Driver Creation Guide for BeagleBone

by Brian Fraser
Last update: June 28, 2018

This document guides the user through:

1. Compiling the Linux kernel and downloading to the BeagleBone.
2. Creating and compiling a new new Linux driver.
3. Loading and unloading the driver.

Table of Contents

Tested under Ubuntu 16.04 with Target kernel 4.4

Note: This guide has not yet been tested in the CSIL Labs. Some changes may be needed.

Formatting:

1. Commands starting with `$` are Linux console commands on the host PC:
   ```
   $ echo "Hello PC world"
   ```
2. Commands starting with `#` are Linux console commands on the target board:
   ```
   # echo "Hello Target world"
   ```
3. Commands starting with => are U-Boot console commands:
   ```
   => printenv
   ```
4. Commands are case sensitive in Linux and U-Boot.

Document History:

- June 28, 2018: Minor updates to formatting / wording.

1. Install TFTP Server

1. Install TFTP server on host:
   ```
   $ sudo apt-get install tftpd-hpa
   ```
2. Configure the directory you wish to make public via TFTP[1]:
   ```
   $ sudo gedit /etc/default/tftpd-hpa
   ```
   - Change the file to the following. Change *user_name* to your user name on your host system:
     ```
     TFTP_USERNAME="tftp"
     TFTP_ADDRESS="0.0.0.0:69"
     TFTP_OPTIONS="--create --listen --verbose /home/user_name/cmpt433/public"
     RUN_DAEMON="yes"
     ```
3. Check if the TFTP server is running:
   ```
   $ netstat -a | grep tftp
   ```
   - It should display a line similar to:
     ```
     udp        0      0 *:tftp                  *:*
     ```
4. Restart the server for changes to take effect.
   ```
   $ sudo service tftpd-hpa restart
   ```
5. Test the TFTP server is operating correctly:
   - Install the TFTP client:
     ```
     $ sudo apt-get install tftp
     ```
   - Create a test file in the `~/cmpt433/public` folder which we'll download via TFTP:
     ```
     $ echo 'Coming via TFTP' > ~/cmpt433/public/test_tftp.txt
     ```
   - Download the file via TFTP (changing user_name to be your user name!):
     ```
     $ tftp
     tftp> connect 127.0.0.1
     tftp> get /home/user_name/cmpt433/public/test_tftp.txt
     Received 17 bytes in 0.0 seconds
     tftp> quit
     $ cat test_tftp.txt
     Coming via TFTP
     $
     ```

1 Using `gksudo` over `sudo` is preferred for running graphical applications as root. However, left as `sudo` here for simplicity.

6. Troubleshooting
   - There is a bug in Ubuntu 14.04 that makes the TFTP server not start at boot-up.
     - You can manually start the server each time with the above restart command.
     - Configure to start automatically by editing `/etc/init/tftpd-hpa.conf` and replacing the line:
       "`start on runlevel [2345]`"
       with:
       "`start on (filesystem and net-device-up IFACE!=lo)`"
       and now it starts on boot.
   - If you get the error "Error code 2: Forbidden directory" then ensure that the `/etc/default/tftpd-hpa` file correctly lists the full path of the folder you are trying to access via TFTP. Did you change "user_name" correctly?
   - If you get the error: "Error code 0: Permission denied.", then check that the file and the shared folder (`~/cmpt433/public/`) has read permissions for all users, and that all directories up to the one shared are readable and executable by all users.
     - Check the file permissions:
       ```
       $ ls -la ~/cmpt433/public/test_tftp.txt
       -rw-rw-r-- 1 brian brian 16 Oct 11 14:58 test_tftp.txt
       ```
       - Must have the 'r' in 3 times in the permissions. Correct with:
         ```
         $ chmod a+r ~/cmpt433/public/test_tftp.txt
         ```
     - Check the directory permissions:
       ```
       $ ls -lad /home ~ ~/cmpt433 ~/cmpt433/public
       drwxr-xr-x  3 root  root  4096 May 13 23:57 /home
       drwxr-xr-x 39 brian brian 4096 Oct 27 19:55 /home/brian
       drwxr-xr-x  5 brian brian 4096 Oct 24 22:38 /home/brian/cmpt433
       drwxrwxr-x 15 brian brian 4096 Oct 27 20:01 /home/brian/cmpt433/public
       ```
       - Correct with:
         ```
         $ chmod a+rx /home/ ~ ~/cmpt433/ ~/cmpt433/public/
         ```
     - If you get the error: "`tftp: test_tftp.txt: Permission denied`" it may mean you are trying to write the file (after downloading it) into a folder you don't have write permission for. Double check you have write permission in the current directory of your TFTP client.

2. Building the Kernel

To build the kernel, we'll use scripts created by Robert Nelson (https://eewiki.net/display/linuxonarm/BeagleBone+Black) which target the BBB. First we get these from GitHub, then we execute them.

It will download ~1GB+, consume ~5 gigs of space on your host to download and build the kernel. It is best to do this in your normal Linux file system on the host; however, you can do all of this onto a USB memory stick of size ~8+ GB if your VM has not got the space. If you need to use a USB stick, first format your USB stick to be EXT4 (via the Disks program on Ubuntu) otherwise permissions and symbolic links will not work correctly. Took 3 hours on USB

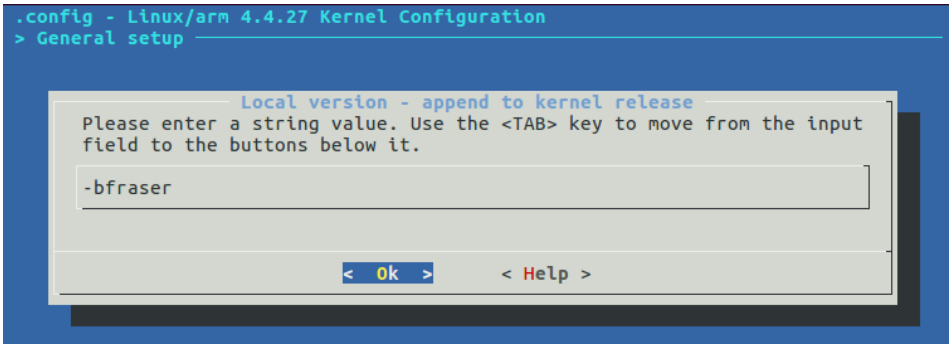
1. Clone the BeagleBone build scripts from GitHub[2]:
   ```
   $ cd ~/cmpt433/work
   $ git clone https://github.com/RobertCNelson/bb-kernel.git
   ```
2. Checkout the scripts to download and build a specific kernel version:
   ```
   $ cd bb-kernel/
   $ git checkout tags/4.4.95-bone19
   ```
   - To get a more recent version, you could also use:
     ```
     $ git checkout origin/am33x-v4.4 -b v4.4
     ```
     While typing "`origin/am`", if you hit TAB then GIT will list all matching branches. You can use this to select a different kernel version (putting it onto the appropriately named branch). Other kernel versions have not been tested for use in this course, but could very well work fine too.

     Note only v3.8 and v4.1 and higher have built-in support the BeagleBone cape manager.
   - If you are unable to checkout due to an error about uncommitted changes, try:
     ```
     $ git stash
     ```
     This will "stash" the changed files for later retrieval (if desired).
3. Run the build script to download the correct compiler, the kernel source code, patch it, and build it. This may take some time! (Took 22 minutes on my home desktop VM; but requires input after about ~10 minutes -- see next step).
   ```
   $ ./build_kernel.sh
   ```
   - This may fail and ask you to install additional packages. Install the packages, then retry the command.
   - This may fail with GIT asking for your email and name. You can use:
     ```
     $ git config --global user.email "yourId@sfu.ca"
     $ git config --global user.name "Your Name"
     ```
     - Change `yourId` and "`Your Name`" to the correct values.
   - If a failure occurs, correct the problem and re-run the script.

2 If building off a USB drive, it will likely be mounted on your host in /media/<user-name>/ ...

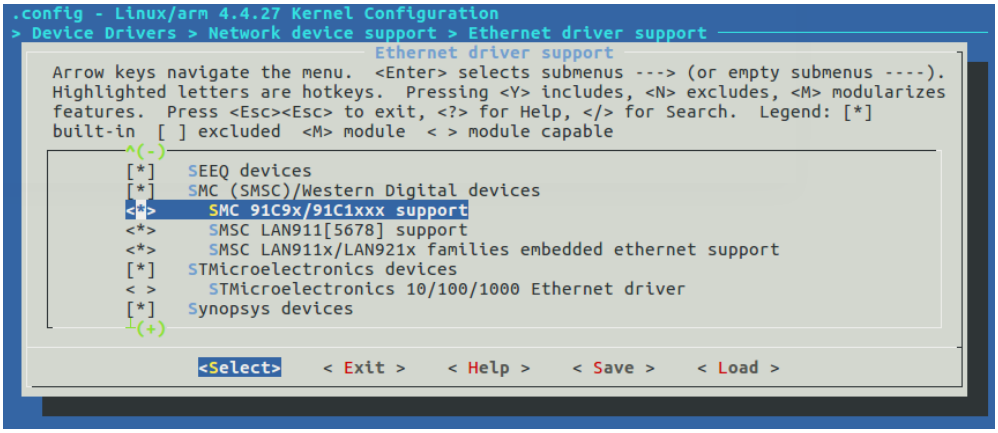
4. When the blue kernel configuration menu appears, change the Local Version to be your user ID:
   - Select `General Setup -->` , and press Enter.
   - Select `Local version - append to kernel release` and press Enter.
   - Type in a dash and your SFU ID (your login) and press Enter, such as, and shown below in the screen shot:
     `-bfraser`

     Screen shot:



   - Go back up to the top menu by pressing Right and select Exit

5. Also in the menu, build in drivers for the BeagleBone's Ethernet chip to the kernel image
   - Select `Device Drivers --->`, and press Enter
   - Select `Network device support --->`, and press Enter
   - Select `Ethernet driver support --->`, and press Enter
   - Make the three SMC (or SMSC) Ethernet devices be built-in ("included") in the kernel image by selecting them and typing 'Y' to get the * beside them.
     - `SMC 91C9x/91C1xxx support`
     - `SMSC LAN911[5678] support`
     - `SMSC LAN911x/LAN921x families embedded ethernet support`
   - Screenshot



6. Save and exit the menu:
   - Save the file as `.config` (press Right to select `Save`, accept `.config` file name).
   - Select `Exit` until you exit the menu (press Right to select Exit from menu).

- Note: if you rebuild the kernel again, this setting (via the blue kernel configuration menu) should persist correctly.
- **Once you exit, the kernel will continue to build. This may take a long time (3+ hours) on some systems. It may also seem to stop for a minute or two and then continue.**

  However, rebuilding the system in the future will be quite a bit faster because it does not need to download any tools, nor re-compile much of the work initially compiled.

7. Build Products:
   The custom build script you just ran will produce the following files:

   - Device tree file for the BeagleBone Green, loadable by UBoot to configure the peripherals on the board:
     `~/cmpt433/work/bb-kernel/KERNEL/arch/arm/boot/dts/am335x-bonegreen.dtb`

     Use `.../am335x-boneblack.dtb` if on a BeagleBone Black.
   - Linux kernel to booted on the target. The minor revision number may change as the scripts are updated (on GitHub) to refer to newer kernel versions, and your script will build a file with your user ID's name
     `~/cmpt433/work/bb-kernel/deploy/4.4.95-bfraser-bone19.zImage`

   Copy the build products into `~/cmpt433/public/` for access via TFTP on the target.

```
$ cd ~/cmpt433/work/bb-kernel/
$ cp KERNEL/arch/arm/boot/dts/am335x-bonegreen.dtb ~/cmpt433/public
$ cp deploy/4.4.95-bfraser-bone19.zImage ~/cmpt433/public
```

8. Troubleshooting

   - If you are running Ubuntu 14.04, the script may fail with an error stating you may need:
     - A new version of Git:
       ```
       $ sudo apt-add-repository ppa:git-core/ppa
       $ sudo apt-get update
       $ sudo apt-get install git
       ```
     - Git clone fails with gnutls_handshake():
       ```
       $ sudo apt-get upgrade libcurl3-gnutls
       ```

3. Downloading the Kernel

3.1 Supported Connections

To perform these steps, you need *both*:

1. A **TTL serial** connection to the target (such as through the Zen Cape's TTL over USB conection, or a separate USB to TTL cable).
2. A **real Ethernet** connection between the host and the target.
    - Can be via a normal Ethernet connection to an existing LAN. In this case, both the host and target likely use DHCP to automatically get an IP address (which may change over time).
    - Can be via a direct Ethernet cable connection between the target and the host (such as a laptop). This likely uses a static IP address. No need to use a cross-over cable; the target hardware works with a standard Ethernet cable.
        - If running Linux natively (no virtual machine), then Linux should automatically detect the direct Ethernet connection to the target. You may need to set the IP address of the Linux host manually.
        - If using a virtual machine to run Linux on your host PC with a direct cable connection between the host PC and the target, then you'll want to configure the VM to have two network connections: one as NAT connecting the guest OS to the host OS's normal internet connection (likely WiFi), and the second as a Bridged connection connecting the guest OS to the physical Ethernet port (and hence the target). See the quick-start guide for how to test this configuration.
    - **It is not possible to use Ethernet over USB (passthroguh) for working with UBoot.** Ethernet over USB can be used only when the Linux kernel has booted on the target and begun emulating the connection. For working with the Linux kernel, and for working with bare-metal applications, Ethernet over USB is insufficient.

You should be able to develop on the SoSy Linux lab PC (via a Virtual Machine) and have your target connected to one of the spare Ethernet cables. However, if you are using your laptop as the host, it will likely *not* work to have your laptop connected via SFU WiFi and your target via the lab's spare Ethernet cable as the SFU network will likely not allow packets to be transmitted from your laptop (on Wi-Fi) to the lab sub-net (on Ethernet).

Required Actions

1. Connect your BeagleBone to the host PC using Ethernet (described above).
2. Over your Ethernet connection, ping the host from the target. For example, if your host's static IP on the Ethernet cable is 192.168.2.1, then:
   `# ping 192.168.2.1`
3. If there is a problem, use `ifconfig` on both the host and target to ensure both are configuring the Ethernet correctly. Also consult the networking guide for more information.

3.2 Downloading Steps

1. Connect to the target using a serial port connection (Zen cape's TTL, or a USB to Serial cable) using `Screen` or `Minicom`.
   - An SSH connection will not work because we need access to UBoot. If having problems with your serial connection, see the Serial Connection Troubleshooting step at the end of this section.
2. Reboot the target from Linux into UBoot:
   ```
   # reboot
   ```
3. Press the space bar (or any key) when it starts to boot to enter the interactive UBoot prompt.
   ```
   U-Boot SPL 2016.03-00001-g148e520 (Jun 06 2016 - 11:27:44)
   Trying to boot from MMC
   bad magic


   U-Boot 2016.03-00001-g148e520 (Jun 06 2016 - 11:27:44 -0500), Build:
   jenkins-github_Bootloader-Builder-395

          Watchdog enabled
   I2C:   ready
   DRAM:  512 MiB
   Reset Source: Global warm SW reset has occurred.
   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
   =>
   ```
4. The U-Boot prompt accepts commands, such as:
   - `help`: display a list of commands available.
   - `printenv`: display all environment variables currently set.
   - `setenv`: set an environment variable in U-Boot.
   - `boot`: continue booting as normal.
5. Note that U-Boot usually runs on boards which feature some EEPROM to which one can save the environment variables. Unfortunately, the BeagleBone does not have this. Therefore, each time you reboot all changes made to the environment variables in U-Boot will be lost.
   - The best way to work around this is to keep a copy of the necessary commands in a text-editor and copy-and-paste them into the U-Boot connection each boot as needed.
6. Acquire an IP address
   - **If using DHCP** then disable the auto-boot feature which tries to download a kernel image from the DHCP server, and then acquire an address:
     ```
     => setenv autoload no
     => dhcp
     ```
     - Expected output:
       ```
       link up on port 0, speed 100, full duplex
       ```

```
BOOTP broadcast 1
BOOTP broadcast 2
DHCP client bound to address 192.168.0.113 (624 ms)
```

- **If using a static IP address configuration** (such as a direct Ethernet cable connection), set the IP address as follows:
  `=> setenv ipaddr 192.168.2.2`

  Where the host will be on the same subnet as the host for this connection.
- You must use a full Ethernet connection. UBoot does support Ethernet over USB because that is only done once Linux has booted on the target.
- Note that on some DHCP servers, the DHCP command can reset some environment variables in UBoot. By running the DHCP command *before* setting other variables this works around the issue.

7. Configure U-Boot environment variables as needed.
   - Configure the IP address of the server (host). Configure with your host's IP!
     `=> setenv serverip 192.168.2.1`
   - Configure the full path of the root folder storing the TFTP files (change `user_name` to be your user name on your host pc!):
     `=> setenv tftproot /home/user_name/cmpt433/public/`
   - Configure the file name of the Linux kernel image and the device tree file:
     ```
     => setenv bootfile ${tftproot}4.4.95-bfraser-bone19.zImage
     => setenv fdtfile ${tftproot}am335x-bonegreen.dtb
     ```
8. Test your target's ability to connect to the server using `ping`:
   ```
   => ping ${serverip}
   link up on port 0, speed 100, full duplex
   Using cpsw device
   host 192.168.2.1 is alive
   ```
   - If unable to ping (command stops after second line and seems to hang), it means that your network connection between the server and the target is incorrect. Double check the Supported Connections section (3.1) for details.
9. Download the device tree and kernel via TFTP. Should see '#'s printed to screen as it downloads.
   ```
   => tftp ${loadaddr} ${bootfile}
   => tftp ${fdtaddr} ${fdtfile}
   ```
10. Boot the image.
    - Set the kernel's "command line arguments" which are passed to the kernel when it boots. Note this is shown on two lines, but should be typed in on one[3]:
      ```
      => setenv bootargs console=ttyO0,115200n8 root=/dev/mmcblk1p1 \
                  ro rootfstype=ext4 rootwait
      ```
      Here it is on one line (small!) to make copy-paste easy:
      `=> setenv bootargs console=ttyO0,115200n8 root=/dev/mmcblk1p1 ro rootfstype=ext4 rootwait`
    - Boot the zImage you have downloaded. This boots the Linux kernel you compiled!
      `=> bootz ${loadaddr} - ${fdtaddr}`

3 *Note: For kernels 4.4.x and higher, it's mmcblk**1**p1; older kernels may use mmcblk**0**p1. See https://groups.google.com/forum/#!topic/beagleboard/svbS36EDo4A*

11. You can combine multiple U-Boot commands into one line by separating them with a semicolon (';'). Here is the full (one line!) command that I use with DHCP (server at 192.168.0.133):

```
setenv autoload no;dhcp;setenv serverip 192.168.0.133;setenv tftproot
/home/brian/cmpt433/public/;setenv bootfile ${tftproot}4.4.95-bfraser-
bone19.zImage;setenv fdtfile ${tftproot}am335x-bonegreen.dtb;tftp $
{loadaddr} ${bootfile};tftp ${fdtaddr} ${fdtfile};setenv bootargs
console=ttyO0,115200n8 root=/dev/mmcblk1p1 ro rootfstype=ext4
rootwait;bootz ${loadaddr} - ${fdtaddr};
```

12. Once the kernel boots, it should then load the root file system (RFS) installed in the target's eMMC. You should see your normal login prompt (customized in Assignment 1). Log-in via the user name `root`, no password required.
    - Note that many of the drivers will not have loaded, so don’t expect the board to function well at the moment. For example, the cape manager, GPIO, I2C, ... all may not work with this kernel version (yet).

13. Full U-Boot interaction via DHCP:

```
=> setenv autoload no
=> dhcp
link up on port 0, speed 100, full duplex
BOOTP broadcast 1
DHCP client bound to address 192.168.0.113 (299 ms)
=> set serverip 192.168.0.133
=> setenv tftproot /home/brian/cmpt433/public/
=> setenv bootfile ${tftproot}4.4.95-bfraser-bone19.zImage
=> setenv fdtfile ${tftproot}am335x-bonegreen.dtb
=> ping $serverip
link up on port 0, speed 100, full duplex
Using cpsw device
host 192.168.0.133 is alive
=> tftp ${loadaddr} ${bootfile}
link up on port 0, speed 100, full duplex
Using cpsw device
TFTP from server 192.168.0.133; our IP address is 192.168.0.113
Filename '/home/brian/cmpt433/public/4.4.95-bfraser-bone19.zImage'.
Load address: 0x82000000
Loading: *#################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ##################################################
         1.7 MiB/s
done
Bytes transferred = 7398968 (70e638 hex)
=> tftp ${fdtaddr} ${fdtfile}
link up on port 0, speed 100, full duplex
Using cpsw device
TFTP from server 192.168.0.133; our IP address is 192.168.0.113
Filename '/home/brian/cmpt433/public/am335x-bonegreen.dtb'.
Load address: 0x88000000
Loading: *####
         1.6 MiB/s
done
Bytes transferred = 50371 (c4c3 hex)
=> setenv bootargs console=ttyO0,115200n8 root=/dev/mmcblk1p1 ro rootfstype=ext4 rootwait
=> bootz ${loadaddr} - ${fdtaddr}
Kernel image @ 0x82000000 [ 0x000000 - 0x70e638 ]
## Flattened Device Tree blob at 88000000
   Booting using the fdt blob at 0x88000000
   Loading Device Tree to 8fff0000, end 8ffff4c2 ... OK

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0

< -- SNIP: Many lines omitted to fit -- >
Started LSB: Apache2 web server.

beaglebone login: root
Last login: Mon Oct 30 04:11:06 UTC 2017 on ttyS0
Linux beaglebone 4.4.95-bfraser-bone19 #1 Sun Oct 29 20:59:19 PDT 2017 armv7l
root@beaglebone:~# uname -a
Linux beaglebone 4.4.95-bfraser-bone19 #1 Sun Oct 29 20:59:19 PDT 2017 armv7l GNU/Linux
root@beaglebone:~#
```

- **There may be up to a 60s pause with no output when your target is booting; be patient!**

14. At the Linux command prompt, verify the correct kernel version:
```
# uname -r
4.4.95-bfraser-bone19
```
    - If you don't have the correct version, then you have likely booted the kernel stored on the board instead of downloading a new kernel. This may happen when you reboot the board and don't enter U-Boot. If so, realize that these steps must be done each time you reboot the board in order to load the custom kernel.
      It's OK if the last minor number of the kernel version and the user ID are different.
15. Limited access
    - When a new kernel is loaded, it has a different kernel version which makes many of the drivers on the BeagleBone fail to load. Hence, it is likely your system will now have no Ethernet over USB, cape manager, or access to hardware/LEDs, and so on.
    - Note that when you build the kernel, it produces a set of drivers which can be deployed to the board to enable many things, Ethernet over USB included. See section 5.1 (page 19) for how to copy over the drivers for this kernel, and hence to reactivate Ethernet over USB.
16. Serial Connection Troubleshooting:
    - Ensure that you are using the correct command to launch `screen`. Make sure you include the speed as it may work only intermittently if you do not:
      ```
      $ sudo screen /dev/ttyUSB0 115200
      ```
    - If your serial port stops working in the VM, disconnect it from the VM (in software) and reconnect. If it is unable to reconnect then plug it into a different USB. If all else fails, power down VM and VM software (launcher) and kill `vboxsvc.exe` (or restart host OS).
    - Or, use a serial terminal in Windows such as Putty or Tera Term (linked on course website).
17. Troubleshooting:
    - If you are unable to start a download, double check that the target has a correct IP address. You may need to re-run DHCP each time the board re-boots if you are going to download an image via TFTP.
    - Test your connection to the server using the U-Boot `ping` command (see earlier directions). If using a direct Ethernet cable connection, see the Networking guide for setting the static IP address as needed.
    - While downloading, if you see T's, it means that the target is unable to reach the server.
      ```
      => tftp ${loadaddr} ${bootfile}
      link up on port 0, speed 100, full duplex
      Using cpsw device
      TFTP from server 192.168.0.123; our IP address is 192.168.0.138
      Filename '/home/brian/cmpt433/public/4.9.0-rc1-bone0.zImage'.
      Load address: 0x82000000
      Loading: T T T T T T T T T T T T T T T T T T T
      Retry count exceeded; starting again
      using musb-hdrc, OUT ep1out IN ep1in STATUS ep2in
      MAC c8:a0:30:aa:dd:a2
      HOST MAC de:ad:be:af:00:00
      RNDIS ready
      ERROR: The remote end did not respond in time.
      at drivers/usb/gadget/ether.c:2388/usb_eth_init()
      ```
        - Double check you have the `serverip` set correctly (via ping)

    - Check the host is running a TFTP server (earlier section in this guide).
- If you get the error that the file is not found during TFTP download, double check that the file is named correctly, that it is in your public directory, and that you have correctly set the `tftproot` in the UBoot statements.
- Do not encrypt your home folder which is being accessed by TFTP / NFS.
- Ensure you are not running any VPN software such as Hamachi. These can cause connectivity problems.
- When running the `bootz` command, if you see the error:
  `Error: unrecognized/unsupported machine ID (r1 = 0x00000e05)`
  Then you did not have a valid device tree file loaded. Ensure you are downloading the correct `.dtb` file, and passing its address as an argument to the `bootz` command.
- The cape-manager is only found in kernel versions 3.8, and 4.1 (and above).
- You will likely not be able to access much hardware with the kernel as most drivers will fail to load. See section 5.1 (page 19) for how to copy over the drivers for this kernel, and hence to reactivate Ethernet and Ethernet over USB.

4. Creating a Test Driver

4.1 Cross Compiling a Driver

This section will create the driver in the `~/cmpt433/work/driver_demo/` directory of the host, cross-compile it and deploy it to the target.

1. Create a directory for the driver source code:
   ```
   $ cd ~/cmpt433/work/
   $ mkdir driver_demo/
   $ cd driver_demo/
   ```
2. Create a directory for the compiled drivers:
   ```
   $ cd ~/cmpt433/public
   $ mkdir drivers
   ```
3. Create `testdriver.c` in `~/cmpt433/work/driver_demo/` directory with the following contents:
   ```
   // Example test driver:
   #include <linux/module.h>

   static int __init testdriver_init(void)
   {
         printk(KERN_INFO "----> My test driver init()\n");
         return 0;
   }

   static void __exit testdriver_exit(void)
   {
         printk(KERN_INFO "<---- My test driver exit().\n");
   }

   // Link our init/exit functions into the kernel's code.
   module_init(testdriver_init);
   module_exit(testdriver_exit);

   // Information about this module:
   MODULE_AUTHOR("Your Name Here");
   MODULE_DESCRIPTION("A simple test driver");
   MODULE_LICENSE("GPL");        // Important to leave as GPL.
   ```
4. Change the module information to include your name (the `MODULE_AUTHOR()`).
5. **After the `#include`, add the following line:**
   ```
   #error Are we building this file?
   ```
   - When compiled, this will generate an error which will prove you are building this file.
   - Later, once you are sure your file is being compiled correctly, you'll remove this line, but for the moment it will serve as our test that the process is working.

6. Create `Makefile` in `~/cmpt433/work/driver_demo/` with the following contents:

```
# Makefile for driver
# Derived from:
#   http://www.opensourceforu.com/2010/12/writing-your-first-linux-driver/
# with some settings from Robert Nelson's BBB kernel build script

# if KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq (${KERNELRELEASE},)
      obj-m := testdriver.o

# Otherwise we were called directly from the command line.
# Invoke the kernel build system.
else
      KERNEL_SOURCE := ${HOME}/cmpt433/work/bb-kernel/KERNEL/
      PWD := $(shell pwd)
      # Linux kernel 4.4 (which has cape manager support)
      CC=${HOME}/cmpt433/work/bb-kernel/dl/gcc-linaro-5.4.1-2017.05-
x86_64_arm-linux-gnueabihf/bin/arm-linux-gnueabihf-
      BUILD=bone19
      CORES=4
      image=zImage
      PUBLIC_DRIVER_PWD=~/cmpt433/public/drivers

default:
      # Trigger kernel build for this module
      ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} -j${CORES} ARCH=arm \
            LOCALVERSION=-${BUILD} CROSS_COMPILE=${CC} ${address} \
            ${image} modules
      # copy result to public folder
      cp *.ko ${PUBLIC_DRIVER_PWD}

clean:
      ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
endif
```

- **Important:**
  - The file name must be `Makefile` (case sensitive!). Do not name it "makefile".
  - The “CC” statement is one line, no spaces or line-feeds.
- Read the comments to understand what is going on. The basics are:
  - Make will execute the “`else`” portion of the main “`ifneq`” statement. This sets up parameters for the main Linux kernel build system, and then invokes the Linux kernel build system asking it to build this folder.
  - The Linux kernel build then starts running (via Make) on this device driver's folder.
  - The “`then`” portion of the main “`ifneq`” then gets executed and actually builds the driver.
- Note that this `Makefile` alone is not sufficient to build your driver on its own; it leverages the general kernel build system.

7. Prove your `Makefile` works to build your code by having the `#error` directive break the build:

```
~/cmpt433/public/driver_demo$ make
# Trigger kernel build for this module
make -C /home/brian/cmpt433/work/bb-kernel/KERNEL/
SUBDIRS=/home/brian/cmpt433/public/driver_demo -j4 ARCH=arm \
      LOCALVERSION=-bone19 CROSS_COMPILE=/home/brian/cmpt433/work/bb-kernel/
dl/gcc-linaro-5.4.1-2017.05-x86_64_arm-linux-gnueabihf/bin/arm-linux-
gnueabihf-  \
      zImage modules
make[1]: Entering directory '/home/brian/cmpt433/work/bb-kernel/KERNEL'
  Kernel: arch/arm/boot/Image is ready
  Kernel: arch/arm/boot/Image is ready
  LD      /home/brian/cmpt433/public/driver_demo/built-in.o
  CC [M]  /home/brian/cmpt433/public/driver_demo/testdriver.o
  Kernel: arch/arm/boot/zImage is ready
/home/brian/cmpt433/public/driver_demo/testdriver.c:3:2: error: #error Are
we building this file?
 #error Are we building this file?
  ^
scripts/Makefile.build:264: recipe for target
'/home/brian/cmpt433/public/driver_demo/testdriver.o' failed
<... some messages omitted...>
~/cmpt433/public/driver_demo$
```

   - You should see the "`Are we building this file?`" error. If so, the comment out the `#error` and continue; otherwise, double check your `Makefile` and kernel build folder are correct.

8. Build the driver by running make:

```
~/cmpt433/public/driver_demo$ make
# Trigger kernel build for this module
make -C /home/brian/cmpt433/work/bb-kernel/KERNEL/
SUBDIRS=/home/brian/cmpt433/public/driver_demo -j4 ARCH=arm \
      LOCALVERSION=-bone19 CROSS_COMPILE=/home/brian/cmpt433/work/bb-kernel/
dl/gcc-linaro-5.4.1-2017.05-x86_64_arm-linux-gnueabihf/bin/arm-linux-
gnueabihf-  \
      zImage modules
make[1]: Entering directory '/home/brian/cmpt433/work/bb-kernel/KERNEL'
  Kernel: arch/arm/boot/Image is ready
  Kernel: arch/arm/boot/Image is ready
  CC [M]  /home/brian/cmpt433/public/driver_demo/testdriver.o
  Kernel: arch/arm/boot/zImage is ready
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/brian/cmpt433/public/driver_demo/testdriver.mod.o
  LD [M]  /home/brian/cmpt433/public/driver_demo/testdriver.ko
make[1]: Leaving directory '/home/brian/cmpt433/work/bb-kernel/KERNEL'
# copy result to public folder
cp *.ko ~/cmpt433/public/drivers
~/cmpt433/public/driver_demo$
```

9. Check that the .ko file was correctly copied to the `~/cmpt433/public/drivers/` folder:

```
$ ls ~/cmpt433/public/drivers/
```

4.2 OPTIONAL: Natively Compiling Drivers

Instead of cross-compiling the drivers from your PC, you can instead copy the Makefile and your .c source files to your target and build on the BeagleBone directly. This is supported native compilation on the pre-installed kernel because it has the kernel headers already on the board; it likely won't work with a custom kernel you install. Use this approach if you are having problems getting the custom kernel running on the board in some settings. It will compile under the pre-installed kernel, for the pre-installed kernel.

1. Copy your .c driver source code to a directory on your target, such as `~/mydriver/`
2. In the folder with your .c code, create a Makefile with the following contents:

```
# Makefile for driver: native-compile

obj-m := testdriver.o

all:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

3. Driver should be ready to go:

```
# modinfo ./testdriver.ko
# insmod ./testdriver.ko
```

4. Discussion of Folders
   - You could do all your development on the target; however, it makes a lot of sense to use a powerful development environment on the host and share the files to the target via NFS.
   - However, you must manage directory and file permissions so that the target can create new files in the folder. You'll need the NFS folder containing your .c files to be writable by the target, such as:
     $ chmod a+rwx <native-compile-folder-via-NFS>
   - Be very careful to setup a development process where you can efficiently edit your code (say on your host through an editor) and then build on the target. Be sure that you don't lose any of your files or changes due to having multiple copies and getting confused about which one to edit or check into Git.
   - You may want to have your source code in some Git folder on your PC, and then use a script to copy the .c and Makefile to the NFS folder for the target to access.
5. Troubleshooting:
   - If you get any file permission errors, try copying the .c and Makefile to a folder on the target like `~/mydriver/` and re-run make in that folder.

5. Working with Drivers

To use the driver, it must be available on the target board at runtime.

1. Either mount via NFS a shared folder containing the .ko device driver, or copy it onto the target.
   - Note: You are now working across the Ethernet, not Ethernet-over-USB. Therefore you may need to update the NFS share settings on your host, and the NFS mount command used on the target. See NFS guide for details.
2. On the target, change to the directory containing the .ko file. If mounted via NFS:
   ```
   # cd /mnt/remote/drivers/
   ```
3. List existing modules loaded on the target:
   ```
   # lsmod
   ```
   - You will not see the `testdriver` listed.
4. Find information about your compiled module using:
   ```
   # modinfo ./testdriver.ko
   filename:       /mnt/remote/drivers/./testdriver.ko
   license:        GPL
   description:    A simple test driver
   author:         Brian Fraser
   depends:
   vermagic:       4.4.95-bfraser-bone19 mod_unload modversions ARMv7 thumb2 p2v8
   ```
   - The “vermagic” field shows the kernel version your module is targeting.
   - You can run `modinfo` on any machine to see the information about a module, even if that module targets a different architecture.
5. Ensure your booted kernel version matches the vermagic field of the driver:
   ```
   # uname -r
   4.4.95-bfraser-bone19
   ```
   - Any difference between this string and the starting word in the vermagic will cause the driver to fail to load.
6. Load the driver:
   ```
   # insmod testdriver.ko
   ----> My test driver init()
   ```
   - You may not see any output to the screen as the driver only prints to the kernel log. To see this, you may need to execute `dmesg`:
     ```
     # dmesg | tail -1
     [  938.788651] ----> My test driver init()
     ```
   - If you see "`insmod: cannot insert 'testdriver.ko': invalid module format`", it likely means that your target's current kernel was built with a different version string than your host is currently building. Either rebuild and download the kernel, or change the host to build the same version as the target (found by running `uname -r`).
7. View loaded modules on target:
   ```
   # lsmod
   ```
   - You should now see the `testdriver` loaded.
8. Remove on target (output may appear only in `dmesg`);
   ```
   # rmmod testdriver
   <---- My test driver exit().
   ```
9. Troubleshooting:

- If you are unable to mount via NFS, double check:
  - Your mount script is targeting the correct IP address. If you are running on actual Ethernet, you may be at address 192.168.2.2 (instead of ...7.2)! Target the correct server IP.
  - Ensure your NFS settings have been correctly updated to permit connections from the 192.168.2.0 subnet.
- If you get an error that the versions are incomparable (or an invalid file format) consult Section 6.
- If the Ethernet works in UBoot to boot the custom kernel, but does not stay working under Linux, then try:
  - Reboot to the stock kernel version and mount NFS.
  - Install the drivers for your custom kernel by following the directions in the next section.
  - Reboot and re-download and run your custom kernel. Now when it finishes booting it should load the Ethernet over USB driver and get the IP address 192.168.7.2 as before.
- If unable to load Ethernet support at all, you can still write and test a driver using the "Natively Compiling Drivers" directions in the previous section.

5.1 Coping Modules to RFS

You can copy the kernel modules (drivers) which were build when you built your custom kernel to the root file system so that these drivers can be loaded at run-time if a device is connected which requires a driver. Without copying these drivers over, when you boot to your custom kernel your target will not have many device drivers up and running (for example, support for USB devices).

1. On the **host**, copy the modules:
   ```
   $ cd ~/cmpt433/work/bb-kernel/deploy/
   $ cp 4.4.95-bfraser-bone19-modules.tar.gz  ~/cmpt433/public/
   ```
   - Change the user ID in the file name to match your files (i.e., it won't be "-bfraser")!
2. On the **target**, extract the archive of modules from within the root directory (/):
   ```
   # cd /
   # tar -zxvf /mnt/remote/4.4.95-bfraser-bone19-modules.tar.gz
   ```
   - You may get warnings about time stamps being in the future; these can be ignored.
   - This will likely take around 180k on the eMMC
3. Check the modules installed:
   ```
   # ls /lib/modules/
   ```
   - Output:
     ```
     4.1.15-ti-rt-r43/  4.4.95-bfraser-bone19/
     ```
     Note that your initial folder name may vary depending on the kernel version your board initially ran.
   - The `tar` command also copies some files into `/lib/firmware`
4. You now have all the device drivers available for your newly compiled kernel! Connecting a device that requires a device driver to be loaded should now work.

6. Version Incompatibilities

Linux is very strict about what kernel modules (.ko files) it will load. **Specifically, it enforces that the module has the identical version string as the kernel.** The version string is also called version-magic. This is necessary because a driver is linked against a specific kernel version's headers. Any change to these headers can make the drivers perform incorrectly. This section guides you through identifying the versions of a .ko file, and of the Linux kernel, and presents some strategies to get things working.

6.1 Understanding the Problem

1. On the **target**, identify the kernel version you are running. Below is shown the output for a version of the BeagleBone kernel (version installed on your board may be different):
   ```
   # uname -r
   4.1.15-ti-rt-r43
   ```
2. On either the host or the target, find the version of the kernel module you are building:
   - In the folder containing your .ko file, run:
     ```
     # modinfo testdriver.ko
     filename:       /mnt/remote/drivers/testdriver.ko
     license:        GPL
     description:    A simple test driver
     author:         Dr. Evil
     depends:
     vermagic:       4.4.95-bfraser-bone19 mod_unload modversions ARMv7 thumb2 p2v8
     ```
   - Here we see that the "vermagic" is `4.4.95-bfraser-bone19`
   - Sometimes the target will not have the `modinfo` tool and so the host's tool must be used.
3. In the case shown above the version of the kernel does not match the version of the module. The error when attempting to load this module on the incompatibility kernel is:
   ```
   # insmod testdriver.ko
   Error: could not insert module testdriver.ko: Invalid module format
   ```
   - Running `dmesg` shows additional information:
     ```
     # dmesg | tail -1
     [ 57.674358] testdriver: disagrees about version of symbol module_layout
     ```
4. Trouble shooting:
   - If you are working on more than one computer (such as in the lab, or with group members), ensure that all your (or group) build setups are building to the same kernel version, with the same version string. This will reduce the problem of incompatible versions.

6.2 Resolving the Problem

There are a number of options to resolve the above incompatibility:

1. **Rebuild Kernel and Module:**
   - Rebuild both the kernel and the modules (device drivers).
     - If using the scripts described here, run:
       `$ ~/cmpt433/work/bb-kernel/build_kernel.sh`
       and run `make` on your device driver folder.
     - If using the kernel build scripts, run:
       ```
       $ make kernel
       $ make modules
       ```
   - Use U-Boot to re-download the kernel to the target.
   - On the host, copy the newly-built kernel module (.ko) to your NFS directory and then extract them into the `/lib/modules/` directory on the target.
   - Now both the kernel and the module should be the latest version which you are building.
   - If you have any difficulties, re-check that the kernel version and the module versions match. If they don't check out which one did not update correctly. Use "`make menuconfig`" on the Linux kernel to check what version you are trying to build.
   - This is the preferred option because it gets the target fully in synch with the build setup on your host and means that other modules will build and load correctly..
2. **Change the version of the kernel configuration you are building:**
   - Download the exact same version of the kernel that is executing on the target.
   - Use "`make menuconfig`" on the kernel to change the kernel version to match the version of the kernel currently installed on your target.
   - Then, rebuild the module (`make` on your device driver folder) and reload the newly rebuilt .ko module.
   - Since you changed the version to match the kernel you have installed, it should allow the module to load.
3. **Find files that match:**
   - Find a version of the .ko module that matches the version of the kernel that you are running on the target.
   - You might want to look in the /lib directory on the target, and use `modinfo` to check the version magic number.