GPIO DEVICE DRIVER DEVELOPMENT ON BEAGLEBONE BLACK

[1]Daksh Joshi & [2]Jignesh Patoliya
[1]Student, [2]Assistant Professor
[1]Department of Electronics and Communication
[1]Charotar University of Science and Technology, Anand, India

Received: February 14, 2019 Accepted: March 20, 2019

ABSTRACT: *Contemporary era has become the age of information and high performance computing and an open-source platform like linux becomes quite handy to fulfil the requirements with minimum memory consumption. Today, most of the gadgets used for information sharing, (e.g, multimedia projector, wifi router, infotainment system in automobiles, etc.) run on embedded linux. In an embedded system, mpu (micro-processor unit) requires communicating with several devices for which, the kernel requires device drivers. With raise in number to different types of devices, distributed embedded system has become one of the hot-topics in the tech industry. Thus, an engineer working on BSP (Board Support Packages) has plenty of opportunities both in the academia and the industry. With the fast-growing advancements in the fields of embedded electronics and distributed embedded systems, embedded linux has become a primary requirement for the industry to work on.*

Key Words: *Embedded Linux; Embedded Systems; Linux Device Drivers; Linux Kernel Compilation; BeagleBone Black*

INTRODUCTION

Embedded linux based devices have become ubiquitous and essential for the exchange of information. Embedded systems running on embedded linux platform require interfacing with other devices and in order to do that the kernel, require device drivers to communicate with the hardware. However, optimising the kernel, as by maintaining its size through adding driver of the required device interfaces only in order to maintain high performance of the system has become an area of crucial interest for developers. Embedded linux and gpio device driver development is a project that aims of giving access of the general purpose input/output pins of BeagleBone Black to user space for use by a device file. The project gives a fair exposure to the concepts involved in the embedded linux development such as; toolchain, bootloader, kernel and system memory. Decision of embedded linux development on BeagleBone Black has its own justification in form of it being a low-cost high-performance single board computer running on ARM architecture based processor. Moreover, BeagleBone Black offers 60 general purpose input/output pins which enhance its versatility in context to user applications making it one of the favorable boards for embedded linux in the industry [8]. In order to optimize the use of these 60 general purpose input/output pins, a gpio device driver is developed so that, the access of the gpios can be accessed by user for various applications. However, this driver enables user to access only one function from many of the general purpose input/output pins which is, input/output functionality only.

EMBEDDED LINUX DEVELOPMENT FOR BEAGLEBONE BLACK

2.1 BeagleBone Black

The BeagleBone Black is the newest member of the BeagleBoard family. It is a lower-cost, high-expansion focused BeagleBoard using a low cost Sitara XAM3359AZCZ100 Cortex A8 ARM processor from Texas Instruments [2][8]. It is similar to the BeagleBone, but with some features removed and some features added.

2.2 Embedded Linux System Architecture

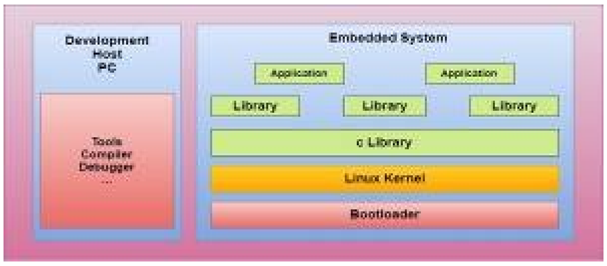


Fig. 2.2.1: Embedded Linux System Architecture

Fig. 2.2.1 describes the embedded linux system architecture which requires a host for building an linux based embedded system. In embedded linux system development, development host is the workplace where the required files are cross-compiled for the targeted embedded system. The software components required for embedded system development are;

1. Cross-compilation Toolchain: A cross-compiler is capable of generating executable file for a different platform than the host [2] [6] [7]. Toolchain provides cross-compilation support for several architectures. Choosing right toolchain plays a crucial role in building a kernel. Below mentioned Fig. 2.2.2 demonstrates types of cross-compilations. However, using the latest version of a toolchain is not always preferable as, some of the latest versions do not build the kernel properly [3]. The cross-compilation toolchain used for this project is:

“gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnueabihf”

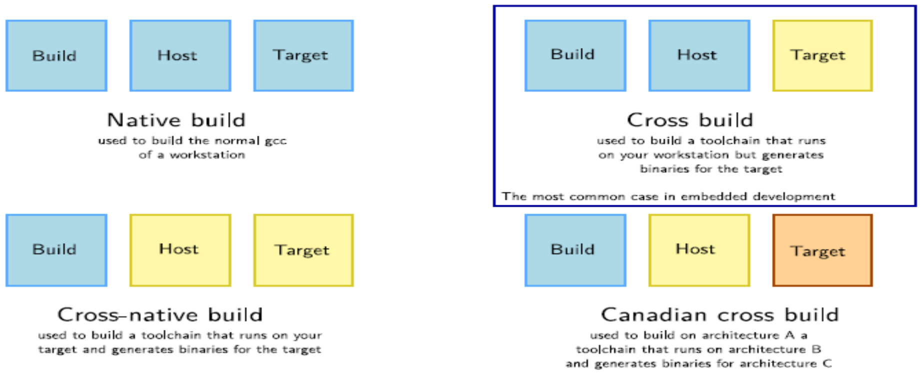

Build
Host
Target
Native build
used to build the normal gcc of a workstation
Build
Host
Target
Cross build
used to build a toolchain that runs on your workstation but generates binaries for the target
The most common case in embedded development
Build
Host
Target
Cross-native build
used to build a toolchain that runs on your target and generates binaries for the target
Build
Host
Target
Canadian cross build
used to build on architecture A a toolchain that runs on architecture B and generates binaries for architecture C

Fig. 2.2.2: Cross-compilation Types [2]

2. Bootloader: It is the first section of code that executes after the system is powered ON or reset on any platform. The bootloader places the OS of a computer into memory. There are various bootloader available for specific architectures [6] [7]. The bootloader used in this project is u-boot as BeagleBone Black SoC is based on arm architecture. Booting sequence of BeagleBone Black needs to be understood thoroughly in order to develop an embedded linux system on BeagleBone Black. Fig. 2.2.3 shows the BeagleBone Black booting sequence.

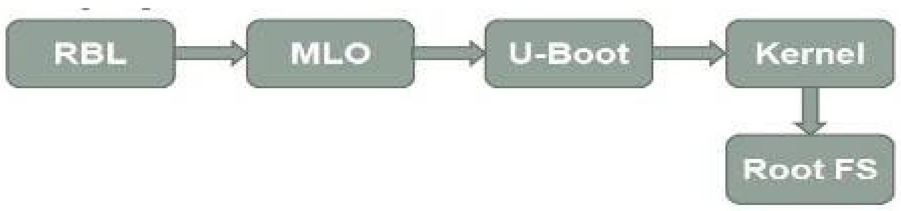


Fig. 2.2.3: BeagleBone Black Booting Sequence

RBL (ROM Bootloader): When the system is first booted, the CPU invokes the reset vector to start the code at a known location in ROM. This code performs minimal clocks, memories and peripheral configurations. Moreover, it searches the booting devices for a valid booting image and loads the MLO into SRAM and executes it.
MLO (Memory Loader): This part of code sets up the pin multiplexing, initializes clocks and memory and loads the u-boot image to the DDR memory and executes it.
U-boot: This part of code performs some additional platform initialization, sets the boot arguments and passes control to the kernel image.
Kernel: It decompresses the kernel into SDRAM, sets up peripherals and mounts the linux file-system (ext-4 for BeagleBone Black)

3. Linux Kernel: The sole aim of kernel is to manage the communication between the software and the hardware. The main tasks of kernel include; process management, device management, memory management and interrupt handling, input/output communication [1] [3]. Kernel source used in this project is “linux kernel v4.2”

4. Root File-system: The root file-system contains many system-specific configuration files. Possible examples include a kernel that is specific to the system, a specific hostname, etc. This means that the root file-system isn't always shareable between networked systems [4]. Keeping it small on servers in networked systems minimizes the amount of lost space for areas of un-shareable files. It also allows workstations with smaller local hard drives [1]. The root file-systems for BeagleBone Black are available on the internet. However, a custom root file-system can be build using tools like "Buildroot" [6].

2.3 Steps for Embedded Linux Development for BeagleBone Black:

2.3.1 Requirements:

Host running on Ubuntu 16.04 (64-bit), Active internet connection, Ethernet cable, Power Cable for BeagleBone Black and UART bridge converter.

2.3.2 Procedure:

Step 1: TFTP Setup:

In order to enable serial booting, tftp service needs to be installed. Start the tftpd-hpa service automatically by adding a command to "/etc/rc.local" file; [2] [6] [7].

"service tftpd-hpa start" (Excluding Quoates)

Save and close the file.

Now, add the tftp directory in config file. Edit following in the file:

TFTP_DIRECTORY="/srv/tftp"

Save and close the file.

Create a tftp directory in "/srv" directory. Finally, restart the tftp service

Step2: NFS Setup

Install nfs server. Create an nfs directory in "/srv" directory. Give proper permissions. Now, to grant Read-Write access to all ip addresses, open nfs export file and enter following in the file: [2] [6] [7]

"/srv/nfs*(rw,no_root_squash,no_subtree_check,sync)" (Excluding Quotes)

Save and close the file.

Now, tell nfs to read the new export file and restart necessary services.

Step 3: Toolchain Setup

Create a directory named "BeagleBone" at desired locaton. Now, download and install appropriate toolchain from the internet using command and add toolchain path in default path-variable [2] [3] [6] [7].

Step 4: Kernel Compilation

Download the linux kernel version v4.2 from the internet and save in the directory named "BeagleBone". Un-tar the downloaded kernel source. Clean the kernel directory and download "bbb_3.8.13_2015.config" file from internet and save it in the "BeagleBone" directory. Copy the kernel config file. ConFig. kernel with this config file [1] [7]. Start the kernel cross-compilation. Fig. 2.3.1

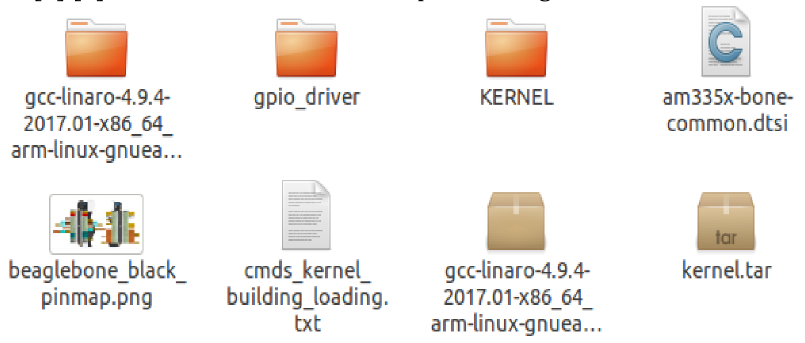

gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnuea...
gpio_driver
KERNEL
am335x-bone-common.dtsi
beaglebone_black_pinmap.png
cmds_kernel_building_loading.txt
gcc-linaro-4.9.4-2017.01-x86_64_arm-linux-gnuea...
kernel.tar

Fig. 2.3.1: "KERNEL" directory after extracting the ".tar" file

Step 5: Prepare tftp Root Directory

Download the "rfs_bb_static_1.23.2" root directory from internet, save it in the "BeagleBone" directory and un-tar it. Clean the tftp directory and copy the zImage and dtb files. Also, copy the downloaded rootfs to the nfs directory.

Step 6: Test the setup

Connect BeagleBone with computer over LAN cable (shown in Fig. 2.3.2), connect beaglebone with USB-UART module and connect USB-UART module with computer (shown in Fig. 2.3.3). Install and start minicom and perform following tasks: [2]

1. Power on the Beaglebone Black using the data cable

2. Check the minicom and once you see the u-boot logs, press space within 2-3 seconds to enter into u-boot prompt. Minicom window after entering u-boot prompt is shown in Fig. 2.3.4.

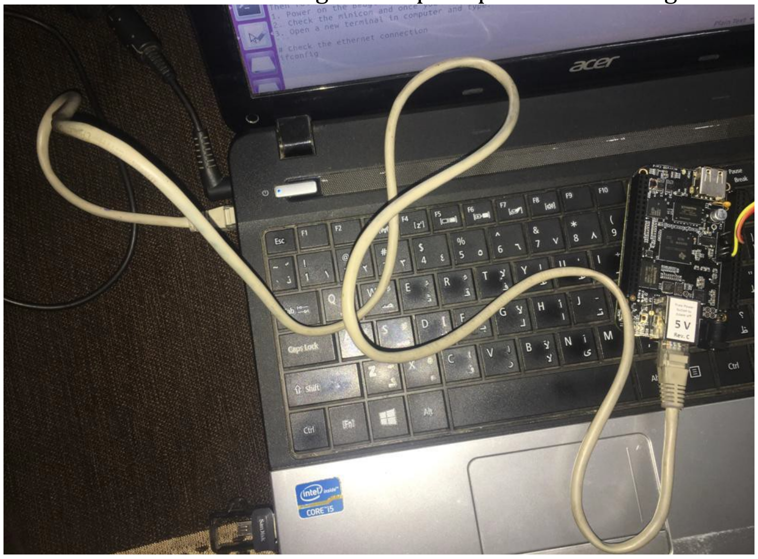


Fig. 2.3.2: Connecting BeagleBone Black over LAN cabel

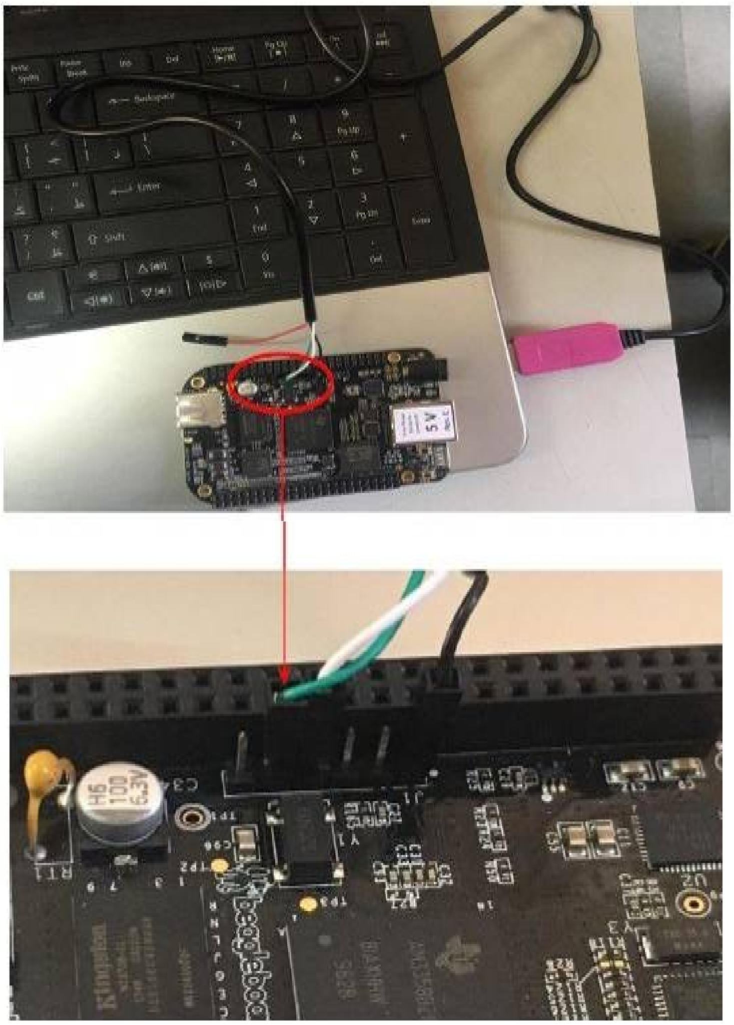
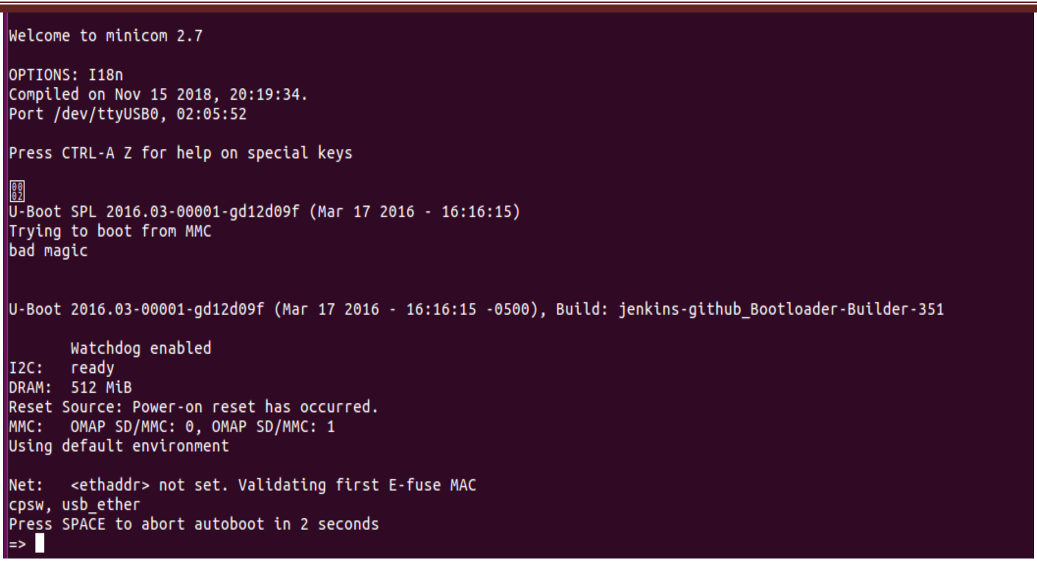


Fig. 2.3.3: Connecting BeagleBone Black with USB_UART bridge module



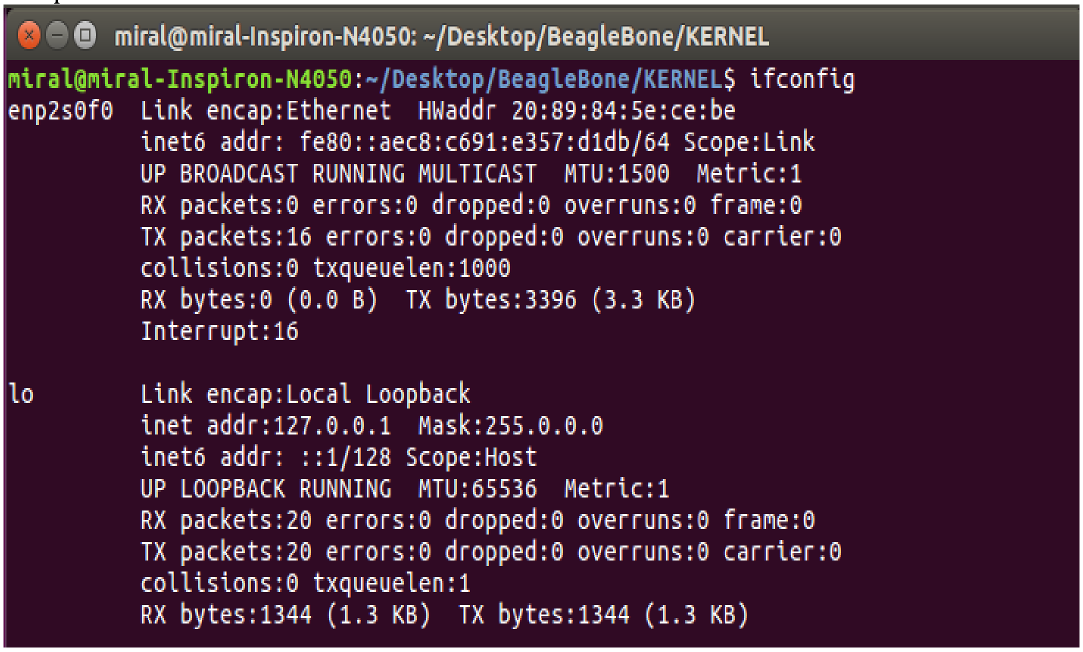
```
Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Nov 15 2018, 20:19:34.
Port /dev/ttyUSB0, 02:05:52

Press CTRL-A Z for help on special keys


U-Boot SPL 2016.03-00001-gd12d09f (Mar 17 2016 - 16:16:15)
Trying to boot from MMC
bad magic


U-Boot 2016.03-00001-gd12d09f (Mar 17 2016 - 16:16:15 -0500), Build: jenkins-github_Bootloader-Builder-351

       Watchdog enabled
I2C:   ready
DRAM:  512 MiB
Reset Source: Power-on reset has occurred.
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
Using default environment

Net:   <ethaddr> not set. Validating first E-fuse MAC
cpsw, usb_ether
Press SPACE to abort autoboot in 2 seconds
=>
```

Fig. 2.3.4: Minicom window after entering u-boot prompt

Open the new terminal and check the Ethernet connection using:

$ ifconfig

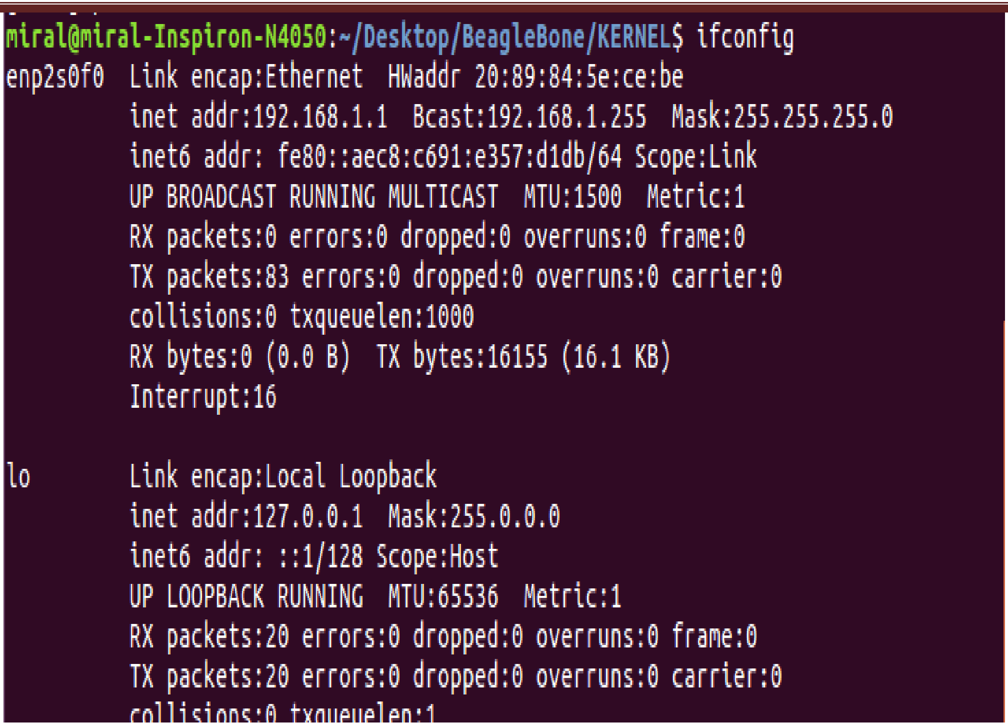
Sample output will be:



```
miral@miral-Inspiron-N4050: ~/Desktop/BeagleBone/KERNEL
miral@miral-Inspiron-N4050:~/Desktop/BeagleBone/KERNEL$ ifconfig
enp2s0f0  Link encap:Ethernet  HWaddr 20:89:84:5e:ce:be
          inet6 addr: fe80::aec8:c691:e357:d1db/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:16 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:3396 (3.3 KB)
          Interrupt:16

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:20 errors:0 dropped:0 overruns:0 frame:0
          TX packets:20 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:1344 (1.3 KB)  TX bytes:1344 (1.3 KB)
```

Fig. 2.3.5: Ehternet name checking

Assign a specific IP address to your Ethernet. Check the Ethernet name in your computer. Here, name of Ethernet is “enp2s0f0” (Specified in Fig. 2.3.5) so the command would be “$ sudo ifconfig enp2s0f0 192.168.1.1” and verify the same using “$ ifconfig”. Note “inet addr: 192.168.1.1” (Specified in Fig. 2.3.6)



```
miral@miral-Inspiron-N4050:~/Desktop/BeagleBone/KERNEL$ ifconfig
enp2s0f0  Link encap:Ethernet  HWaddr 20:89:84:5e:ce:be
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::aec8:c691:e357:d1db/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:83 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:16155 (16.1 KB)
          Interrupt:16

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:20 errors:0 dropped:0 overruns:0 frame:0
          TX packets:20 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
```

Fig. 2.3.6: Specifying "inet addr:"

Step 7: U-boot Commands
Once the fixed IP address is set, follow the Uboot Commands given below to load kernel and root file-system (rootfs). Set server ip(192.1668.1.1) and ip address(192.168.1.2) of device as environment variables. Load zImage and dtb file through tftp at 0x80007fc0 and 0x80f80000 addresses respectively [2] [8]. Now, set up the boot arguments:

```
=>  setenv bootargs 'console=ttyO0,115200 root=/dev/nfs rw ip=192.168.1.
     nfsroot=192.168.1.1:/srv/nfs/rfs_bb_static_1.23.2'
=> setenv bootargs ${bootargs} nfsrootdebug earlyprintk
```

Fire the final boot command:
=>bootz 0x80007fc0 - 0x80f80000
Press "Enter" when asked and we will get the console of BeagleBone use following command to see the root file-system contents:
# ls
Finally the kernel is loaded successfully and thus, denotes the completion of Embedded Linux Development for BeagleBone Black.

2.4 GPIO Device Driver Development:
2.4.1 Linux Device Driver:
Devices in the kernel are block or character devices and are identified using a major and a minor number. Where, the major number denotes the family of device while the minor number denotes the number of device. Device driver is software that handles or manages a hardware controller. The linux kernel device drivers are, essentially, a shared library of privileged, memory resident low-level hardware handling routines [2] [3] [5]. It is linux device driver that handle the particularities of the devices they are managing. One of the fundamental features is that it makes a written summary of the handling of devices. Fig. 2.4.1 shows the typical linux device driver types and how they are invoked.

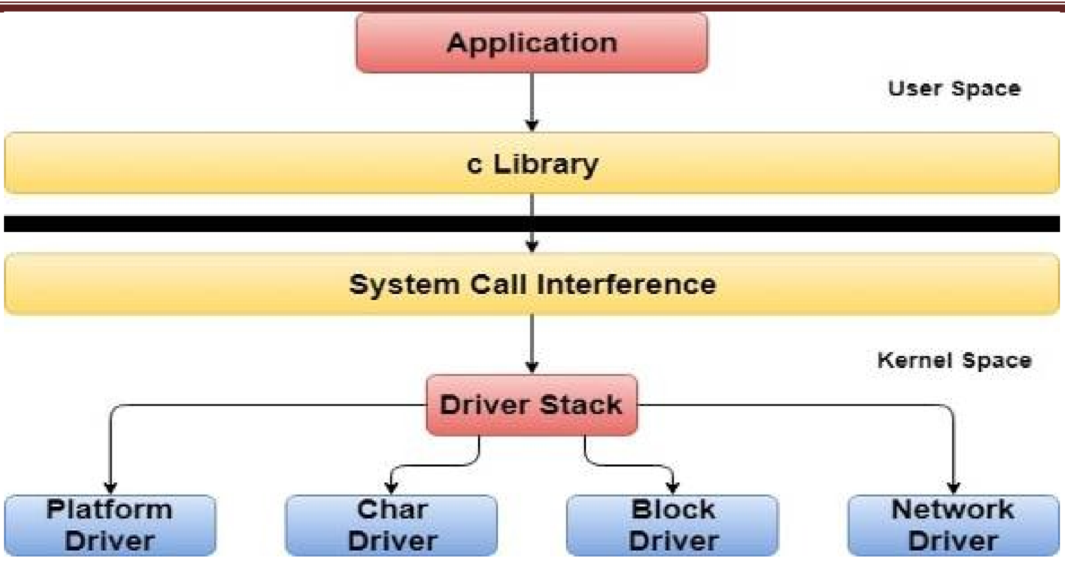


Fig. 2.4.1: Linux Device Driver types and route to their access

2.4.2 GPIO Device Driver:

GPIO device drives falls into the category of Platform Device Driver. For development of a gpio device driver, fair understanding of platform device driver is necessary. Hardware devices may be connected through a bus allowing enumeration, hot plugging, or providing unique identifiers for devices. Platform devices on embedded systems are often not connected through a bus which allows the devices to be uniquely identified [5]. Many devices are directly part of a system on chip: UARTs Ethernet controllers, SPI or I2C controllers, GPIOs, etc. In this case, the devices must be statically described in the kernel source-code or the device tree instead of being dynamically detected. Each device managed by a particular driver typically uses different hardware resources such as interrupts and I/O addresses [3] [5]. For adding platform devices to the platform bus, device tree processing in the kernel is responsible. Thus, a platform driver is a device driver for specific platform device on the platform bus. Platform drivers inherently have no interface to user-space without involving into the kernel framework. In ARM architecture, device tree processing does the enumeration of the platform bus. Platform devices and drivers are described in the kernel tree at: “Documentation/driver-model/platform.txt”.

In order to explicitly enumerate the gpio driver, some tweaking is done in the device tree source include file. Device tree source include file describes the hardware of the certain platform and thus, a platform driver needs to be defined in this file. Below mentioned is the modification to be done into the device tree source include file where the highlighted part is just an indication of the location in the file at which the code is to be edited and the box denotes the lines to be added.

Tabel 2.4.1: Pin Multiplexing Edit

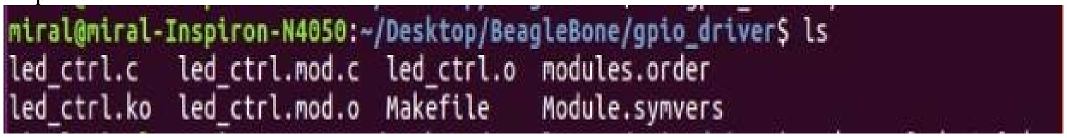
```
///Pin Muxing Edit///
extled_pins: pinmux_extled_pins {
pinctrl-single,pins = <
0x24 0x0f
0x70 0x0f
0x74 0x0f
>;
};
///Pin Muxing Done///
ocp: ocp {
```

Pin multiplexing is done in the code shown in table 2.4.1. In system on chip, each pin has multiple functionalities and thus, defining the desired functionality to be used becomes extremely crucial and primary task [8]. Here, the desired pins on system on chip are expected to perform as data I/O pins.

Tabel 2.4.2: Custom User LEDs Edit

```
/// Custom User LEDs ///

custom-user-leds{
  compatible = "gpio-control-leds";
  pinctrl-names = "default";
  pinctrl-0 = <&extled_pins>;

ext-led0{
            label = "ext-usr0";
            gpios = <&gpio1 23 0>;
            default-state = "off";
        };

ext-led1{
            label = "ext-usr1";
            gpios = <&gpio1 30 0>;
            default-state = "off";
        };

ext-led2{
            label = "ext-usr2";
            gpios = <&gpio1 31 0>;
            default-state = "off";
        };

};

/// Custom User LEDs End ///
```

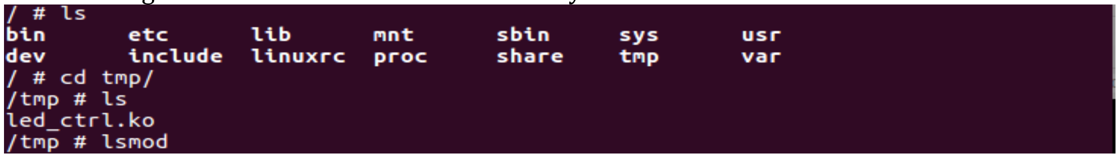
In the code shown in table 2.4.2, 3 LEDs are defined as devices attached to the pins used in previous section of code. Defining the devices is important as, while using the same device through user-space, the commands are given considering the name of the devices and thus, for user-space to identify the entered device name, defining the devices is done.
Now, we need to program the device driver. While programming the gpio device driver, as this is a platform device driver, we need to use functions platform_device_register() and platform_device_unregister() for registering the and deregistering the module. As soon as the platform_device_register() function is read by the kernel, it invokes the probe() function which is responsible for initializing the device, mapping I/O memory, registering the interrupt handlers and registering the device to the kernel framework [2] [5]. The newly written .c file is now cross-compiled for ARM architecture and .ko file is generated (Shown in Fig. 2.4.2) which is the kernel module file that needs to be placed into the "/tmp" directory in root file system saved in the "/srv/nfs" directory. Fig. below shows the led_ctrl.c file and led_ctrl.ko file generated after cross-compilation.



```
miral@miral-Inspiron-N4050:~/Desktop/BeagleBone/gpio_driver$ ls
led_ctrl.c   led_ctrl.mod.c  led_ctrl.o  modules.order
led_ctrl.ko  led_ctrl.mod.o  Makefile    Module.symvers
```

Fig. 2.4.2: Cross-compilation of .c file to generate .ko file for BeagleBone Black

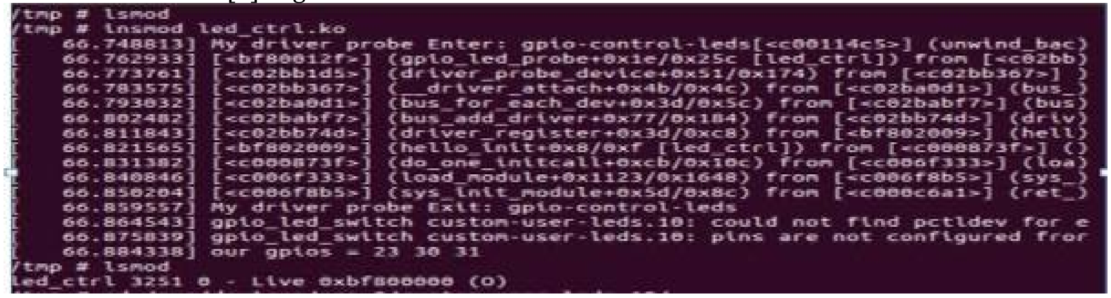
Further, reach the kernel terminal by following the previously discussed steps and check whether led_ctrl.ko exists in the /tmp directory. Fig. 2.4.3 demonstrates the same wherein the "# lsmod" is the command that shows the existing inserted modules from that directory.



```
/ # ls
bin      etc      lib      mnt      sbin     sys      usr
dev      include  linuxrc  proc     share    tmp      var
/ # cd tmp/
/tmp # ls
led_ctrl.ko
/tmp # lsmod
```

Fig. 2.4.3: Checking the contents of /tmp directory

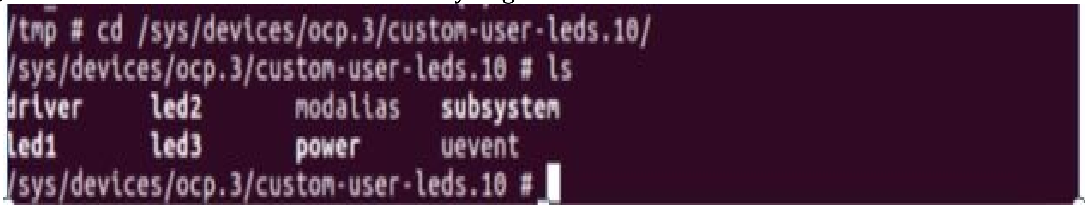
To make the driver accessible by the user, it needs to be inserted into the kernel using command "# insmod". After inserting this module into the kernel, check whether the module is inserted correctly or not by "# insmod" command [2]. Fig. 2.4.4 below shows the same.



```
/tmp # lsmod
/tmp # insmod led_ctrl.ko
[   66.748813] My driver probe Enter: gpio-control-leds[<c00114c5>] (unwind_bac)
[   66.762933] [<bf80012f>] (gpio_led_probe+0x1e/0x25c [led_ctrl]) from [<c02bb)
[   66.773761] [<c02bb1d5>] (driver_probe_device+0x51/0x174) from [<c02bb367>] )
[   66.783575] [<c02bb367>] (__driver_attach+0x4b/0x4c) from [<c02ba0d1>] (bus_)
[   66.793032] [<c02ba0d1>] (bus_for_each_dev+0x3d/0x5c) from [<c02babf7>] (bus)
[   66.802482] [<c02babf7>] (bus_add_driver+0x77/0x184) from [<c02bb74d>] (driv)
[   66.811843] [<c02bb74d>] (driver_register+0x3d/0xc8) from [<bf802009>] (hell)
[   66.821565] [<bf802009>] (hello_init+0x8/0xf [led_ctrl]) from [<c000873f>] ()
[   66.831382] [<c000873f>] (do_one_initcall+0xcb/0x10c) from [<c006f333>] (loa)
[   66.840846] [<c006f333>] (load_module+0x1123/0x1648) from [<c006f8b5>] (sys_)
[   66.850204] [<c006f8b5>] (sys_init_module+0x5d/0x8c) from [<c000c6a1>] (ret_)
[   66.859557] My driver probe Exit: gpio-control-leds
[   66.864543] gpio_led_switch custom-user-leds.10: could not find pctldev for e
[   66.875839] gpio_led_switch custom-user-leds.10: pins are not configured fror
[   66.884338] our gpios = 23 30 31
/tmp # lsmod
led_ctrl 3251 0 - Live 0xbf800000 (O)
```

Fig. 2.4.4: Inserting led_ctrl.ko module

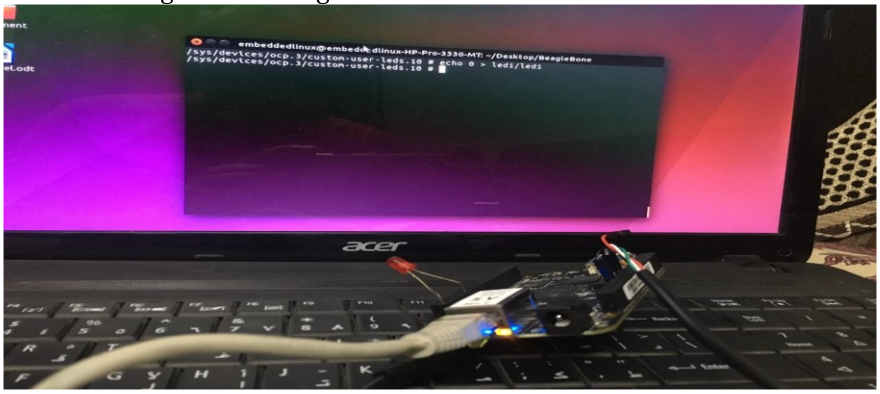
Now, enter the /sys/devices/ocp.3/custom-user-leds.10/ directory and note whether the drivers name "led1", "led2" and "led3" exist into the directory. Fig. 2.4.5 shows the same.



```
/tmp # cd /sys/devices/ocp.3/custom-user-leds.10/
/sys/devices/ocp.3/custom-user-leds.10 # ls
driver     led2      modalias   subsystem
led1       led3      power      uevent
/sys/devices/ocp.3/custom-user-leds.10 #
```

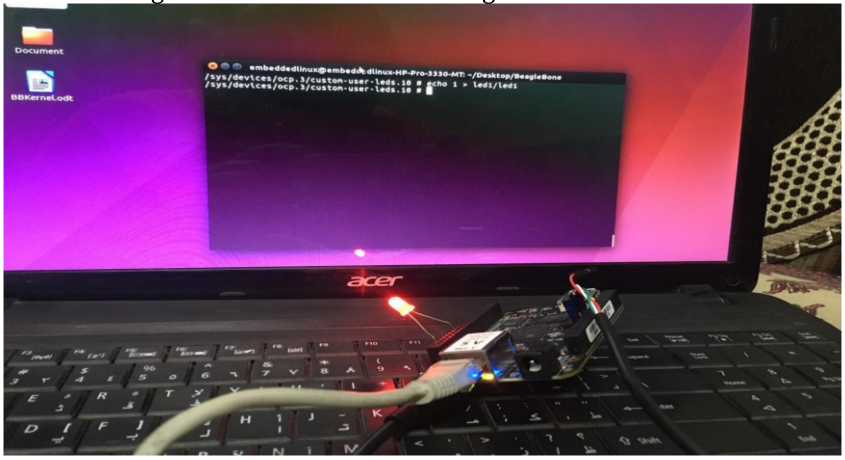
Fig. 2.4.5: Checking whether the devices are added

The GPIOs number 13, 30 and 31 are chosen for access of the driver named "led1", "led2" and "led3" respectively. To control LEDs, use "# echo 0 > led(number)/led(number)" for turning OFF and "# echo 1 > led(number)/led(number)" for turning ON. The experimental results taken on pin number 13 i.e. led1 are shown in below mentioned Fig. 2.4.6 and Fig. 2.4.7.



```
embeddedlinux@embeddedlinux-HP-Pro-3330-MT: ~/Desktop/BeagleBone
/sys/devices/ocp.3/custom-user-leds.10 # echo 0 > led1/led1
/sys/devices/ocp.3/custom-user-leds.10 #
```

Fig. 2.4.6: LED turned OFF using terminal command



```
embeddedlinux@embeddedlinux-HP-Pro-3330-MT: ~/Desktop/BeagleBone
/sys/devices/ocp.3/custom-user-leds.10 # echo 1 > led1/led1
/sys/devices/ocp.3/custom-user-leds.10 #
```

Fig. 2.4.7: LED turned ON using terminal command

FUTURE SCOPE

This project is an experimental demonstration for the embedded linux device driver development. It, not giving the full-fledged functionality, has some serious scope of expanding the same project. Furthermore, not just adding the functionality to GPIOs, but also the functionality of UART, I2C, SPI, USB and Wifi can be added to make fully equipped embedded linux based system of BeagleBone Black. As discussed in the introduction section, most of the technology leaded devices are working on the embedded linux based platform and thus optimizing the resource usage in order to improve the performance of the devices will shortly be an extreme need of the industry, resulting into plenty of opportunities in the field of embedded systems. Moreover, with solid research work going on in the field of high performance computer architecture, building light yet, powerful embedded software has become an area of research interest.

CONCLUSION

The work done in this project covers concepts of; kernel building for specific architecture, booting sequence of BeagleBone Black, kernel module programming and the difference between user-space and kernel-space. There are few of things involved in making a real-time full-fledged embedded system which still remains unexplored in this project which can be worked on. However, with the fundamentals to everything covered in this work, extension up to any level would seem appealing.

REFERENCES

1. D.Dhivakar and L.K.Indumathi. “Common Kernel development for Heterogeneous Linux Platforms”. International Journal of Engineering Research & Technology (IJERT) ISSN: 2278-0181.
2. Bootlin. “Linux Kernel and Driver Development”. Bootlin: 2018: 1-49.
3. Greg Kroah-Hartman. “Linux Kernel in a Nutshell”. Edition 1. Shroff Publishers: Delhi; 2006: 5-121.
4. Lars Wirzenius, Joanna Oja, Stephen Stafford and Alex Weeks. “The Linux System Administrator's Guide” [online]. 2004 [cited 2004 Jan 14]. Available from: URL: http://tldp.org/LDP/sag/html/root-fs.html.
5. “Platform Devices and Drivers” [online]. Available from: URL: https://www.kernel.org/doc/Documentation/driver-model/platform.txt.
6. Robert Nelson. “BeagleBone Black” [online]. 2018 [cited 2018 Oct 12]. Available from: URL: https://www.digikey.com/eewiki/display/linuxonarm/BeagleBone+Black.
7. Arun M Kumar. “Building for BeagleBone” [online]. 2014 [cited 2014 Jun 23]. Available from: URL: https://elinux.org/Building_for_BeagleBone.
8. Gerald Coley. “BeagleBone Black System Reference Manual”. Texas Instruments: 2013: 1-10.