Hardware and Software Requirements

- Ubuntu 22.04 or higher based host PC or VM with a minimum of 50 GB free disk space & better to have atleast 8GB of RAM
- Beagle Bone Black Kit (BBB+USB Cable)
- USB2TTL for accessing the board console
- 4GB or more uSD
- Stable Internet connection to download the packages
- Unlocked USB Ports
- Access to uSD slot or USB card reader for flashing the images
- Access to USB port for connecting the USB to TTL converter to access the board console
- Access to USB port for powering up the board & accessing the network over USB

Setup

Installing the required Packages and Setting up the Toolchain

1. Clone the Git repo

```
$ mkdir ~/Exercises/
$ git clone https://github.com/pradeep-tewani/bbb-builds
$ cd bbb-builds
```

2. Install the toolchain

```
$ make install_toolchain
```

Next, logout & login back for its PATH activation. After logging back, system should have the command arm-linux-gnueabihf-gcc. This means that the toolchain successfully installed

3. Next, step is to install the required package

Execute 'make install_libs' (From bbb-builds directory) and 'make install_pkgs'

4. sudo apt install lzop bison flex libssl-dev bc

Board Setup

Flashing the Images Using the Scripts

1. Plug SD Card into the PC/Laptop. This should show up as the device file in the system. Typically, its /dev/mmcblk0, if its connected in SD Card Slot or /dev/sdb, if its connected using the card reader. However please confirm the exact device file using the 'dmesg' command. **Make sure to execute this command for SD Card only, otherwise it may wipe out the disk contents. Execute ‘dmesg’ to check the exact device file**

Get into the bbb_builds directory & execute the following command:

```
$ cd bbb_builds
$ make generate_prepare_usd
$ cd Utils
```

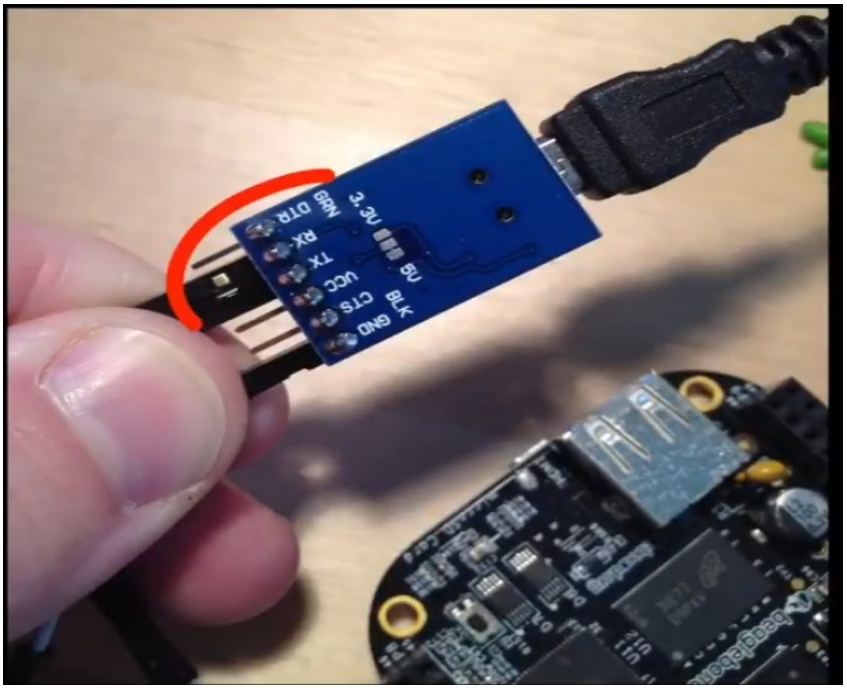
Connect SD Card to the PC/Laptop and make sure to confirm the corresponding device file & then execute the following command:

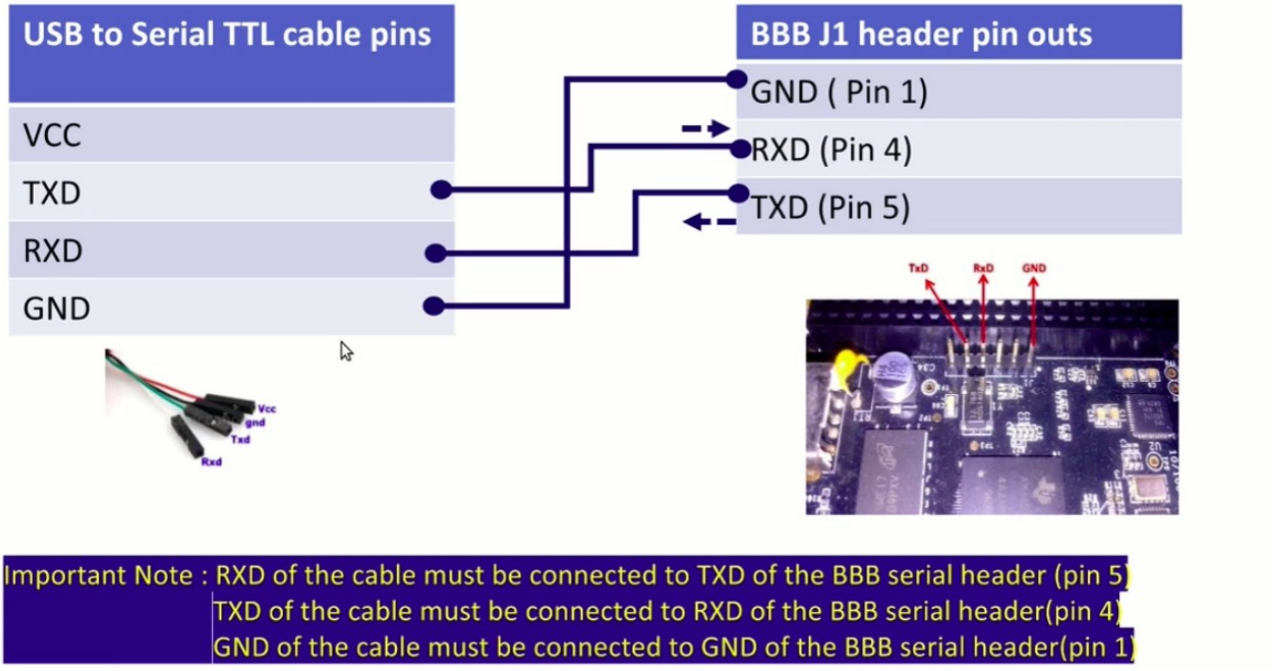
```
$ sudo ./prepare_usd -d <sd card device file>
```

Note: When using the Card Reader, device file can be anything like /dev/sda, /dev/sdb or /dev/sdc depending on the available harddrives/removable drives on the system. It would

/dev/mmcblk0 when the SD Card is plugged directly into the card slot.
Once the command is successfully executed, the card would have the required images

Connect USB2TTL Adapter





Getting the Serial Console Messages

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

1. sudo minicom -s and do the setup for baud 115200 bits 8n1 hw & sw flow control off
2. sudo minicom -o
3. Power on BBB to boot into Linux
4. This should show up the boot up message on minicom and should finally the provide the login prompt

Connecting to the Network Over USB

Refer https://www.digikey.in/en/maker/blogs/how-to-connect-a-beaglebone-black-to-the-internet-using-usb
The link contains the instructions to get the internet on BBB. However, for sessions, the only need is to be able to ssh & scp to the board for transferring the files. No need to connect it to the internet.

Building the Kernel for Beaglebone Black

1. Navigate to the OS directory of bbb-builds
$ cd bbb-builds/OS
2. Get the kernel source code
$ wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.19.103.tar.gz
3. Unpack the kernel source code
$ tar -xvf linux-4.19.103.tar.gz
4. Get into the kernel source code
$ cd linux-4.19.103/
5. Configure the kernel with already available configuration file for Beaglebone Black
$ cp ../Configs/config.4.19.103.default .config
6. Update the Kernel Makefile to cross compile for arm architecture
Add following in Kernel Makefile (Search for ARCH and add as
below)
$ vi Makefile (Open the makefile & add the following at line no. 323)
CROSS_COMPILE=arm-linux-gnueabihf-
ARCH=arm
7. Finally, compile the kernel
$ make zImage
If you get an error as below:
/usr/bin/ld: scripts/dtc/dtc-parser.tab.o:(.bss+0x10): multiple definition of `yylloc'; scripts/dtc/dtc-lexer.lex.o:(.bss+0x0): first defined here
Open scripts/dtc/dtc-lexer.lex.c and find the line 'YYLTYPE yylloc' and change it to 'extern YYLTYPE yylloc'
Restart the build with '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/
Note: Board ip is typically 192.168.7.2
9. Unmount & Reboot the board to boot up with updated kernel
$ umount /mnt (On board)
$ reboot (On board)
10. Verify if the kernel is updated
$ uname -a (Should show the latest kernel build time)

Assignments on Driver Basics

Assignment 1: Testing a Simple Linux Kernel Module

1. Navigate to the P02_lkm directory under device_drivers repo
   $ cd P02_lkm
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 address>:

The board_ip for bbb is 192.168.7.2

5. Load the kernel module
   $ insmod first_driver.ko

This should display “mfd registered” on the console

Assignment 2: Module Parameters

This activity intends to pass the aprameters to Kernel Module during the load time

1. Refer <device_drivers>/P02_lkm/driver_params.c
2. Build the driver with ‘make’ and transfer to the board.
3. Load the module without setting any parameters
4. Unload the module & Load it by setting the parameters as below:

$ insmod driver_params.ko irq=20 name=testdriver addr=0x5000,0x400

Assignment 3: Integrating the Driver

This Activity intends to describe the steps required to integrate any driver into the Kernel build system

1. Copy the file first_driver.c from the <device_drivers>/P02_lkm/ to bbb-builds/OS/linux-4.19.103/drivers/char

$ cd bbb-builds/OS/linux-4.19.103/

$ cp <device_drivers>/P02_lkm/first_driver.c drivers/char

2. Edit the Kconfig file under linux-4.19.103/drivers/char and add the following:

  config MY_DRIVER
  tristate “My Driver”
  help
    Adding this small driver to the kernel

3. Edit the makefile under linux-4.19.103/drivers/char to add the following entry:

obj-$(CONFIG_MY_DRIVER) += first_driver.o

4. Once the modifications are done, configure the kernel:

$ make menuconfig

5. Under the Driver Driver->Character Devices, there will be a menu option MY Driver. Just select it, exit and save the config.

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

6. Compile the kernel

$ make zImage
7. Transfer the newly build kernel to the target board $ mount /dev/mmcblk0p1 /mnt (On board)
$ scp arch/arm/boot/zImage root@<board ip address>:/mnt/
8. Unmount & reboot the board to boot up with updated kernel
$ umount /mnt
$ reboot (On board)
9. Verify if the driver is registered during the boot up
$ dmesg | grep “mfd registered”
Reference: https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system

Character Drivers

Assignments on Character Drivers-1

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 P03_char_driver directory under device_drivers repo folder
$ cd P03_char_driver
Make sure that the KERNEL_SOURCE variable in the Makefile is updated to point to the compiled kernel source
2. Complete all the TODOs (1 and 2) in first_driver.c (Refer apis.txt file in lki directory for the corresponding APIs)
3. Build the kernel module with make. This would generate the kernel module by name first_char_driver.ko
4. Transfer the kernel module to the board.
$ scp first_char_driver.ko root@<board ip address>:
The board_ip for bbb is 192.168.7.2
5. Load the kernel module
$ insmod first_char_driver.ko
This should display “[Major,Minor]
NB: The major/minor number may vary in your case
6. Verify if the driver is registered for corresponding major number
$ cat /proc/devices (first_char_driver should be displayed against Major number 240)

File Operations Registration

The idea over is to register the call back handlers with VFS. Below are the steps:
1. Navigate to the P03_char_driver directory under device_drivers folder
$ cd LKI/lki/P03_char_driver
2. Complete all the TODOs (1, 2 & 3) in null.c (Refer apis.txt file in lki directory for the corresponding APIs)
3. Build the kernel module with make. This would generate the kernel module by name null.ko

4. Transfer the kernel module to the board.
$ scp null.ko root@<board ip address>:
The board_ip for bbb is 192.168.7.2
5. Load the kernel module
$ insmod null.ko
This should display “[Major,Minor]
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 “Embitude” > /dev/abc
(Should display the ascii value of the above string in user space)
8. Test the read operation
$ cat /dev/abc

Sending the data to the 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. Navigate to the P03_char_driver directory under device_drivers folder
$ cd LKI/lki/P03_char_driver
2. Complete TODO 4 in null.c (Refer apis.txt file in lki directory for the corresponding APIs)
3. Build the kernel module with make. This would generate the kernel module by name null.ko
4. Transfer the kernel module to the board.
$ scp null.ko root@<board ip address>:
The board_ip for bbb is 192.168.7.2
5. Load the kernel module
$ insmod null.ko
6. This should display “
NB: The major/minor number may vary in your case
7. Create the device file as per the major number allocated above
$ mknod /dev/abc c 240 0
8. Test the write operation
$ echo -n “Embitude” > /dev/abc
(Should display the ascii value of the above string)
9. Test the read operation
$ cat /dev/abc (Should display ‘e’, which is last character of string provided during write operation)

Assignments on Character Drivers-2

Assignment 1: Automatic device file creation

1. Complete TODOs (1 to 4) in final_char_driver.c under P04_char_driver (Refer apis.txt file in lki directory 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 address>:
   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 entries /dev/finalchar0 and /sys/class/char/finalchar0 should disappear

Assignment 2: Autoloading the driver

The intention of this activity is to demonstrate the autoloading of the driver with mdev rule

1. 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)
2. Create the directory for mdev scripts on the board
   $ mkdir /lib/mdev/ (To be execute on the board)
3. Transfer the mdev configuration file to the board
   $ scp P04_char_driver/AutoLoad/mdev.conf root@192.168.7.2:/etc/
4. Transfer the autoloader script to the board
   $ scp P04_char_driver/AutoLoad/autoloader.sh root@192.168.7.2:/lib/mdev/
5. Transfer the driver to the board
   $ scp P04_char_driver/CharDriver/final_char_driver.ko root@192.168.7.2:
6. Change the permissions for autoloader.sh on the board
   $ chmod a+x /lib/mdev/autoloader.sh (To be executed on the board)
7. Next step is to plug the pen drive or either carefully remove & re-insert the uSD. This should load the final_char_driver.ko
8. Unplug the pen drive or carefully remove the uSD. This should unload the driver

Assignment 3: Controlling the on-board Led

The idea over is to demonstrate controlling the hardware with character driver. However, there is already an driver controlling few of the leds. So, we can detach it using the below commands:

$ echo none > /sys/class/leds/beaglebone\:green\:heartbeat/trigger

$ $ echo none > /sys/class/leds/beaglebone\:green\:usr2/trigger

Below are the steps:
1. Complete all the TODOs (1 & 2) in P04_char_driver/led.c (Refer apis.txt file in lki directory 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 lki/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

Assignment 4: 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 P04_char_driver/ (Refer apis.txt file in lki directory 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 Templates/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, Try playing around with ioctls for selecting the leds and switching on/off the leds

Interrupt Management

Assignment 1: Registering the interrupt handler for the switch

Get into the P07_interrupts_bh directory
$ cd P07_interrupts_bh
2 Complete all the TODOs (1 & 2) in interrupts.c. Refer apis.txt for the corresponding APIs
3 Build the module with make and transfer it to the board
$ scp interrupts.ko root@<board_ip>:
4 Load the module on the board
$ insmod interrupts.ko (With this, the insmod should be put to sleep for 2 secs)
5. Verify if the handler is registered
$ cat /proc/interrupts (This should display the label used in request_irq against the IRQ number)
6. Press the switch S2 (The handler should get invoked and print the corresponding GPIO value)

Enabling the switch S2 & try unblocking with switch interrupt

1. Get into the Kernel Directory:
 $ cd bbb-builds/OS/linux-4.19.103
2. Apply the patch as below:
 $ patch -p1 < ../Patches/0001-Set-up-the-pin-mux-for-button-S2.patch
3. Build the DTB
 $ make dtbs
4. Mount the uSD first partition on board
 $ mount /dev/mmblk0p1 /mnt (To be executed on the board)

5. Transfer the DTB to the board:
$ scp arch/arm/boot/dts/am335x-boneblack.dtb root@192.168.7.2:/mnt/
6. Unmount the first partition
$ umount /mnt (To be executed on the board)
7. Reboot the board with reboot command (On board)
8. Transfer the kernel module to the board.
$ scp spin_lock.ko root@
9. Load the driver on the board
 $ insmod spin_lock.ko
10. Now, press the Switch S2 (S2 is located just above uSD) and observe the behaviour

Assignment 2: Replace polling with interrupt

1 Copy P06_timing_mm/monitor_sw.c to P07_interrupts_bh
2 Modify monitor_sw.c to register the interrupt and thereby replacing polling with the interrupts
3 Build the module for workqueue with make and transfer it to the board
$ scp monitor_sw.ko root@<board_ip>:

4 3. Load the module on the board
5 $ insmod monitor_sw.ko
Press the Switch S2 and this should print the button status

Assignment 3: Register the tasklet

1 Navigate to the P07_interrupts_bh directory
$ cd P07_interrupts_bh
2 Build the module for tasklet with make and transfer it to the board
$ scp tasklet.ko root@<board_ip>:
3 Load the module on the board
$ insmod tasklet.ko
4 This should invoke the tasklet handler which displays
“my_tasklet_function_was_called”

Assignment 4: Register the workqueue

1 Navigate to the P07_interrupts_bh directory
$ cd P07_interrupts_bh
2 Build the module for workqueue with make and transfer it to the board
$ scp work_queue.ko root@<board_ip>:
3 Load the module on the board
4 $ insmod work_queue.ko
5 This should invoke the handlers for the both the registered works and should display:
my_work.x 1
my_work.x 2