Create Linux Kernel Driver for Installed BeagleBone System

Tested under host Debian 11 (bullseye) with target kernel 5.10.168

by Brian Fraser
Last update: Mar 25, 2024

This document guides the user through

1. Downloading the version of the Linux kernel running on your BeagleBone and compiling it.
2. Creating and compiling a new new Linux driver.
3. Loading and unloading the driver into your existing BeagleBone system.

Table of Contents

1. Building the Kernel....2
2. Creating a Test Driver....4
  2.1 Cross Compiling a Driver....4
  2.2 OPTIONAL: Natively Compiling Drivers on Target....7
3. Working with Drivers....9
4. Version Incompatibilities....11
  4.1 Understanding the Problem....11
  4.2 Resolving the Problem....12

Formatting

1. Commands starting with `(host)$` are Linux console commands on the host PC:
   `(host)$ echo "Hello PC world"`
2. Commands starting with `(bbg)$` are Linux console commands on the target board:
   `(bbg)$ echo "Hello Target world"`

Document History

- Nov 2: Corrected typo in makefile step 4.1.6.
- Nov 13: Added `iptables` command to resolve `tftp` blocking issue.
- Mar 1, 2021: Updated to Ubuntu 20.04; clarified directions
- Mar 17, 2021: Updated language around kernel make file; corrected paths in examples.
- Feb 23, 2023: Changed to targeting the current kernel vs running a new one.
- Mar 19, 2024: Replaced `jenkins_build.sh` with straight commands.
- Mar 25, 2004: Updated guide to Kernel version -r75

1. Building the Kernel[1]

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 Debian/Ubuntu) otherwise permissions and symbolic links will not work correctly. For me, it took 3 hours on USB.

1. Clone the BeagleBone build scripts from GitHub[2]:
   ```
   (host)$ cd ~/cmpt433/work
   (host)$ git clone https://github.com/beagleboard/linux.git
   ```
   - This step may take quite a while because it downloads 4GB of data.
2. Find the tag which matches your BeagleBone.
   On target, find your version:
   ```
   (bbg)$ uname -r
   5.10.168-ti-r75
   ```
   On host, find the tag for the correct version:
   ```
   (host)$ cd ~/cmpt433/work/linux/
   (host)$ git tag | grep '5.10.168'
   ```
   - Pick the tag which matches your board.
3. Checkout the scripts to download and build a specific kernel version:
   ```
   (host)$ git checkout tags/5.10.168-ti-r75
   ```
   - If you are unable to checkout due to an error about uncommitted changes, try:
     ```
     (host)$ git stash
     ```
     This will "stash" the changed files for later retrieval (if desired).
   - OK to ignore a warning about "detached HEAD"
4. Install needed utilities to build the kernel[3]:
   ```
   (host)$ sudo apt install flex bison rsync gettext libmpc-dev lz4
   (host)$ sudo apt install lzop lzma libncurses5-dev:native bc cpio
   (host)$ sudo apt install build-essential libssl-dev:native
   ```

1 An alternative is to build a full fresh kernel rather than matching one on the official BeagleBone GitHub repo. For example, I have often used a build script maintained by Robert C. Nelson:
(host)$ `git clone https://github.com/RobertCNelson/bb-kernel.git`
`(host)$ cd bb-kernel`
`(host)$ git checkout tags/6.2-rc6-bone6` # find options with `git tag`
`(host)$ ./build_kernel.sh`

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

3 Other tools to install include:
(host)$ sudo apt-get install u-boot-tools libncurses5-dev:amd64 fakeroot libssl-dev:amd64

5. Run the following build commands[4]. It may take a while! Took ~**10 minutes** on my home desktop VM.

```
export CC=/usr/bin/arm-linux-gnueabihf-

make ARCH=arm CROSS_COMPILE=${CC} clean
make ARCH=arm CROSS_COMPILE=${CC} bb.org_defconfig

echo "make -j12 ARCH=arm KBUILD_DEBARCH=armhf CROSS_COMPILE=${CC} bindeb-pkg"
make -j12 ARCH=arm KBUILD_DEBARCH=armhf KDEB_PKGVERSION=1xross CROSS_COMPILE=${CC} bindeb-pkg
```

   - These commands may fail and ask you to install additional packages, or try to use a tool that is not yet installed. Install the packages, then retry the command.
   - If a failure occurs, correct the problem and re-run the command.

6. When the commands finishes, output may look like:

```
...
  HDRINST usr/include/asm/ipcbuf.h
  HDRINST usr/include/asm/termios.h
  HDRINST usr/include/asm/sockios.h
  HDRINST usr/include/asm/sembuf.h
  HDRINST usr/include/asm/poll.h
  INSTALL debian/linux-libc-dev/usr/include
dpkg-deb: building package 'linux-libc-dev' in '../linux-libc-dev_1xross_armhf.deb'.
dpkg-deb: building package 'linux-image-5.10.168' in '../linux-image-5.10.168_1xross_armhf.deb'.
 dpkg-genbuildinfo --build=binary
 dpkg-genchanges --build=binary >../linux-5.10.168_1xross_armhf.changes
dpkg-genchanges: info: binary-only upload (no source code included)
 dpkg-source --after-build .
dpkg-buildpackage: info: binary-only upload (no source included)
brian@PC-debian:~/cmpt433/work/linux$
```

7. Troubleshooting
   - If the build fails, read the error messages carefully; it may be missing some tools. Try installing the missing tools and restarting the build.

4 These commands used to be in jenkins_build.sh file.

2. Creating a Test Driver

2.1 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 compiled drivers:
   `(host)$` **`mkdir -p ~/cmpt433/public/drivers`**
2. Create a directory for the driver source code:
   `(host)$` **`mkdir -p ~/cmpt433/work/driver_demo`**
   `(host)$` **`cd ~/cmpt433/work/driver_demo`**
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
# modified by Brian Fraser

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

# Otherwise we were called directly from the command line.
# Invoke the kernel build system.
else
      KERNEL_SOURCE := ~/cmpt433/work/linux/
      PWD := $(shell pwd)
      CC=arm-linux-gnueabihf-

      CORES=4
      PUBLIC_DRIVER_PWD=~/cmpt433/public/drivers

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

all: default

clean:
      ${MAKE} -C ${KERNEL_SOURCE} M=${PWD} clean

endif
```

- **Important:**
  - The file name must be `Makefile` (case sensitive!); do *not* name it "makefile".
- Read the comments to understand what is going on. The basics are:
  - Make will evaluate the following statement as false:
    ```
    ifneq (${KERNELRELEASE},)
    ```
    so make will execute the "`else`" portion of the main "`ifneq`" statement. This sets up parameters for the main Linux kernel build system, and then invokes the Linux kernel build system asking it to build our current folder.
  - The Linux kernel build then starts running (via Make) on this device driver's folder.
  - The following statement is evaluated and will now be true:
    ```
    ifneq (${KERNELRELEASE},)
    ```
    It adds our driver(s) to the `obj-m` variable, which tells the kernel build what to build.
- 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/work/driver_demo$ make
# Trigger kernel build for this module
make -C ~/cmpt433/work/linux/ M=/home/brian/cmpt433/work/driver_demo -j4
ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-   modules
make[1]: Entering directory '/home/brian/cmpt433/work/linux'
  CC [M]  /home/brian/cmpt433/work/driver_demo/testdriver.o
/home/brian/cmpt433/work/driver_demo/testdriver.c:3:2: error: #error Are we
building this file?
    3 | #error Are we building this file?
      |  ^~~~~
make[2]: *** [scripts/Makefile.build:286:
/home/brian/cmpt433/work/driver_demo/testdriver.o] Error 1
make[1]: *** [Makefile:1835: /home/brian/cmpt433/work/driver_demo] Error 2
make[1]: Leaving directory '/home/brian/cmpt433/work/linux'
make: *** [Makefile:23: default] Error 2
~/cmpt433/work/driver_demo$
```

   - You should see the "`Are we building this file?`" error. If so, great! You are building the file you expect!

     Comment out the `#error` statement in your `testdriver.c` and continue; otherwise, double check your `Makefile` and kernel build folder are correct.
   - You may get a warning about `SUBDIRS` in the Linux kernel makefile. It seems we can build fine even with this warning.

8. Once you have commented out the #error directive, build the driver by running `make`
   When successful, it will look like:

```
~/cmpt433/work/driver_demo$ make
# Trigger kernel build for this module
make -C ~/cmpt433/work/linux/ M=/home/brian/cmpt433/work/driver_demo -j4
ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-   modules
make[1]: Entering directory '/home/brian/cmpt433/work/linux'
  CC [M]  /home/brian/cmpt433/work/driver_demo/testdriver.o
  MODPOST /home/brian/cmpt433/work/driver_demo/Module.symvers
  LD [M]  /home/brian/cmpt433/work/driver_demo/testdriver.ko
make[1]: Leaving directory '/home/brian/cmpt433/work/linux'
#
# copy result to public folder
mkdir -p ~/cmpt433/public/drivers
cp *.ko ~/cmpt433/public/drivers
chmod a+x ~/cmpt433/public/drivers/*.sh
~/cmpt433/work/driver_demo$
```

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

```
(host)$ ls -l ~/cmpt433/public/drivers/
```

2.2 OPTIONAL: Natively Compiling Drivers on Target

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. Use this approach if you are having problems getting the kernel building on the host.

In this course, you are expected to cross-compile from the host. These steps here are a backup in case that fails; however, there may be a mark penalty.

These directions will compile your driver **under the pre-installed kernel, for the pre-installed kernel.**

1. Ensure your board has an internet connection:
   ```
   (bbg)$ ping google.ca
   ```
2. Your board will need the kernel headers for your version of the kernel. Install them:
   ```
   (bbg)$ sudo apt-get install linux-headers-`uname -r`
   ```
   - This will create the `/lib/modules/<kernel-version>/build/` folder.
3. Have your `.c` driver source code in a directory accessible from the target, such as `/mnt/remote/mydriver/`
4. 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
   ```
5. Build the driver
   ```
   (bbg)$ cd /mnt/remote/mydriver/
   (bbg)$ make
   make -C /lib/modules/5.10.168-ti-r75/build/ M=/mnt/remote/driver_demo
   modules
   make[1]: Entering directory '/usr/src/linux-headers-5.10.168-ti-r75'
     CC [M]  /mnt/remote/driver_demo/testdriver.o
     MODPOST /mnt/remote/driver_demo/Module.symvers
     CC [M]  /mnt/remote/driver_demo/testdriver.mod.o
     LD [M]  /mnt/remote/driver_demo/testdriver.ko
   make[1]: Leaving directory '/usr/src/linux-headers-5.10.168-ti-r75'
   ```
6. Driver should be ready to go:
   ```
   (bbg)$ modinfo ./testdriver.ko
   (bbg)$ sudo insmod ./testdriver.ko
   ```
7. 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.

     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. Or, better yet, work out how to do it with file system links (`man ln`)

8. 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.
   - If `make` fails to find the `/lib/modules/<kernel-version>/build/ folder`, then you likely need to install the kernel headers for your current specific version of the kernel. Find the version by running `uname -r`

3. 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.
   - If you are now working across the Ethernet (instead of not Ethernet-over-USB) 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, use:
   ```
   (bbg)$ cd /mnt/remote/drivers/
   ```
3. List existing modules loaded on the target:
   ```
   (bbg)$ lsmod
   ```
   - You will *not* see the `testdriver` listed.
4. Find information about your compiled module using:
   ```
   (bbg)$ modinfo ./testdriver.ko
   filename:       /mnt/remote/drivers/./testdriver.ko
   license:        GPL
   description:    A simple test driver
   author:         Your Name Here
   depends:
   name:           testdriver
   vermagic:       5.10.168 SMP preempt mod_unload modversions ARMv7 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:
   ```
   (bbg)$ uname -r
   5.10.168-ti-r75
   ```
   - Any difference between this string and the starting word in the vermagic will cause the driver to fail to load.
6. Load the driver:
   ```
   (bbg)$ sudo 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`:
     ```
     (bbg)$ 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.

     Try repeating the steps to build the Linux kernel you downloaded from GitHub. Ensure you checkout the correct tag and build it. You can find the info on the module by running:
     `modinfo testdriver.ko`
   - If you see a message "loading out-of-tree module taints kernel", you may ignore this.
7. View loaded modules on target:
   ```
   (bbg)$ lsmod
   ```

- You should now see the `testdriver` loaded.

8. Remove on target (output may appear only in `dmesg`);

```
(bbg)$ sudo rmmod testdriver
<---- My test driver exit().
```

9. Troubleshooting:
   - If commands like `insmod` and `rmmod` fail with “Operation not permitted”, then ensure you are running the command as root (i.e., using `sudo`).
   - If you get an error that the versions are incomparable (or an invalid file format) consult Section 4.
   - If your driver will not correctly load and register (likely as a misc driver) then you may need to reboot your board and try again.

4. 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.

4.1 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):
   ```
   (bbg)$ uname -r
   5.10.168-ti-r75
   ```
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:
     ```
     (bbg)$ modinfo testdriver.ko
     filename:       /mnt/remote/drivers/testdriver.ko
     license:        GPL
     description:    A simple test driver
     author:         Dr. Evil
     depends:
     vermagic:       5.3.7 mod_unload modversions ARMv7 thumb2 p2v8
     ```
   - In this specific example, we see that the "vermagic" is `5.3.7`
   - 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:
   ```
   (bbg)$ insmod testdriver.ko
   Error: could not insert module testdriver.ko: Invalid module format
   ```
   - Running `dmesg` shows additional information:
     ```
     (bbg)$ 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.

4.2 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:
       `(host)$ ~/cmpt433/work/linux/jenkins_build.sh`
       and run `make` on your device driver folder.
     - If using the kernel build scripts, run:
       `(host)$ **make kernel**`
       `(host)$ **make modules**`
   - 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.
   - 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. **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.