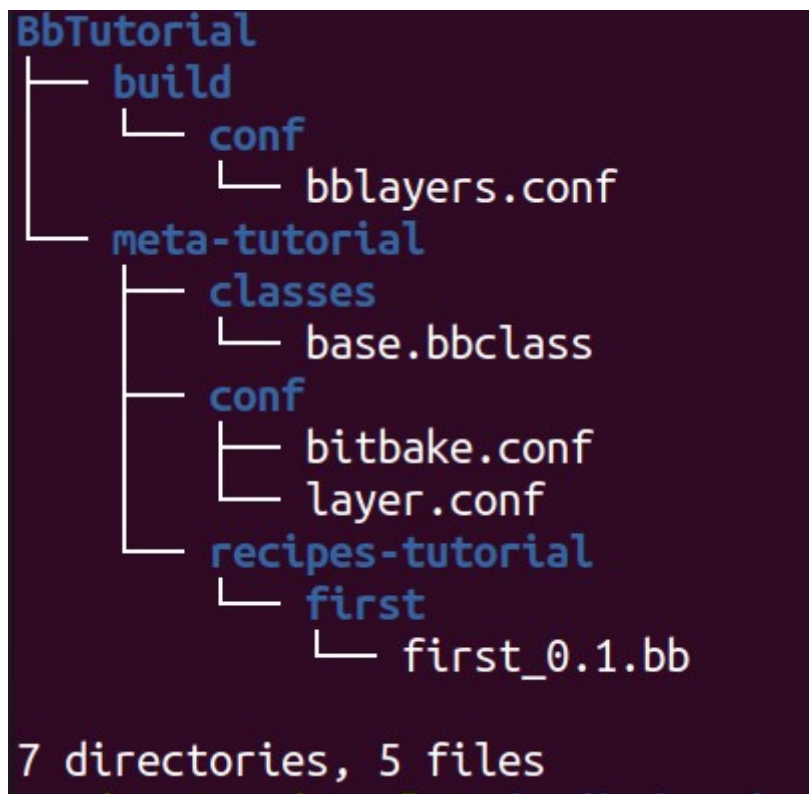2. Set the cache location by adding the below things in meta-tutorial/conf/bitbake.conf
   CACHE = "${TMPDIR}/cache/default"

3. Adding the recipe location to the layer. For this, edit meta-tutorial/conf/layer.conf to add the below:

   BBFILES += "${LAYERDIR}/recipes-*/*/*.bb"
4. Create directory for the recipes. This is needed to group the related recipes in the same directory.
   $ cd BbTutorial
   $ mkdir -p meta-tutorial/recipes-tutorial/first
5. Next, create first recipe in the first group & add the contents as well
   $ vi meta-tutorial/recipes-tutorial/first/first_0.1.bb
   Description = "I am the first recipe"
   PR= "r1"
   do_build () {
           echo "first: Running the build shell script"
   }
   The directoy structure looks as below:

```
BbTutorial
├── build
│   └── conf
│       └── bblayers.conf
└── meta-tutorial
    ├── classes
    │   └── base.bbclass
    ├── conf
    │   ├── bitbake.conf
    │   └── layer.conf
    └── recipes-tutorial
        └── first
            └── first_0.1.bb

7 directories, 5 files
```

6. Execute the bitbake command as below:
   $ cd BbTutorial/build
   $ bitbake -s
   If everything is correct, the bitbake would list the first recipe
7. Bitbake the first recipe
   $ bitbake first
   This would generate the logs file – build/tmp/work/first-0.1-r1/temp/log.do_build

Adding the class & using the same

1. Create the file for the class & add the contents
   Add following in meta-tutorial/classes/mybuild.bbclass
   addtask build
   mybuild_do_build() {
       echo "Executing mybuild_do_build."
   }
   EXPORT_FUNCTIONS do_build
2. Add the new recipe folder & add the file
   $ mkdir meta-tutorial/recipes-tutorial/second
   Add the following in meta-tutorial/recipes-tutorial/second_1.0.bb
   DESCRIPTION = "I am the second recipe"
   PR = "r1"
   inherit mybuild
   def pyfunc(o):
   print (dir(o))
   python do_mypatch () {

```
        bb.note ("running mypatch")
        pyfunc(d)
}
addtask mypatch before do_build
```
3. List the recipes & tasks
   ```
   $ bitbake -s
   $ bitbake -c listtasks second
   ```
4. Build one recipe
   ```
   $ bitbake second
   ```
5. Executing the specific task
   ```
   bitbake -c mypatch second
   ```
6. Build everything
   ```
   bitbake world
   ```
7. Verify the logs
   ```
   $ build/tmp/work/first_1.0-r1/temp
   ```

Adding an additional layer

1. Create a new folder named meta-two
   ```
   $ cd BbTutorial
   $ mkdir meta-two
   ```
2. Configure the new layer
   ```
   $ mkdir meta-two/conf
   ```
   Copy the layer.conf from meta-tutorial/conf to meta-two/conf
3. Updating the bitbake configuration to add the support for new layer. For this, edit build/conf/bblayers.conf and extend the BBLAYERS variable
   ```
   BBLAYERS = " \
        ${TOPDIR}/../meta-tutorial \
        ${TOPDIR}/../meta-two \
   "
   ```
4. Enchancing the Layer Configuration. Add the following in meta-tutorial/conf/layer.conf
   ```
   BBFILE_COLLECTIONS += "tutorial"
   BBFILE_PATTERN_tutorial = "^${LAYERDIR}"
   BBFILE_PRIORITY_tutorial = "5"
   ```
5. Add above things for meta-two as well:
   ```
   BBFILE_COLLECTIONS += "two"
   BBFILE_PATTERN_two = "^${LAYERDIR}"
   BBFILE_PRIORITY_two = "5"
   ```
6. Examine the layers
   ```
   $ bitbake-layers show-layers
   ```
7. Layer Compatibility
   Define the Layer series core name by defining the project core name.
   Add the following to meta-tutorial/conf/layer.conf
   ```
   LAYERSERIES_CORENAME = "bitbakeguide"
   ```
   Next define the LAYERSERIES_COMPAT variable in meta-tutorial/conf/layer.conf
   ```
   LAYERSERIES_COMPAT_tutorial = "bitbakeguide"
   ```

Add the similar line to meta-two/conf/layer.conf
LAYERSERIES_COMPAT_tutorial = "bitbakeguide"

8. Layer dependency
Add the dependency for 'tutorial' layer in layer 'two' conf
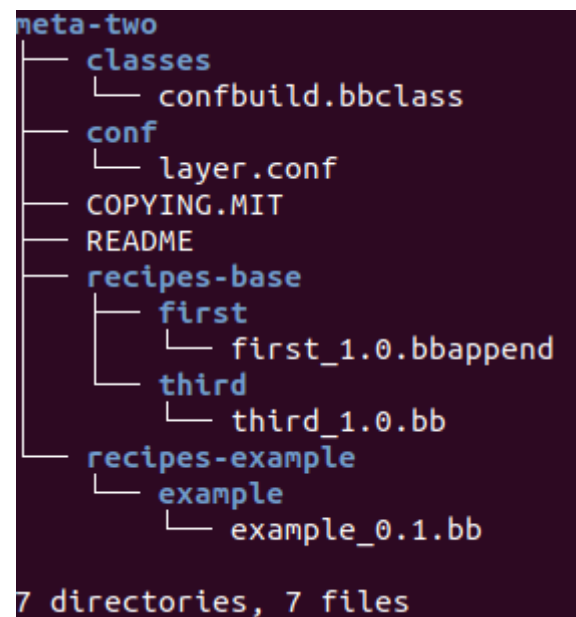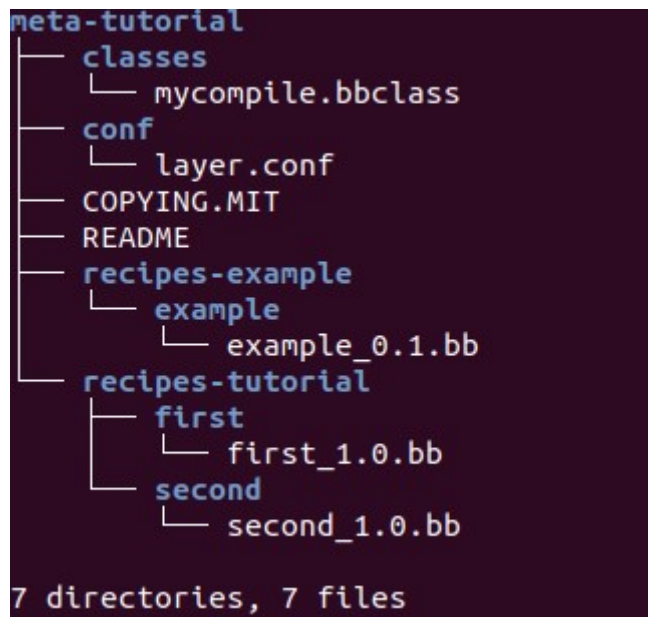LAYERDEPENDS_two = "tutorial"

9. Class Inheritence – The idea over here is to reuse the 'mybuild' class. First, create a class confbuild.bbclass under meta-two/classes and add the following:

```
inherit mybuild
confbuild_do_config () {
        echo "running confibuild_do_configure"
}
addtask do_configure before do_build
EXPORT_FUNCTIONS do_configure
```

10. Next step is to define the new recipe & use the confbuild
Create the file with name meta-two/recipes-base/third/third_0.1.2.bb and add the following:

```
DESCRIPTION = "I am a third recipe"
PR = "r1"
inherit confbuild
```

```
meta-tutorial
├── classes
│   └── mycompile.bbclass
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
├── recipes-example
│   └── example
│       └── example_0.1.bb
└── recipes-tutorial
    ├── first
    │   └── first_1.0.bb
    └── second
        └── second_1.0.bb

7 directories, 7 files
```

```
meta-two
├── classes
│   └── confbuild.bbclass
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
├── recipes-base
│   ├── first
│   │   └── first_1.0.bbappend
│   └── third
│       └── third_1.0.bb
└── recipes-example
    └── example
        └── example_0.1.bb

7 directories, 7 files
```

11. Next, build the 'third' recipe
$ bitbake third (This would execute the configure & build tasks for the third)

Extending an existing recipe

1. The append files have an extenstion of .bbappend. To be able to use append files, the layers needs to be set up to load also them in addition to normal recipes. For this, modify 'BBFILES' variable as follows:
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend

2. Next step is to reuse & extend the existing 'first' recipe in the meta-two layer. That's the reason to use LAYERDEPENDS_two. The requirement is to run a patch function before running the build task. For this, create the bbappend file first
   $ mkdir -p meta-two/recipes-base/first
   Add the following to meta-two/recipes-base/first/first_0.1.bbappend file:
   python do_patch () {
           bb.note ("first: do_patch")
   }
   addtask patch before do_build
3. Next, list the tasks with
   $ bitbake -c listtasks first
   This should show up do_patch task as well

Including files

The idea over here is to demonstrate the usage of include & require directives. Its worth mentioning that include & require file name is relative to any directory in BBPATH

1. Modify bitbake.conf to include the local.conf as below:
   require local.conf
2. Execute bitbake. This would show a error message. To fix this, create local.conf under build directory

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-two/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-two/classes
   ```
   varbuild_do_build () {
     echo "build with args: ${BUILDARGS}"
   }
   ```
   addtask build
   EXPORT_FUNCTIONS do_build
2. Use the above class in the recipe.
   $ vi meta-two/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

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

Adding the support for the custom kernel

1. $ cd meta-mt
2. $ mkdir -p .recipes-kernel/linux
3. Add the following in recipes-kernel/linux/linux-yocto_5.4.bbappend
   COMPATIBLE_MACHINE += "|demo-board"

4. Next, refer ../sources/poky/meta-yocto-bsp/recipes-kernel/linux/linux-yocto_5.4.bbappend and as per that add the following:
KBRANCH_demo-board = "v5.4/standard/beaglebone"
KMACHINE_demo-board ?= "beaglebone"
SRCREV_machine_demo-board ?= "706efec4c1e270ec5dda92275898cd465dfdc7dd"
LINUX_VERSION_demo-board = "5.4.58"
5. Next step is to prepare the image
6. $ bitbake core-image-minimal
7. 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 would be found 'myconfig' would be found under build_bbb/tmp/work/demo_board-poky-linux-gnueabi/linux-yocto/5.4.58+gitAUTOINC+e8c675c7e1_706efec4c1-r0/linux-demo_board-standard-build/myconfig
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-mt
$ mkdir recipes-kernel/linux/linux-yocto
Copy myconfig as defconfig under this folder
$ cp ../../build_bbb/tmp/work/demo_board-poky-linux-gnueabi/linux-yocto/
5.4.58+gitAUTOINC+e8c675c7e1_706efec4c1-r0/linux-demo_board-standard-build/myconfig recipes-kernel/linux/linux-yocto/defconfig
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_5.4.bbappend
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://defconfig"
6. Next, build the kernel as follows:
$ bitbake -c compile virtual/kernel
7. Next, copy the kernel image from build_bbb/tmp/work/demo_board-poky-linux-gnueabi/linux-yocto/5.4.58+gitAUTOINC+e8c675c7e1_706efec4c1-r0/linux-demo_board-standard-build/arch/arm/boot/zImage to the uSD card.
8. Boot up the target board & execute 'ifconfig'. This should show 'usb0' interface

Creating & applying the configuration fragments to the kernel

1. 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
2. Create the configuration fragment
$ bitbake -c diffconfig virtual/kernel

This would generate fragment.cfg under build_bbb/tmp/work/demo_board-poky-linux-gnueabi/linux-yocto/5.4.58+gitAUTOINC+e8c675c7e1_706efec4c1-r0/

3. Copy the configuration fragment as myconfig.cfg under recipes-kernel/linux/linux-yocto/
4. Modify recipes-kernel/linux/linux-yocto_5.4.bbappend to add the following:
   SRC_URI += "file://myconfig.cfg"
5. Build the kernel
   $ bitbake -c compile virtual/kernel

Patching the Kernel

1. Open the devshell
   $ bitbake -c devshell virtual/kernel
2. Open drivers/char/misc.c and add a printk in the init_module
3. Close the devshell
4. Test the changes by compiling the Kernel
5. Next step is to create the patch. For this, open the devshell
   $ bitbake -c devshell virtual/kernel
6. git status
7. git add drivers/char/misc.c
8. Commit the changes with git commit
9. Create the patch
   $ git format-patch -1
10. Exit the devshell
11. Copy the patch to recipes-kernel/linux/files/ with the name test.patch
12. Modify recipes-kernel/linux/linux-philips_5.4.bb to add the following:
    SRC_URI += "file://test.patch"
13. Compile the kernel
    $ bitbake -c virtual/kernel
14. compile the core-image-minimal

Applying the patches using the SCC scripts

1. Get into the recipes-kernel under meta-philips
2. Copy the patches & the .scc file into the recipe directory
   $ cp Day5/Scc/* linux/linux-philips/
3. Modify the linux/linux-philips_5.4.bb to add the following:
   SRC_URI += "test.scc"
4. Compile the kernel
   $ bitbake -c virtual/kernel

Building the out of tree Module

1. $ cd meta-mt
2. $ mkdir recipes-kernel/hello-world/files
3. Copy the recipe and driver code
   $ cp Day3/Driver/hello-work.bb recipes-kernel/hello-world/
   $ cp Day3/Driver/hello_world.c recipes-kernel/hello-world/files

$ $ cp Day3/Driver/Makefile recipes-kernel/hello-world/files
4. Get into the build_bbb directoy
   $ cd build_bbb
   $ bitbake hello-world
   This generates hello_world.ko under ../../build_bbb/tmp/work/demo_board-poky-linux-gnueabi/hello-world/1.0-r0/

Customize the u-boot
1. Open the devshell
    $ bitbake -c devshell virtual/bootloader
2. Get into the cmd directory
    $ cd cmd
3. $ cp Day2/Uboot/my_print.c Day2/Makefile .
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-mt
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 <yocto_build>/sources/meta-philips/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

Building the C Program

1. Create the recipe with the name helloworld_1.0.bb under recipe-examples/helloworld and include the following contents:
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_bbb/tmp/work/cortexa8hf-neon-poky-linux-gnueabi/helloworld/1.0-r0
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 cleanall helloworld
   $ bitbake helloworld
   This should generate the executable under WORKDIR

Building the Program located on git

1. Get into the meta-mt/recipes-example/helloworld directory & make a copy of helloworld_1.0.bb
   $ cd meta-mt/recipe-example/helloworld
   $ mkdir recipes-example
   $ cp helloworld_1.0.bb helloworld-git_1.0.bb
2. Modify SRC_URI in helloworld_git_1.0.bb to point to the git repo
3. Next, build the recipe
   $ bitbake helloworld-git
4. Fix the errors if any

Static Library based build

1. Create a directory with Libraries and under that another directoy by StaticLib
   $ mkdir Libraries
   $ cd Libraries
   $ mkdir StaticLib
   $ cd StaticLib
2. Next thing is to create two files hello.c and world.c with following contents

```
hello.c:
char *hello()
{
        return "Hello";
}
world.c:
char *world()
{
    return "World";
}
```

3. Next, source the build environment
   $ source Toolchain/environment-setup-cortexa8hf-neon-poky-linux-gnueabi
4. Compile the libraries
   ${CC} -c hello.c world.c
   ${AR} -cvq libhelloworld.a hello.o world.o
   ${AR} -t libhelloworld.a
5. Next, create the helloworld.c which invokes the library functions
   helloworld.c
   ```
   #include <stdio.h>
   int main (void)
   {
       return printf("%s %s\n",hello(),world());
   }
   ```
6. Compile the applications
   ${CC} -o helloworld helloworld.c libhelloworld.a
   This would generate the executable 'helloworld'

Static Library based Yocot recipe

1. Create the directory with the name libhello-static under recipes-example
   $ cd recipe-example
   $ mkdir -p libhello-static/libhello-static
2. Next step is to copy hello.c, helloworld.h & world.c to recipes-example/libhello-static/libhello-static
3. Next step is to create the pkg-config build settings file with .pc suffix. It is distrubuted with the library and is installed in a common location known to the pkg-tool. Place the helloworld.pc pkg-conifg file inside libhello-static/libhello-static/
4. Next step is to create the recipe say libhello-static_1.0.bb, which builds the library libhelloworld.a.
   $ bitbake libhello-static
5. Next step is to build the program which uses the above library. For this, first create the directories hello-static/hello-static and copy the helloworld.c file under it.
   helloworld.c:
   ```
   #include <stdio.h>
   #include "helloworld.h"
   int main (void)
   {
   ```

```
        return printf("%s %s\n",hello(),world());
    }
```
6. Next, create the recipe hello-static/hello-static_1.0.bb to build this program
   $ bitbake hello-static

Dynamic/Shared Library based Yocot recipe

1. Create the directory with the name libhello-dyn under recipes-example
   $ cd recipe-example
   $ mkdir -p libhello-dyn/libhello-dyn
2. Next step is to copy hello.c, helloworld.h & world.c to
   recipes-example/libhello-dyn/libhello-dyn
3. Next step is to create the pkg-config build settings file with .pc suffix. It is distrubuted
   with the library and is installed in a common location known to the pkg-tool. Place the
   helloworld.pc pkg-conifg file inside libhello-dyn/libhello-dyn/
4. Next step is to create the recipe say libhello-dyn_1.0.bb, which builds the library
   libhelloworld.a.
   $ bitbake libhello-dyn
5. Next step is to build the program which uses the above library. For this, first create
   the directories hello-dyn/hello-dyn and copy the helloworld.c file under it.
   $ bitbake hello-dync

Building an autotools based project

1. Get into the meta-mt & create the directory by recipes-example
   $ cd meta-mt
   $ mkdir recipes-example

Creating a simple recipe and installing it in the rootfs

2. Get into the meta-philips
   $ cd meta-mt
3. Create the recipe directory
   $ mkdir -p recipes-example/helloworld/files
   $ cp Recipes/helloworld.c recipes-example/helloworld/files/
4. Copy the recipe & the simple program
   $ cp Recipes/helloworld_0.1.bb recipes-example/helloworld/
   $ cp Recipes/helloworld.c recipes-example/helloworld/files/
5. Test the recipe
   $ bitbake hellloworld
6. Create the .bbappend file for core-image-minimal
   $ cd meta-mt
   $ mkdir -p recipes-mt/images/
   Add the following in core-image-minimal.bbappend
   $ vi recipes-mt/images/core-image-minimal.bbaappend
   Add 'IMAGE_INSTALL += " helloworld"
7. Build the image
   $ bitbake core-image-minimal

Creating a simple sysV init service and installing it in the rootfs

1. Get into the meta-mt
   $ cd meta-mt
2. Create the recipe directory for initd
   $ mkdir -p recipes-example/initd/files
   $ cp Initd/sysvtest recipes-example/initd/files/
3. Copy the recipe & the systemd service
   $ cp Initd/systvinit_1.0.bb recipes-example/initd/
   $ cp Initd/sysvtest recipes-example/initd/files/
4. Install the reciple into the image
   Add the following in core-image-minimal.bbappend
   $ vi recipes-mt/images/core-image-minimal.bbaappend
   Add 'IMAGE_INSTALL += " sysvinit"
5. Build the image
   $ bitbake core-image-minimal

Creating a simple systemd service and installing it in the rootfs

1. Get into the meta-mt
   $ cd meta-mt
2. Create the recipe directory for systemd
   $ mkdir -p recipes-example/systemd/files
3. Copy the recipe & the systemd service
   $ cp Systemd/hellosystemd_1.00.bb recipes-example/systemd/
   $ cp Recipes/helloworld.service recipes-example/systemd/files/
4. Install the reciple into the image
   Add the following in core-image-minimal.bbappend
   $ vi recipes-mt/images/core-image-minimal.bbaappend
   Add 'IMAGE_INSTALL += " hellosystemd"
5. Build the image
   $ bitbake core-image-minimal
6. Insert the uSD in the PC and flash the rootfs by using the following command:
   $ sudo tar -C <second partition mount point> -xvf
   tmp/deploy/images/demo-board/core-image-minimal-demo-board-xxx.rootfs.tar.bz2
7. Once contents are copied, insert the uSD in the board & boot up with the same.
   Execute the following command to check if the systemd service had started
   $ systemctl status hellosystem.service
8. Execute below to verify the logs
   $ journalctl | grep World
   PS To enable systemd init manager, Add the following in local.conf
   DISTRO_FEATURES_append = " systemd"
   VIRTUAL-RUNTIME_init_manager = "systemd"

Using devtool to create the new recipe

1. Get the Source Code
   $ cp -r Day4/TestProg $HOME/

2. Get into the eSDK
   $ cd $HOME/poky_sdk
3. Use devtool to set up the workspace and create the recipe
   $ devtool add testdev $HOME/TestProg/
   This would create the recipe by name testdev_0.1.bb under <esdk dir>/workspace/recipes/testdev
4. Build the recipe
   $ devtool build testdev
   The corresponding binary would be generated under
   <eSDK_dir>/tmp/work/aarch64-poky-linux/testdev/0.1-r0/image/usr/bin/test
5. Test the generated binary

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 helloworld
   The package is unpacked at <eSDK_dir>/workspace/sources/helloworld/
2. Modify helloworld.c
3. Rebuild helloworld
   $ devtool build helloworld
4. Test the helloworld application
5. Once tested, the next step is to commit the changes
   $ cd <eSDK_dir>/workspace/sources/helloworld/
   $ git status
   $ git helloworld.c
   $ git commit
6. Turn the commit you have made into the patch
   $ cd <eSDK_dir>
   $ devtool update-recipe helloworld
   The patch would be placed under
   <eSDK_dir>/layers/meta-philips/recipes-example/helloworld/files/0001-Modification-to-test-the-dev-tool.patch

Using devtool to modify the existing kernel module

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 hello-world
   The package is unpacked at <eSDK_dir>/workspace/sources/hello-world/
2. Modify hello_world.c
3. Rebuild hello-world
   $ devtool build hello-world
4. Test the hello-world.ko module

Using Packagegroups in Yocto

The idea over here is to create 2 images, one for development and other for production.

1. Get into the meta-philips
2. Copy the image recipes into images folder
   $ cp Day4/PackageGroups/demo-board-* recipes-philips/images/
3. Create the directory by name packagegroups under recipes-philips & copy the packagegroup from Day4
   $ cp Day4/PackageGroups/packagegroup-philips-testapps.bb recipes-philips/packagegroup/
4. The image recipe demo-board-image-dev.bb utilizes the packagroup

Remote debugging with gdbserver

1. Get into the SDK directory and source the environment
   $ cd /opt/poky/3.1.10
   $ source environment-setup-aarch64-poky-linux
2. Start the program with gdbserver on the target
3. gdbserver :10000 helloworld
   This should show the message:
   Attached; pid = 160
   Listening on port 10000
4. On the host, start the gdb client
   $ gdb sysroots/aarch64-poky-linux/usr/bin/helloworld
5. Set the sysroot
   $ set sysroot /opt/poky/3.1.10/sysroots/aarch64-poky-linux/
6. $ target remote <ip_addr>:10000
   This would estabilish the connection between the gdb client & the server

Custom image partitioning with wic

1. Get into the meta-philips
2. Create a directory by name wic and copy the .wks file
   $ mkdir wic
   $ cp $Day5/Wic/sdimage-demo-board.wks wic/
3. Modify the meta-philips/conf/demo-board.conf to add the following
   WKS_FILE = "sdimage-demo-board.wks"
4. Rebuild the image
   $ bitbake core-image-minimal
   This should image with 3 partitions

Wic related commands

Wic related commands are available once the bitbake environment is sourced or as part of SDK

1. $ wic ls <image>
   This would list the partition details on the wic file
2. Mounting the partition of wic file
   $ loopdev1=`losetup -f`
3. Use losetup to create the loop device.

  $ sudo losetup -o <starting offset> --sizelimit <size of partition> $loopdev1 <image>

  starting offset and size can be figured out from the 'wic ls'

4. Mount the partition

  $ sudo mount $loodev1 /mnt

5. Once done unmount and delete the loop device

  $ sudo umount /mnt

  $ losetup -d $loopdev1