Beaglebone Black Exercises

Training Material

All the training material is maintained at Github. Clone the repository with:

$ git clone https://github.com/embitude/training

Session-1 Exercises

Configure & Build the Kernel for BBB

1. Navigate to the OS directory of Builds
   $ cd <path_to_training_repo>/Builds/OS
2. Get the kernel source code
   $ wget https://mirrors.edge.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 Builds/OS/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 update as below)
   CROSS_COMPILE=arm-linux-gnueabihf-
   ARCH=arm
7. Finally, 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/ (On the system)
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)

Write a simple Linux Kernel Module

1. Navigate to the FirstDriver directory under Exercises folder
   $ cd <path_to_training_repo>/Exercises/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 Copy the file *first_driver.c* from the Exercises/FirstDriver/ to Builds/OS/linux-4.19.103/drivers/char
$ cp Exercises/FirstDriver/first_driver.c Builds/OS/linux-4.19.103/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
Under the Driver Driver->Character Devices, there will be a menu option *MY Driver*. Just select it, exit and save the config.
5 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>:/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 “driver registered”

Reference: https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system

Session-2 Exercises

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 Exercises folder
$ cd Exercises/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 “Timmins” > /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 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 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

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 “Timmins” > /dev/abc
(Should display the ascii value of the above string)

8 Test the read operation

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

Session-3 Exercises

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 entries */dev/finalchar0* and */sys/class/char/finalchar0* should disappear

Autoloading the driver

The intention of this activity is to

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 AutoLoad/mdev.conf root@192.168.7.2:/etc/
4. Transfer the autoloader script to the board
   $ scp AutoLoad/autoloader.sh root@192.168.7.2:/lib/mdev/
5. Transfer the driver to the board
   $ scp Exercises/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

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 Exercises/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 Exercises/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 Exercises/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 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

Session-4 Exercises

Consumer/Producer

The objective over here is to handle the synchronization issues with multiple threads of execution. The relevant codes could be found under
Exercises/ProcessMngmt/Synchronization

Part (i) Protect the link list head

1. Refer cons_prod.c under Exercises/ProcessMngmt/Synchronization directory. It has the global variable *job_queue*, which is shared across the consumer threads and the producer thread.
2. Use appropriate mechanism to protect the global variable.
3. Refer apis.txt for the API details.
4. Navigate to Synchronization directory and build the kernel module with make. This would generate the kernel module by name cons_prod.ko
5. Transfer the kernel module to the board.
   $ scp cons_prod.ko root@<board_ip>:
6. Load the driver on the board
   $ insmod cons_prod.ko
7. This should spawn the consumer and producer threads. Observe the behaviour.

Part (i) Enhance the driver to keep on processing the added nodes

Spinlocks

Part (i)

1. Refer spin_lock.c under Exercises/ProcessMngmt/Synchronization directory
2. Navigate to Synchronization directory & build the kernel module with make. This would generate the kernel module by name spin_lock.ko
3. Transfer the kernel module to the board.
   $ scp spin_lock.ko root@<board_ip>:
4. Load the driver on the board
   $ insmod spin_lock.ko
   Wait for 30 secs and observe the behaviour

Part (ii) Enabling the switch S2 & try unblocking with switch interrupt

1. Get into the Kernel Directory:
   $ cd Builds/OS/linux-4.19.103
2. Apply the patch as below:
   $ patch -p1 < Builds/OS/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@<board_ip>:
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

Basic driver to block the process

The objective here is to write the basic driver which blocks the user space process. All the relevant code could be found under Exercises/ProcessMngmt/WaitingBlocking directory

Part (i) Demonstrate using schedule() API

The intention over here is to demonstrate the usage of schedule() API for blocking the process

1. Navigate to Exercises/*ProcessMngmt/WaitingBlocking* directory.The *sched.c* invokes schedule() API in read handler. Build the kernel module with make. This would generate the kernel module by name sched.ko
2. Transfer the kernel module to the board.
   $ scp sched.ko root@<board_ip>:
3. Load the driver on the board
   $ insmod sched.ko
   This would create the device file with the name /dev/mychar0
4. Invoke *cat* on the device file and observe the behaviour
   $ cat /dev/mychar0

Part (ii) Setting the process state

This hands-on demonstrates the usage of set_current_state() API to update the process state

1. Navigate to Exercises/*ProcessMngmt/WaitingBlocking* directory The *sched1*.c invokes set_current_state() before schedule() API in read handler. Build the kernel module with make. This would generate the kernel module by name sched1.ko
2. Transfer the kernel module to the board.
   $ scp sched1.ko root@<board_ip>:
3. Load the driver on the board
   $ insmod sched1.ko
   This would create the device file with the name /dev/mychar0
4. Invoke *cat* on the device file and observe the behaviour
   $ cat /dev/mychar0 (This should block the *cat* process)
5. Next step is to wake up the process. For this, use ssh to get another shell for the board
   $ ssh root@<board_ip>
6. Invoke *echo* on /proc/wait
   $ echo 1 > /proc/wait
   This would unblock the *cat* process blocked earlier

Part (iii) Waiting on the condition

The demonstration in part (ii) blocks/unblocks the process unconditionally. The intention for this demonstration is to make the process wait for some condition say ‘flag == y’.

1. Navigate to Exercises/*ProcessMngmt/WaitingBlocking*. The sched2.c waits on the condition ‘flag == y’. Build the kernel module with make. This would generate the kernel module by name sched2.ko
2. Transfer the kernel module to the board.
   $ scp sched2.ko root@<board_ip>:
3. Load the driver on the board
   $ insmod sched2.ko

This would create the device file with the name /dev/mychar0

4. Invoke *cat* on the device file:
   $ cat /dev/mychar0 (This should block the *cat* process)
5. Next step is to wake up the process. For this, use ssh to get another shell for the board
   $ ssh root@<board_ip>
6. Invoke *echo* on /proc/wait
   $ echo 1 > /proc/wait
   This would not unblock the *cat* process, since the condition 'flag == y' is not met
   $ echo y > /proc/wait (This should unblock the *cat* process)

Using Wait Queue to block the process

The previous examples use basic APIs to block/unblock the process. However, this may lead to issues, if not handled cautiously. The kernel provides the programming wait queue programming constructs to avoid such issues.

1. Complete all the TODOs (1 to 3) in wait_queue.c under Exercises/ProcessMngmt/WaitingBlocking directory (Refer apis.txt for wait queue APIs and wait_queue_ref.c for example usage)
2. Build the kernel module with make. This would generate the kernel module by name wait_queue.ko
3. Transfer the kernel module to the board.
   $ scp wait_queue.ko root@<board_ip>:
4. Load the driver on the board
   $ insmod wait_queue.ko
   This would create the device file with the name /dev/mychar0
5. Invoke *cat* on the device file:
   $ cat /dev/mychar0 (This should block the *cat* process)
6. Next step is to wake up the process. For this, use ssh to get another shell for the board
   $ ssh root@<board_ip>
7. Invoke *echo* on /proc/wait
   $ echo 1 > /proc/wait
   This would not unblock the *cat* process, since the condition 'flag == y' is not met
   $ echo y > /proc/wait (This should unblock the *cat* process)

Blocking the process using select API

The objective for this exercise is to understand the changes required in the driver to enable the support for the select system call. The relevant code could be found under Exercises/ProcessMngmt/Select directory.

Part (i) Using select with single file descriptor

1. Navigate to Exercises/ProcessMngment/Select directory
2. Build the kernel module with make. This would generate the kernel module by name select.ko
3. Transfer the kernel module to the board.
   $ scp select.ko root@<board_ip>:
4. Navigate to the Exercises/Apps and build the application with *make.* This would generate the application with the name select_ap.
5. Transfer the application to the board.
   $ scp select_ap root@<board_ip>:
6. Load the driver on the board
   $ insmod select.ko

This would create the device file with the name /dev/mychar0 and the entry /proc/wait

7. Execute the application
   $ ./select_ap (This would block the application)
   The driver should display the following message:
   In Poll
   Out Poll
8. Open another shell for the board and execute *echo* on /proc/wait:
   $ ssh root@<board_ip>
   $ echo y > /proc/wait (The application should unblock and print the value received from the driver. The application runs in while loop)

Part (ii) Using select with couple of file descriptors

The objective of this exercise is to understand how application works with multiple file descriptors at the same time

1. Navigate to Exercises/ProcessMngment/Select directory
2. Build the kernel module with make. This would generate the kernel module by name select.ko and select_gpio.ko
3. Transfer both the kernel modules to the board.
   $ scp select.ko select_gpio.ko root@<board_ip>:
4. Navigate to the Exercises/ProcessMngment/Apps and build the application with *make.* This would generate the application with the name select_multiple_fds.
5. Transfer the application to the board.
   $ scp select_multiple_fds root@<board_ip>:
6. Load the drivers on the board
   $ insmod select.ko
   $ insmod select_gpio.ko
   This would create the device file with the name /dev/mychar0 and /dev/gpio0 and the entry /proc/wait
7. Execute the application
   $ ./select_multiple_fds (This would block the application)
   The driver should display the following message:
   In Poll
   Out Polls
   In Poll – GPIO
   Out Poll - GPIO
8. Open another shell for the board and execute *echo* on /proc/wait:
   $ ssh root@<board_ip>
   $ echo y > /proc/wait (The application should unblock and print the value received from the driver. The application runs in while loop)
   The application should display – “Got input on second device file”
9. Next, press the switch S2 for 2 secs. This should unblock the application and it should display – “Got input on second device file”