Orange Pi

**Chapter 1: why Linux?**

First question why do we need embedded linux for embedded sytems ? well this post from IBM back in 2001 is a very good introductory <https://www.ibm.com/developerworks/library/l-embl/index.html>

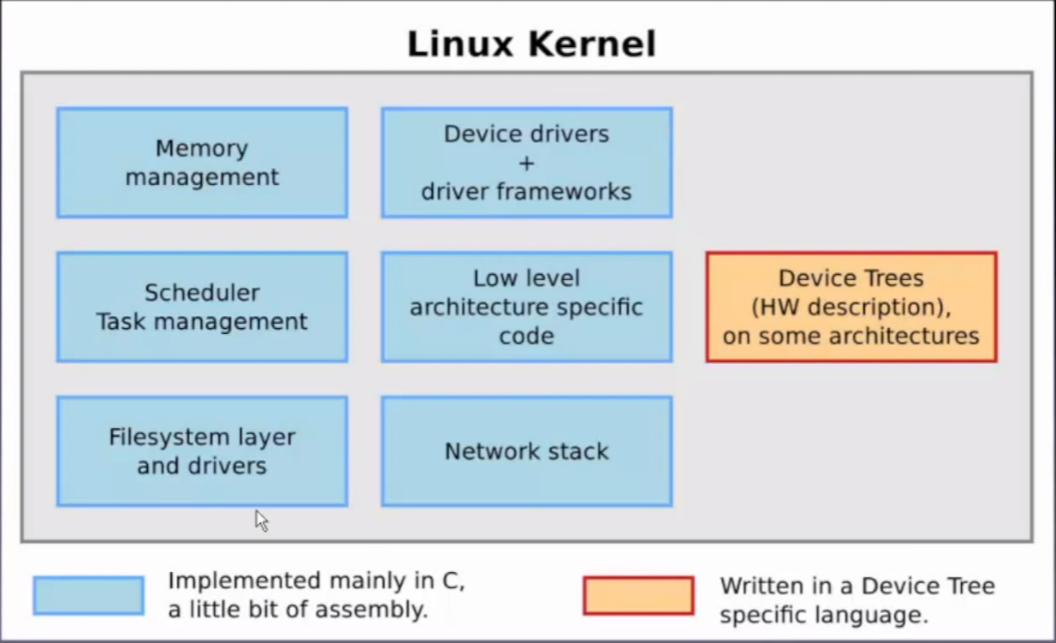
Here are some of the key points

* A Linux system can actually be adapted to work with as little as 256 KB ROM and 512 KB RAM
* Another benefit of using an open source operating system like Embedded Linux over a traditional real-time operating system (RTOS), is that the Linux development community tends to support new IP and other protocols faster than RTOS vendors do.
* The core Linux operating system itself has a fairly simple micro-kernel architecture. Networking and file systems are layered on top of the micro-kernel in modular fashion. Drivers and other features can be either compiled in or added to the kernel at run-time as loadable modules.
* Linux can run on most microprocessors with a wide range of peripherals and has a ready inventory of off-the-shelf applications. So can run on cortex M0 ? TODO
* Linux also supports multicore processor system. For example you can run linux system on one processor and running GUI on another
* Linux is not for hard real time applications

**Chapter 2: Linux Components**

Yocto and Buildroot can both give you the same end product: these are

1. RootFS root filesystem image for your embedded device,
2. kernel,
3. bootloader, and
4. Compatible toolchain.



**RootFS :** Root File system , it holds all the files , programs, directories

**Device Tree:** is a text file describing all the device and their drivers

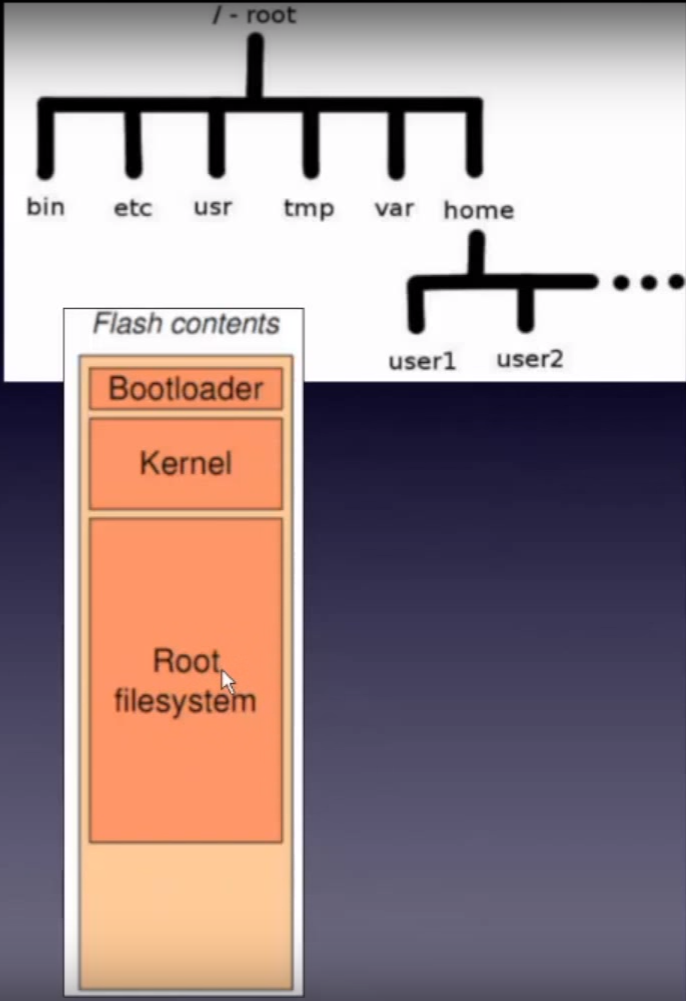
**Kernel:** the main thing, the OS core

**Bootloader:**

FSBL is first stage bootloader

U-Boot : boots the Linux kernel

In Flash/SD the orientation is like this

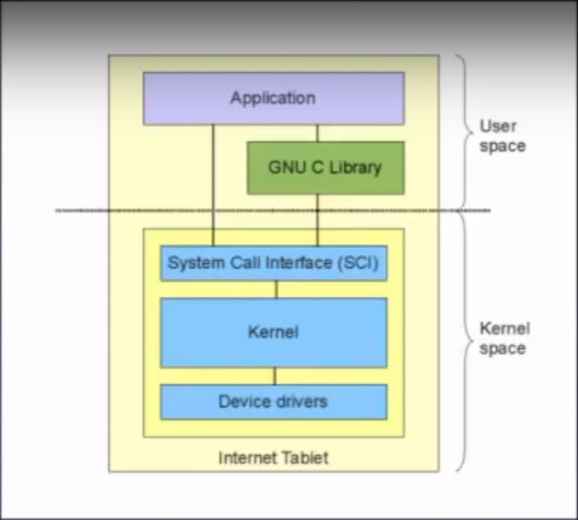


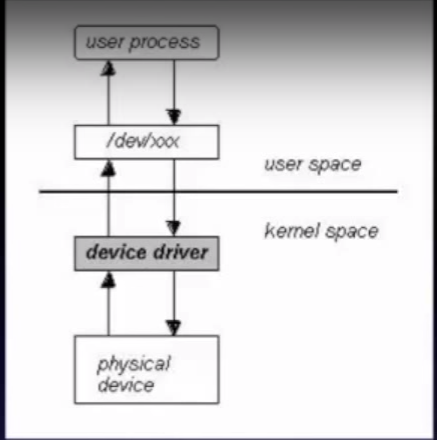
**Toolchain :** the tools needed to compile , port the kernel , drivers and etc

**Memory :** In linux memory is divided into 2 parts

* + 1. Kernel space
    2. User space : user application runs

They communicate through system calls

The spaces can be depicted like this



On linux the hardware parts are made available to the user space through files

The job of the device driver is to make the user space think that the hardware is just a file that is placed in the /dev/ directory

An application must make a request to the kernel via system calls when it needs to use a certain hardware resource. The kernel in turn evaluates the request and manages the hardware resource on behalf of the userspace application.

Comparing to baremetal systems where when we create a device driver we actually have the access to physical memory , in linux the memory is virtual and is translated by a hardware called MMU (memory management unit)

<https://www.youtube.com/watch?v=92-uLpWIRaI>

**Chapter 3: Hands on experience with orange Pi**

I have orange pi PC with me

This link has the differences listed between different Orange Pi boards

<https://diyprojects.io/orange-pi-mini-pc-low-cost-competitor-raspberry-pi/#.WmIQ_aiWYuQ>

orange pi PC board has Allwinner H3 SoC which has a quad core cortex -A ARM CPU

Now for hands on experience we are going to load a ready-made ARM compatible distribution, which in this case is Armbian

Go to <https://www.armbian.com/download/> and download the distro for orange Pi PC

Extract it

Firstly format the SD card using SD-formatter

Then burn the image using Etcher

Now use the 5V 2A adapter to power up the board and connect it to hdmi / put uart ttl to the rx tx pins and you will see the boot message

I am using Lubuntu as VM

\*MagPi magazines got a hell lot of references

**Chapter 4: Device Driver write in Raspberry pi**

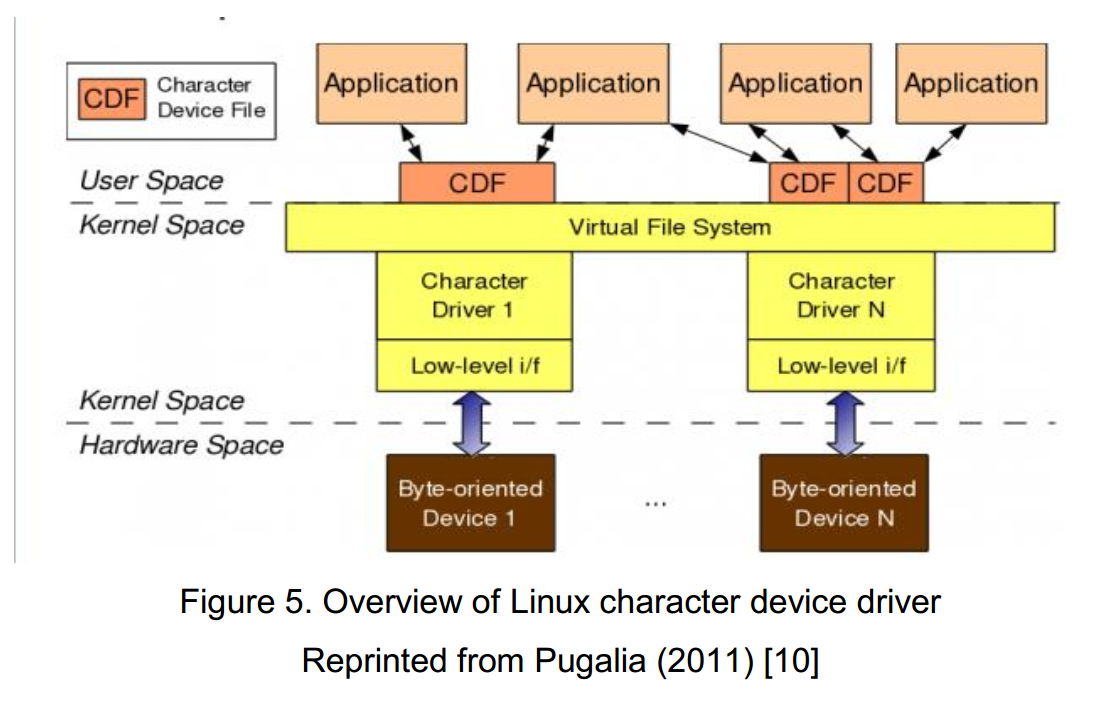
Device driver is designed to control a hardware device & consists of a set of low level interface Raspberry Pi also has ARM processor

In linux there are 3 kinds of devices

1. Network
2. Storage
3. Character

These all are visible to user space in the in /dev directory

A majority of Linux device drivers are character device drivers, these are capable of capturing raw data from devices. Character device driver is accessed by character device file at user space



The device driver appears as a device file in the user space. User space application performs file operations on the device file

Now how to write for example a GPIO device driver for raspberry pi

Firstly the device driver should provide file like operation/ **methods**. The following methods so that the user app can treat the device driver as a file. These methods are

* llseek()
* read()
* write()
* open()
* release()

Other than these methods the device driver should have an initialization function, in this initialization function cdev\_add() function is called , by calling this the device driver is registered to the kernel.

Then there are other function like timer handlers, interrupt service routines.

Other than the methods the device drivers should have **structures** too. These are :

* Driver specific data structure > used by the device driver itself
* cdev structure > used by the linux kernel
* file\_operations structure > used by the user space application

**device driver = kernel module**

Now let’s talk about what should go in the initialization function.

TODO: make slides of the device driver design thesis paper

**Chapter 5: Host development environment setup**

We need

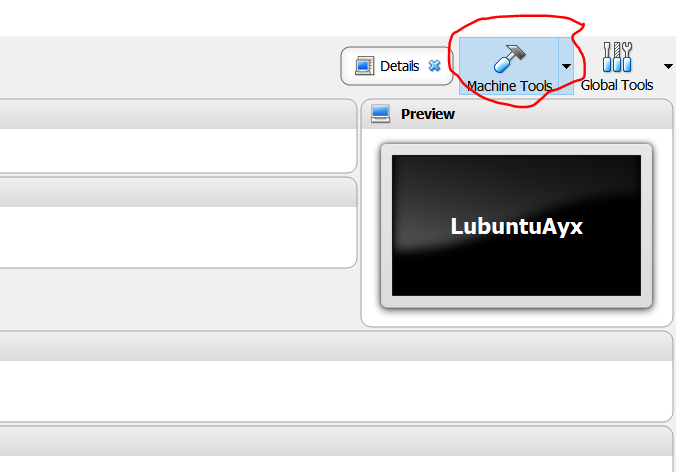
1. java installed
2. Eclipse installed C/C++
3. Target embedded Linux platform’s toolchain [varies according to processor architecture]
4. Source Code of the linux kernel

<https://www.embedded.com/electronics-blogs/open-mike/4420567/Learning-Linux-for-embedded-systems>

* BusyBox is a shell ,  packages about 200 commands into a single executable program
* Hidden file in linux starts with . (dot)
* Other operating systems identify executables by a suffix, like .exe. Linux executables generally do not have a suffix. Instead a file system flag indicates that a file can be executed.
* Most libraries or programs are built using the GNU *make* utility, along with bash scripts or support utilities like *automake* and *autoconf*.
* *make* checks which files need to be compiled and manages (using a Makefile written by the developer) the order in which these compilations are performed. Makefile invokes *make* to build subdirectories or invoking itself recursively.
* *Automake* is designed to generate Makefiles, identifying dependencies and invoking *libtool*,
* Libtool is a utility to create shared libraries.
* *Autoconf* allows libraries or programs to be compiled for different targets and operating systems or with different options.

…………………………………………………………………………………………………………………………………..

Now we will build a kernel in VM , but first take a snapshot of it after shutting it off



Ubuntu = Lubuntu

So we will download Ubuntu sources

$ git clone git://kernel.ubuntu.com/ubuntu/ubuntu-precise.git

* The uname command reports basic information about a computer’s software and hardware
* **BASH** is the most common shell in linux
* Shell accepts human readable commands and translates them into something that can read and process
* **PWD**: print working directory

<http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/>

A loadable kernel module (LKM) is a mechanism for adding code to, or removing code from, the Linux kernel at run time. They are ideal for device drivers, enabling the kernel to communicate with the hardware without it having to know how the hardware works.

The downside of LKMs is that driver files have to be maintained for each device. LKMs are loaded at run time, but they do not execute in user space — they are essentially part of the kernel.

Let’s make a LKM for the VM

ayx@ayx-VirtualBox:~$ sudo apt-get update

Now find out the linux header

ayx@ayx-VirtualBox:~$ apt-cache search linux-headers-$(uname -r)

Install the Linux header

ayx@ayx-VirtualBox:~$ sudo apt-get install linux-headers-4.13.0-32-generic

LKM is not a typical computer program. Important characteristics of LKM are:

* There is no main() function .
* *Do not execute sequentially* : LKM registers itself to the kernel by initialization function

It then handles requests

* *Do not have automatic cleanup:* any dynamic allocation must be manually released
* LKM lives and runs in kernel space
* LKM uses printk()
* LKM can be used by several programs / processes , so multiple access consideration should be accounted while designing the LKM
* More CPU cycles are allocated to LKM
* You should avoid using global variables in kernel , if needed use the static keyword to restrict the variable’s scope within the module

ayx@ayx-VirtualBox:~$ mkdir lkm

download the hello.c and makefile from here <https://github.com/derekmolloy/exploringBB/tree/master/extras/kernel/hello> and put it in the lkm folder

in the Makefile, The –C option switches the directory to the kernel directory before performing any make tasks. The M=$(PWD) variable assignment tells the **make** command where the actual project files exist.

ayx@ayx-VirtualBox:~/lkm$ make

ayx@ayx-VirtualBox:~/lkm$ sudo insmod simple\_km.ko

ayx@ayx-VirtualBox:~/lkm$ lsmod

ayx@ayx-VirtualBox:~/lkm$ modinfo simple\_km.ko

ayx@ayx-VirtualBox:~/lkm$ sudo rmmod simple\_km.ko

Open a new terminal

ayx@ayx-VirtualBox:~$ sudo su –

root@ayx-VirtualBox:~# cd /var/log

root@ayx-VirtualBox:/var/log# tail -f kern.log

You will see the printk() output

A character device typically transfers data to and from a user application — they behave like pipes. They provide the framework for many typical drivers, such as those that are required for interfacing to serial communications, video capture, and audio devices

character driver that can be used to pass information between a Linux user-space program and a loadable kernel module (LKM), which is running in Linux kernel space.

Device drivers are listed at /dev and has a major number associated with it. Major number is unique for that device driver

------------------------------------------------------------------------------------------------------------------------

**Chapter: Beagle Bone Black setup**

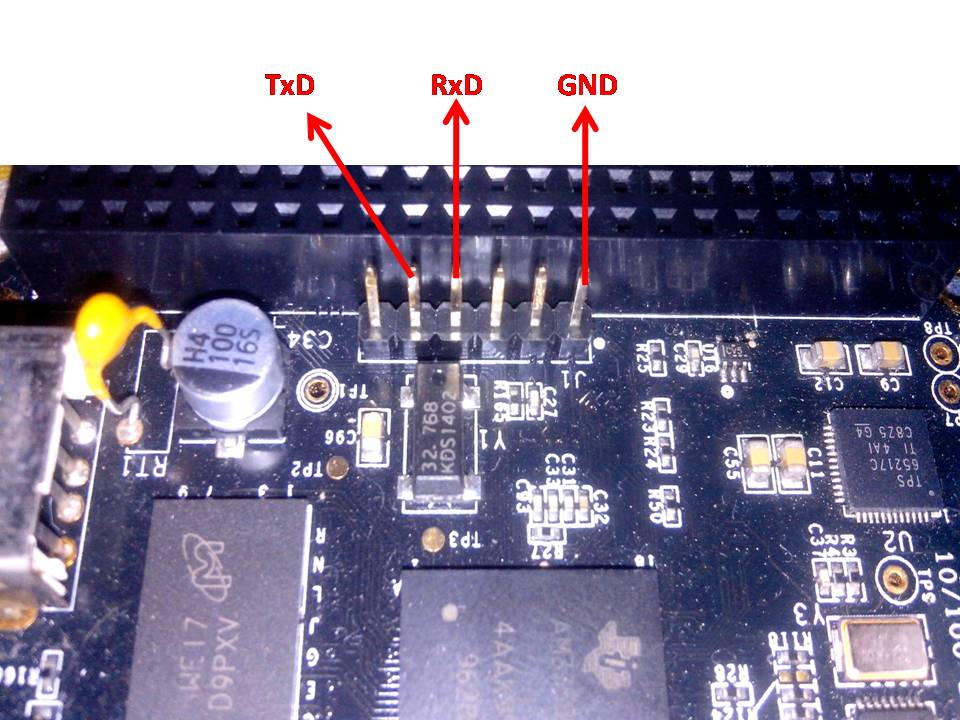
<https://beagleboard.org/latest-images> download the latest debian iot image

Upload to the SD card using ethcer

Holding down the power button insert Power with the 5V 2A adapter

Boot off the bbb from SD. [there is an option to write the image of SD to the onboard eMMC]

Use putty to connect to the bbb over serial



Login: root

Pass: root

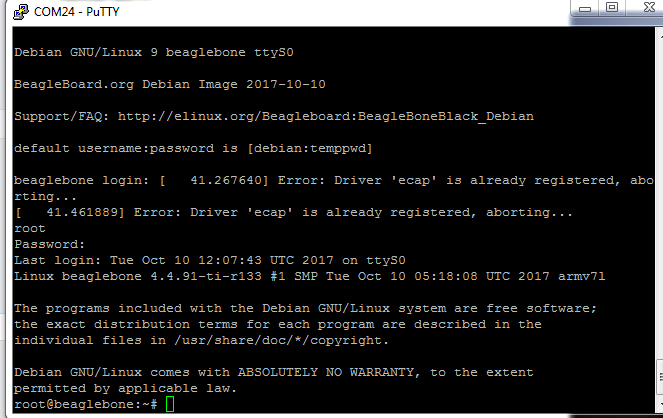


Image : bone-debian-9.2-iot-armhf-2017-10-10-4gb.img

root@beaglebone:~# ifconfig

Notice the ip that the bbb got from network

Shutdown the board

root@beaglebone:~# shutdown –h now

Or press the shutdown button once if not then by press the power button for 8 sec

Insert Ethernet cable to the bbb and connect to your network

Now use putty to connect to the ip

Login as debian because root over ssh is by default disabled for security purpose

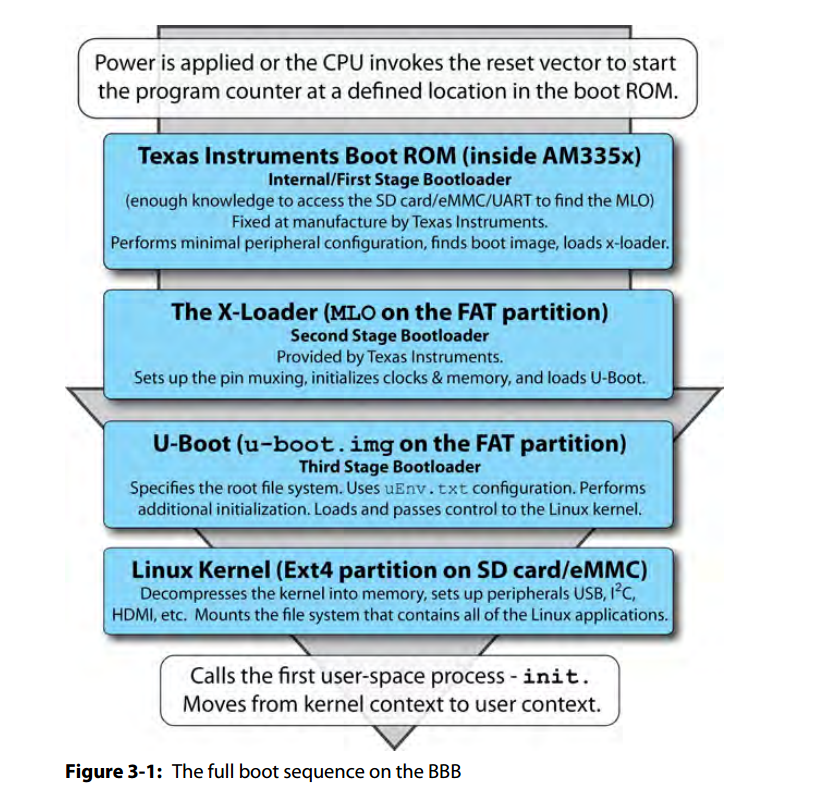
Pass: temppwd

root@beaglebone:/sys/class/leds/beaglebone:green:usr3# echo 1 > brightness

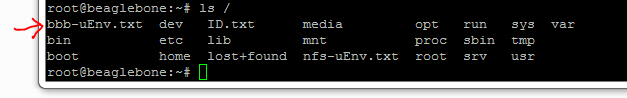
root@beaglebone:/sys/class/leds/beaglebone:green:usr3# echo 0 > brightness

if you insert usb then you will see a FAT partition named “Beagle Bone Getting Started”

the boot sequence of bbb is



The uEnv.txt file is in the sd card



Bash is a scripting language. You can use the Linux commands in the bash script

In Linux exit value of 2 indicates incorrect usage

Success is indicated by 0

*gcc* and *g++* compilers, which are installed by default.

<https://stackoverflow.com/questions/3024197/what-does-int-argc-char-argv-mean>

---------------------------------------------------------------------------------------------------------------------------

Now cloned <https://github.com/derekmolloy/exploringBB.git> in the bbb

Tested makeLED of chapter 5

The Linux GNU C Library, *glibc*, provides an extensive set of wrapper functions  
for system calls. It includes functionality for handling files, signals, mathematics,  
processes, users, and much, much more. The syscall function performs a general system call using the argument passed to it. The argument is called system call number and it varies according to processor architecture. For example

you can call functions like chmod (see /chp05/syscall/callchmod.cpp) by passing the required properties as second and third arguments: int ret = syscall(SYS\_chmod, "test.txt", 0777);

memory mapped GPIO access is also possible but it will bypass the linux kernel

BBB GPIOs are 3.3V tolerant and

* Source current is 4-6 mA
* Sink current 8 mA

in old linux kernel the device tree was in the main kernel . But as there are many manufacturers who use ARM processor and design soc around it so as result There are many variations per one ARM processor , for this variations a single device tree file in the kernel was always growing . for example the board-am335xev.c file got 4000 line long . this increase in size angered linus tovalds and he changed the way device tree is managed in linux. So for the latest linux kernel now you have to add the device tree file

to **/boot/uboot/dtbs** directory

in the device tree file (.dtb) actually the function of a pin is defined . For example say a SOC has a pin which can be GPIO , UART or CAN . now in the device tree you can choose only one option for this pin and it will static . SO the process is

1. you make a device tree for example am335-bone.dtsi
2. am335-bone.dtsi.dtsi > compile > am335-bone.dtsi.dtb
3. put it in the **/boot/uboot/dtbs** directory
4. reboot

So as a pin’s functionality is static so it means it cannot be change dynamically . in order to solve this DTO / device tree overlay is used

PThread = POSIX thread

Usleep() : thread manager can switch to the main thread

Linux poll : a file can be monitored

Poll by the kernel runs with a 0% CPU load

**Cross compilation using eclipse**

<http://exploringbeaglebone.com/chapter7/> [debian]

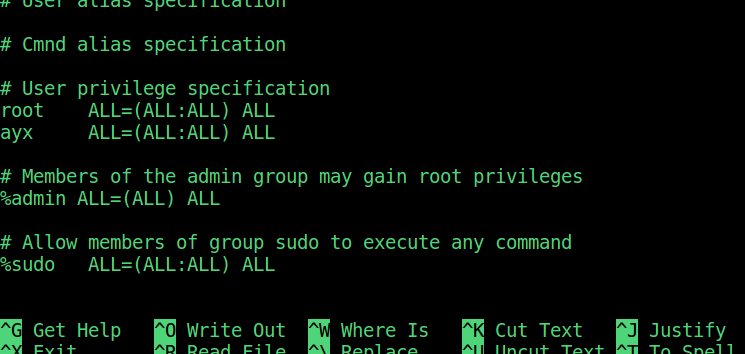
<https://www.youtube.com/watch?v=T9yFyWsyyGk> [debian]

<http://blog.embeddedcoding.com/2015/05/beaglebone-black-development-part-6.html> [eclipse – debian ]

ayx@ayx-VirtualBox:~$ sudo su –

root@ayx-VirtualBox:~# nano /etc/sudoers

Now add your user account the privilege like the root



root@ayx-VirtualBox:~# more /etc/sudoers|grep ayx //check the privilege with this cmd

root@ayx-VirtualBox:~# exit

ayx@ayx-VirtualBox:~$ cd /etc/apt/sources.list.d/

ayx@ayx-VirtualBox:/etc/apt/sources.list.d$ sudo nano crosstools.list

Add this to the file and save “deb http://emdebian.org/tools/debian jessie main”

ayx@ayx-VirtualBox:/etc/apt/sources.list.d$ sudo apt-get install curl

ayx@ayx-VirtualBox:/etc/apt/sources.list.d$ sudo apt-get install build-essential

ayx@ayx-VirtualBox:/etc/apt/sources.list.d$ curl http://emdebian.org/tools/debian/emdebian-toolchain-archive.key | sudo apt-key add -

ayx@ayx-VirtualBox:/etc/apt/sources.list.d$ cd ~

ayx@ayx-VirtualBox:~$ sudo dpkg --add-architecture armhf

ayx@ayx-VirtualBox:~$ dpkg --print-architecture

ayx@ayx-VirtualBox:~$ dpkg --print-foreign-architectures

Some errors may be due to Ubuntu debian mismatch but this does not matter because Starting with Ubuntu 14.04, cross toolchain packages are provided on amd64 and i386 targeting armel, armhf, arm64, ppc64el and powerpc; on ppc64el a cross toolchain is provided for powerpc, on arm64 a cross toolchain is provided for armhf. Ref:<https://wiki.ubuntu.com/ToolChain#Cross_development_toolchain>

So you don’t need to process the commands and they are marked by black cross line

ayx@ayx-VirtualBox:~$ sudo apt-get install crossbuild-essential-armhf

ayx@ayx-LubuntuVm:~/bbb$ sudo apt install g++-arm-linux-gnueabihf

ayx@ayx-VirtualBox:~$ nano test.cpp

Now write a simple print out program

ayx@ayx-VirtualBox:~$ arm-linux-gnueabihf-g++ test.cpp -o test

Now transfer it to your bbb

ayx@ayx-VirtualBox:~$ sftp [debian@192.168.11.233](mailto:debian@192.168.11.233)

sftp> put test

now go to bbb and login as debian

debian@beaglebone:~$ ./test

you will see hello from ARMHF architecture printed

So cross compilation is tested

<https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/v2v_guide/preparation_before_the_p2v_migration-enable_root_login_over_ssh> follow this step to enable root access over ssh. It is not recommended but doing it for ease of access to the bbb

ayx@ayx-LubuntuVm:~$ sudo apt-get install libicu-dev:armhf

libiu armhf installation failed in lubuntu

ayx@ayx-LubuntuVm:/usr/lib$ sudo apt-get install qemu-user-static

ayx@ayx-LubuntuVm:~/bbb$ ./test

/lib/ld-linux-armhf.so.3: No such file or directory

-----------------------------------------------------------------------------------------------------------------

ayx@ayx-LubuntuVm:~$ wget -c <https://releases.linaro.org/components/toolchain/binaries/6.4-2017.11/arm-linux-gnueabihf/gcc-linaro-6.4.1-2017.11-x86_64_arm-linux-gnueabihf.tar.xz>

ayx@ayx-LubuntuVm:~$ export CC=`pwd`/gcc-linaro-6.4.1-2017.11-x86\_64\_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-

Still errors

-----------------------------------------------------------------------------------------------------------------

<https://www.debian.org/blends/hamradio/get/live> downloading debian jessie

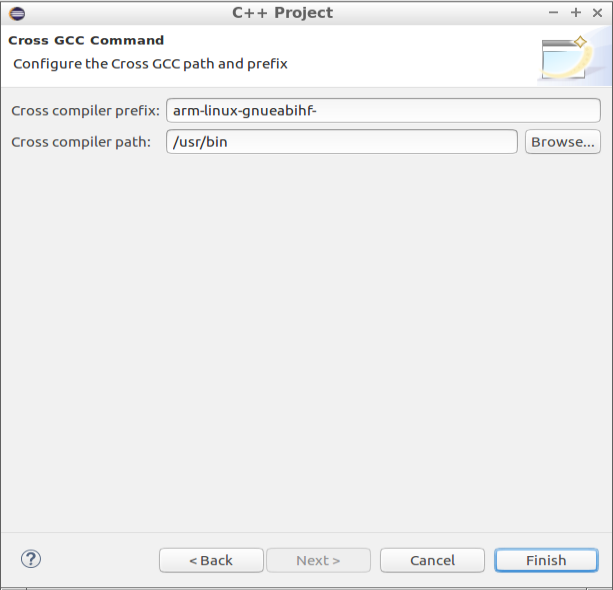
-----------------------------------------------------------------------------------------------------------------

Uninstalled java 9 , removed eclipse

Now reinstalling java 8 and eclipse following esp8266 environment setup guide

**New project setup**

#eclipse > File > new > other > C/C++ > C++ project > create new C++ project > cross gcc, hello world C++ project > put your project name

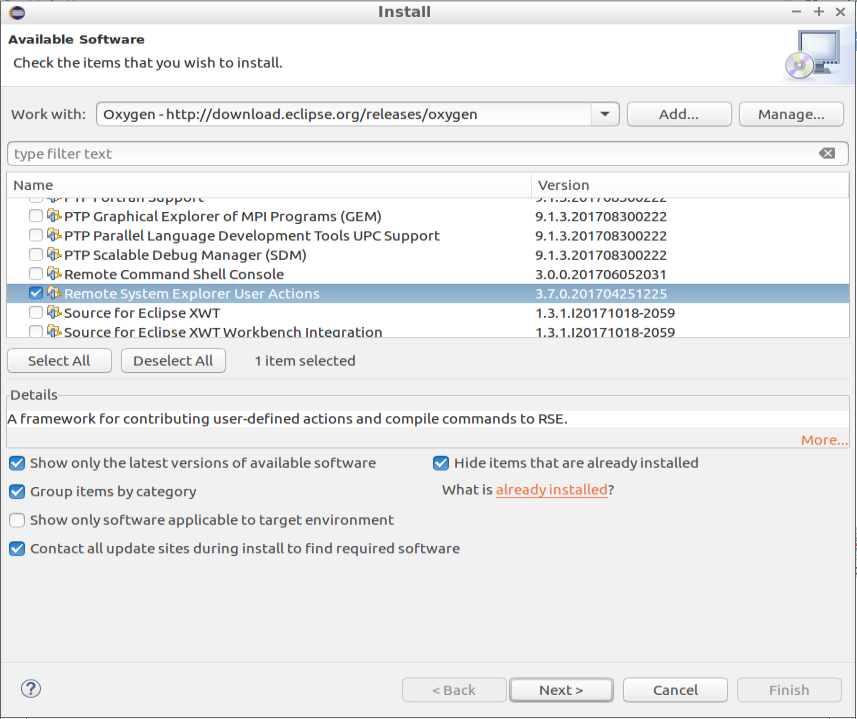


Now write your code e.g. the simple hello world ,

Project > build

**RSE: remote system explorer [from 21:00 video]**

Help > install new software > select work with oxygen [eclipse edition]



Window >show view > other >remote systems > remote systems > open

Select the RSE tab > click the New connection icon > linux type > next > BBB ip address > connection name : beaglebone > next

Files: ssh files > process : process.shell.linux > shells : ssh.shells > next

Right clink on begalebone > connect

Now you can copy application from your build directory and put it in the beaglebone by RSE

Using remote shell you can run the application too but Better option is to automate the process so that when you build the project the binary output will be automatically placed in the remote system build directory

Go to the eclipse folder & generate a key with no passphrase

ayx@ayx-LubuntuVm:/opt/eclipse$ ssh-keygen

Now copy the id to the bbb

ayx@ayx-LubuntuVm:/opt/eclipse$ ssh-copy-id [root@192.168.0.103](mailto:root@192.168.0.103)

Now try to login by ssh

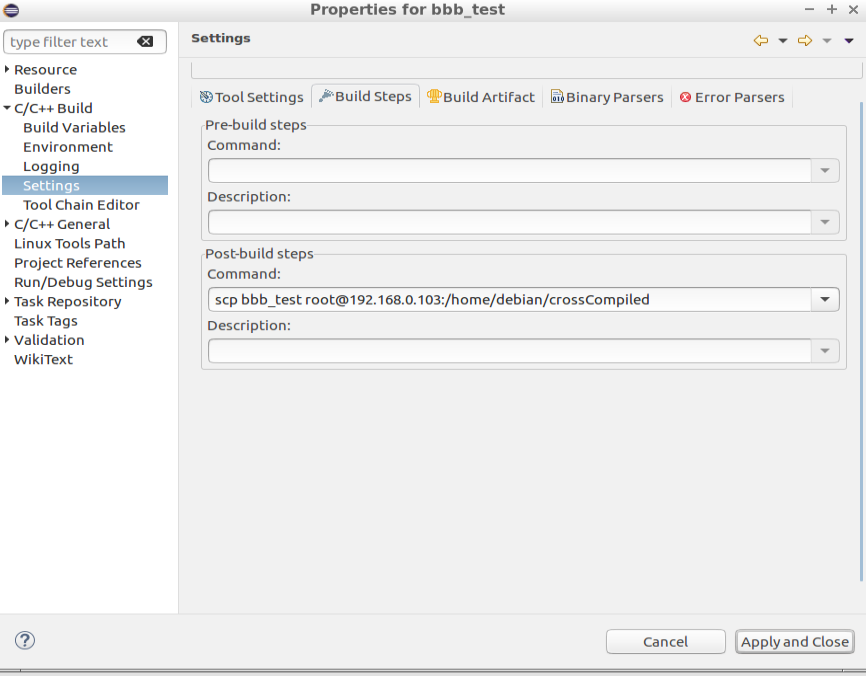
ayx@ayx-LubuntuVm:/opt/eclipse$ ssh [root@192.168.0.103](mailto:root@192.168.0.103)

This is done so that we can copy the build output to the bb without any passphrase

Now lets setup things for the automatic copy

Choose the project tab if you are in RSE

#eclipse: Projects > Properties > C/C++ build > settings > build step >



Now build and if bbb is connected then the build output will be automatically copied to the bbb crossCompiled folder , now by using Remote shell you can run the app

**GDB integration and use [from 31:00 video ]**

In this case the bbb will play the role as a gdbserver so install it in bbb

root@beaglebone:~$ sudo apt-get install gdbserver

Now return back to the local desktop machine

ayx@ayx-LubuntuVm:/opt/eclipse$ cd ~

ayx@ayx-LubuntuVm:~$ sudo apt-get install gdb-multiarch

Now create the gdbinit file into your workspace directory

ayx@ayx-LubuntuVm:~/eclipse-workspace-embeddedLinux$ nano .gdbinit

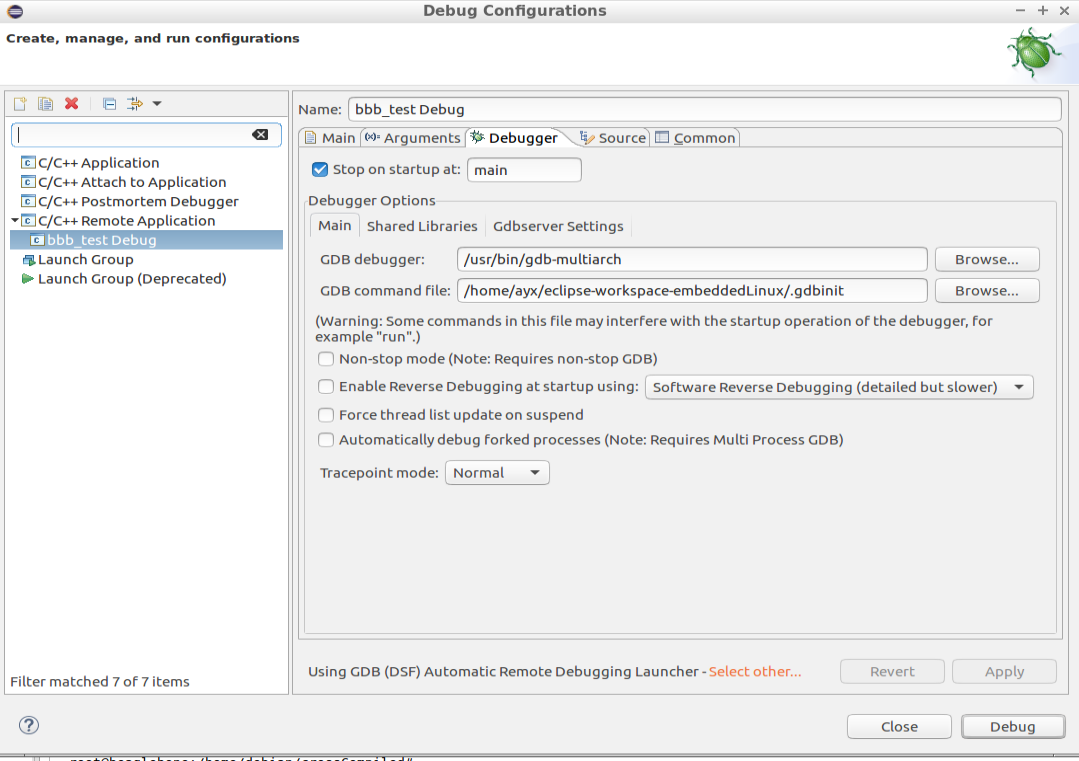
Put “set architecture ARM” and save

Check the file by

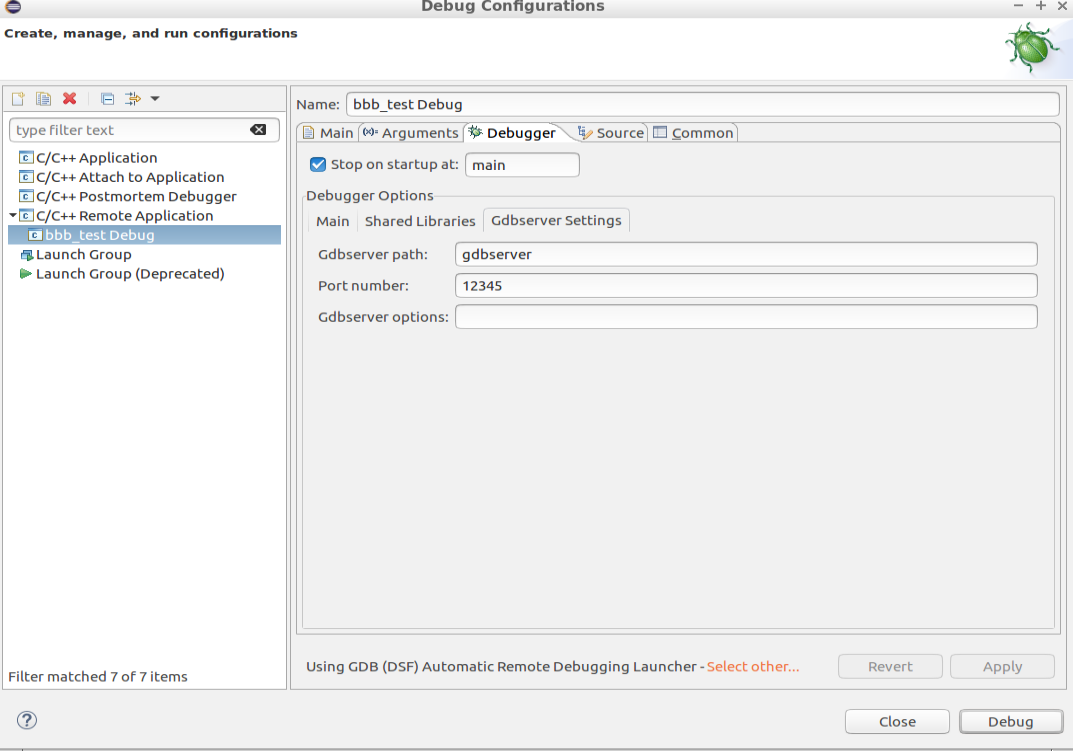
ayx@ayx-LubuntuVm:~/eclipse-workspace-embeddedLinux$ more .gdbinit

Now return to eclipse

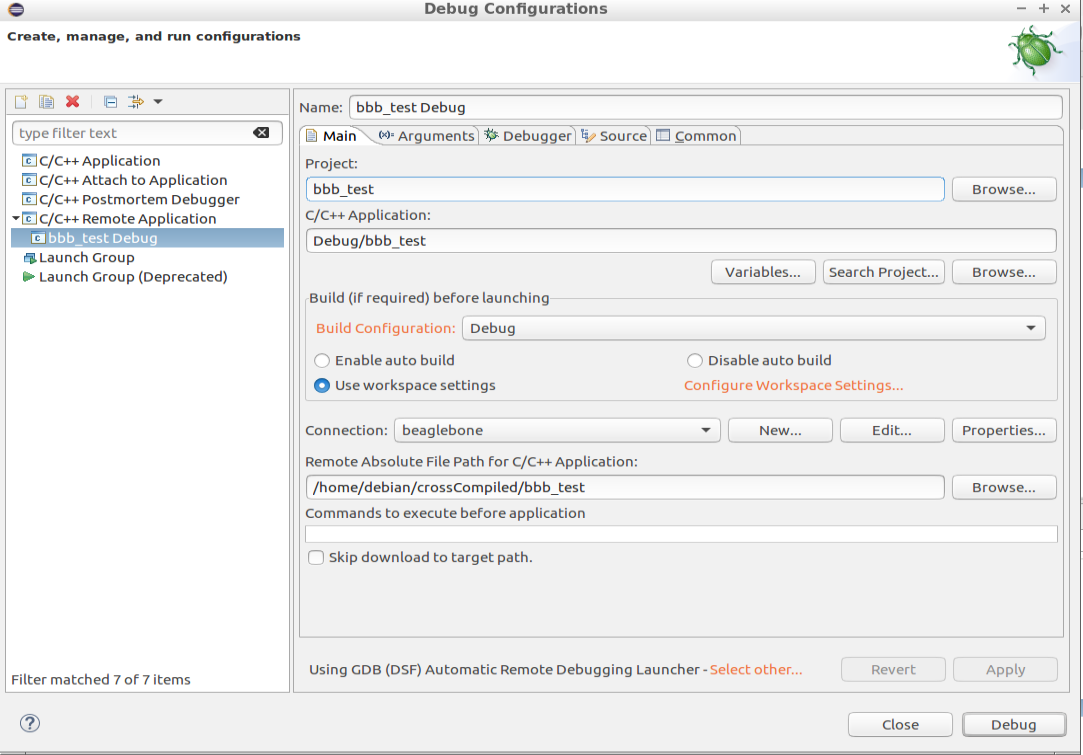
#eclipse: Run > Debug configurations > C/C++ applications: delete the application if any > C/C++ Remote Application right click new > Debugger > select the workspace directory gdbinit file > rename the gdb debugger command



Then go to gdbserver settings



Now go to main



Now apply and debug > accept the debug perspective

Set breakpoint > step into statement

**Github integration:**

As I am following derek mollyes bbb book and codes I will fork his github repo and work on that.

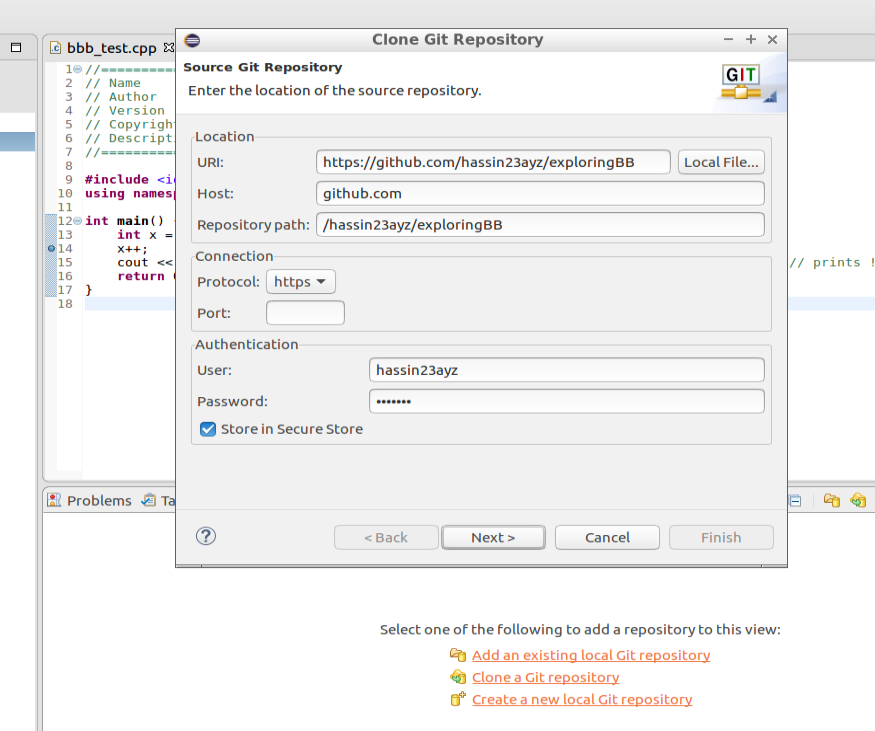
<https://github.com/DerekMolloy/ExploringBB> forked the repo in my github account

#eclipse : Help > install new software > work with Oxygen > search Git

Mark Git integration for eclipse – Task focused interface > next > select, next > accept > finish

Restart

Window > view > other > git > git repository > clone a git repository > put the uri > next



Now let’s open a project from the repo in eclipse and work on it

#eclipse >

----------------------------------------------------------------------------------------------------------------------------

Download the latest stable kernel from <https://www.kernel.org/>

Extract It

Take a moment to browse the folders

**Makefile.** Use to control configuring and building the kernel  
**Arch.**Contains architecture specific files  
**Drivers.** Sources for drivers included with the kernel  
**Include.** Headers used to build the kernel and drivers  
**Kernel.**The “core” target-independent parts of the kernel  
**README.** A description of the directory, with make directives  
**Scripts.** Bash scripts used to build the kernel and drivers

Now copy the folder to the home directory

$ mkdir ~/linux

$ cp -r linux-4.14.15 ~/linux

$ make oldconfig //will ask configuration options in terminal

$ make menuconfig //will ask configuration in ui

$ gedit Makefile

Rename Name = your name

$ make clean

$ make