Lab session 8

Kernel Module development

Last time we looked at the cross-compiler environment that would be used for developing more serious applications for the Linux system. In this lab we are interested in kernel modules and since these are usually small modules and not developed very often, we will go back to using the “nano” editor and the compiler on the Beaglebone.

Step 1: Connect the Beaglebone board to your computer via the USB cable only!

Step 2: Launch Putty, select SSH and enter the IP address 192.168.7.2. Login as debian:temppwd. You should have a Linux console. This shows that all the connections are working.

Now we will basically be following some examples given by:
http://derekmolloy.ie/writing-a-linux-kernel-module-part-1-introduction/

However, be warned!! Some of the example code on the website has errors so please make sure you download the source code files from the wiki.

Step 3: Linux headers install.
To be able to compile kernel modules, we need to have a set of .h files as well as some build tools. These are usually contained in a package that can be downloaded and installed.

If you have a proper network connection to the BeagleBone then something like this will do an automatic install:

```
debian@beaglebone:/$ sudo apt-get install linux-headers-$(uname -r)
```

“uname -r” specifies what version of linux you have:

```
debian@beaglebone:/$ uname -r
4.14.108-ti-r113
```

It is important that the headers you use match the version of linux:

They can be downloaded from: http://repos.rcn-ee.com/debian/pool/main/l/linux-upstream/

The linux header installer for our Beaglebone can also be downloaded from the lab wiki.
File name: linux-headers-4.14.108-ti-r113_1stretch_armhf.deb

I suggest you copy them onto a USB stick and then “mount” the USB stick in the Beaglebone.

Plug in the USB stick and then look for the name of the USB drive using:

```
debian@beaglebone:~$ sudo fdisk -l
```

Find the device name that matches your USB stick and then mount the drive:

```
debian@beaglebone:~$ sudo mount /dev/sda1 /media/usb-drive
debian@beaglebone:~$ cd /media/usb-drive
debian@beaglebone:/media/usb-drive$ ls
```

You should see the file:

```
linux-headers-4.14.108-ti-r113_1stretch_armhf.deb
```

Install headers using:

```
debian@beaglebone:/media/usb-drive$ sudo apt install ./linux-headers-4.14.108-ti-
r113_1stretch_armhf.deb
```

Check they are installed:

```
debian@beaglebone:/lib/modules$ ls
4.14.108-ti-r113
```

Step 4: Set up our project area:

```
debian@beaglebone:/$ sudo mkdir exploringBB
debian@beaglebone:/$ cd exploringBB
debian@beaglebone:/exploringBB$ sudo mkdir extras
debian@beaglebone:/exploringBB$ cd /exploringBB/extras
debian@beaglebone:/exploringBB/extras$ sudo mkdir kernel
debian@beaglebone:/exploringBB/extras$ cd /exploringBB/extras/kernel
```

Step 5: Dummy kernel module example “hello”
The first Kernel module example we will look at is very simple in that it only implements the initialisation and exit phases perform during kernel module loading and removing.

Download the file “Hello src” from the lab wiki. Open it up in an editor and have a look through the source code. You will notice that the module is basically two functions, “helloBB_init” and “helloBB_exit”.

Create a directory for our “hello” kernel module project:

```
debian@beaglebone:/exploringBB/extras/kernel$ sudo mkdir hello
debian@beaglebone:/exploringBB/extras/kernel$ cd /exploringBB/extras/kernel/hello
```

Create the file “hello.c” and copy the code “Listing 1: The Hello World Linux Loadable Kernel Module (LKM) Code” into it. (right clicking into the nano window will insert copied text)

```
debian@beaglebone:/exploringBB/extras/kernel/hello$ sudo nano hello.c
```

Save the file and exit nano.

Create the Make file “Makefile” and copy code “Listing 2: The Makefile Required to Build the Hello World LKM” into it. Make sure you maintain the spacing, you may need to add tabs before the make commands.

```
debian@beaglebone:/exploringBB/extras/kernel/hello$ sudo nano Makefile
```

Save the file and exit nano.

Now build the project.

```
debian@beaglebone:/exploringBB/extras/kernel/hello$ sudo make
```

Check to see if you have the .ko file:

```
debian@beaglebone:/exploringBB/extras/kernel/hello$ ls -l *.ko
-rw-r--r-- 1 root root 4828 Oct  7 12:49 hello.ko
```

Insert the kernel module:

```
debian@beaglebone:/exploringBB/extras/kernel/hello$ sudo insmod hello.ko
```

To see if it is loaded use:

```
debian@beaglebone:/exploringBB/extras/kernel/hello$ lsmod
```

You can observe some info using:

```
debian@beaglebone:/exploringBB/extras/kernel/hello$ modinfo hello.ko
```

Now remove the Kernel module:

```
debian@beaglebone:/exploringBB/extras/kernel/hello$ sudo rmmod hello.ko
```

We can look at the Kernel log file to see the message from our Kernel module:

```
debian@beaglebone:/$ sudo su -
root@beaglebone:~# cd /var/log
root@beaglebone:/var/log# tail -f kern.log
```

The last lines in the log should be messages from our Kernel module generated during the loading and unloading phases. Have a look at “Listing 1” to see how this is done.

use ctrl-c to get out of the viewer and “exit” to logout of “superuser”:

```
root@beaglebone:/var/log# exit
```

Step 5: Char driver example

The second Kernel module example we will look at not only implements the initialisation and exit phases but now has the typical "open/close/read/write" functions.

Download the file "Ebbchar src" from the lab wiki. Open it up in an editor and have a look through the code. You will notice that this time we have a separate "test" program. This is an example of a user space program that interacts with a kernel module.

Create a directory for our "ebbchar" kernel module project:

```
debian@beaglebone:/exploringBB/extras/kernel$ sudo mkdir
/exploringBB/extras/kernel/ebbchar
```

As before, create the necessary files and copy the appropriate code into them:

```
debian@beaglebone:/exploringBB/extras/kernel$ cd /exploringBB/extras/kernel/ebbchar
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ sudo nano ebbchar.c
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ sudo nano testebbchar.c
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ sudo nano Makefile
```

Now build the project:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ sudo make
```

Again, check to see if you have the .ko file:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ ls -l *.ko
-rw-r--r-- 1 root root 9256 Oct  7 08:08 ebbchar.ko
```

Also check if we have the test file:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ ls -l test
-rwxr-xr-x 1 root root 8744 Oct  7 08:08 test
```

Insert the kernel module:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ sudo insmod ebbchar.ko
```

To see if it is loaded use:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ lsmod
```

The device is now present in the /dev directory, with the following attributes:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ cd /dev
debian@beaglebone:/dev$ ls -l
crw-rw-rw- 1 root root    240,   0 Oct  7 08:15 ebbchar
```

We can run our test program:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ sudo ./test

Starting device test code example...
```

```
Type in a short string to send to the kernel module:
I hope this works
Writing message to the device [I hope this works].
Press ENTER to read back from the device...

Reading from the device...
The received message is: [I hope this works(17 letters)]
End of the program
```

Remove the kernel module:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ sudo rmmod ebbchar
```

Look at the kernel log:

```
debian@beaglebone:/exploringBB/extras/kernel/ebbchar$ sudo tail -f /var/log/kern.log
Oct  7 08:15:39 beaglebone kernel: [ 3651.725969] EBBChar: registered correctly with major number 240
Oct  7 08:15:39 beaglebone kernel: [ 3651.726091] EBBChar: device class registered correctly
Oct  7 08:15:39 beaglebone kernel: [ 3651.726399] EBBChar: device class created correctly
Oct  7 08:20:45 beaglebone kernel: [ 3958.165936] EBBChar: Device has been opened 1 time(s)
Oct  7 08:21:40 beaglebone kernel: [ 4013.511580] EBBChar: Received 17 characters from the user
Oct  7 08:21:47 beaglebone kernel: [ 4020.071500] EBBChar: Sent 29 characters to the user
Oct  7 08:21:47 beaglebone kernel: [ 4020.072039] EBBChar: Device successfully closed
Oct  7 08:23:37 beaglebone kernel: [ 4130.345251] EBBChar: Goodbye from the LKM!
```

Step 6: Reflection

Now the example we just looked at is not really a device driver because it doesn't actually interact with a peripheral device. However, it does give us the framework and building blocks to implement an actual device driver. Now, using this example as a guide, describe how you would go about implementing a device driver for a serial port. Describe what you would do for each of the functions: open/close/read/write. Three things need to be considered here:

1. The use of char buffers so more than one character can be dealt with at a time.
2. Interrupts. We haven't used those in these examples but they can be easily used and managed by the kernel module. The next example in the "Molloy" series deals with this.
3. There is a mechanism for a few parameters to be passed to the driver during the "open" phase. What parameters do you think you would need to set up the serial port.