

Hands-On Exercises for Embedded Linux

v. 2019.10

WARNING:

The order of the exercises does not always follow the same order of the explanations in the slides. When carrying out the exercises, carefully follow the exercise requirements. Do **NOT** blindly type the commands or code as found in the slides. Read every exercise **in its entirety** before carrying out the instructions.



These exercises are made available to you under a Creative Commons Share-Alike 3.0 license. The full terms of this license are here:

<https://creativecommons.org/licenses/by-sa/3.0/>

Attribution requirements and misc.:

- This page must remain as-is in this specific location (page #2), (almost) everything else you are free to change.
- You are forbidden from removing our name or logo.
- Use of figures in other documents must feature the below “Originals at” URL immediately under that figure and the below copyright notice where appropriate.
- You are free to fill in the space in the below “Delivered and/or customized by” section as you see fit.

(C) Copyright 2003-2019, Opersys inc.

These exercises created by: Karim Yaghmour

Originals at: www.opersys.com/training/

Delivered and/or customized by:

Software components versions

Hardware:

BeagleBone White – has serial integrated into USB

Class Drive:

Location to be provided in class

Files:

Prebuilt SD card image:	el-image-bb-191007.img.xz
Workspace:	bbone-white-3.0.0-191014.tar.bz2

Linux Kernel:

Version:	5.3.2
Patch:	NONE
Configuration for target:	multi_v7_defconfig
Kernel image:	arch/arm/boot/zImage
Device tree blob:	arch/arm/boot/dts/am335x-bone.dtb

Bootloader:

Package:	U-Boot
Version:	u-boot-2019.07
Patch:	NONE
Configuration for target:	am335x_evm_defconfig
Images to use on target:	u-boot.img and MLO

Toolchain:

Prefix (Ubuntu ARM toolchain):	arm-linux-gnueabi-
Prefix (Buildroot uClibc toolchain):	arm-linux-

TARGET SERIAL NUMBER:

“Linux” basics:

1) Use the following commands to install the following packages:

```
$ sudo apt-get install libc6-armel-cross libc6-dev-armel-cross \  
> binutils-arm-linux-gnueabi libncurses5-dev  
$ sudo apt-get install gcc-arm-linux-gnueabi g++-arm-linux-gnueabi  
$ sudo apt install bison flex  
$ sudo apt install gnupg2  
$ sudo apt install libssl-dev
```

2) Configure your PC to connect to beaglebone. You'll need to make sure you are part of the “dialout” group and that you can see /dev/ttyUSB0 on your host.

3) Download prebuilt SD card image and use the “dd” command to put it on the SD card and boot with it.

4) Use “picocom” to shell into the BeagleBone.

5) Create a local-LAN configuration for your Ethernet card to allow connection to the target using a cross-over Ethernet cable (host: 192.168.202.100; target: 192.168.202.79).

Example Ethernet card configuration:

```
IPADDR=192.168.202.100  
GATEWAY=192.168.202.79  
NETWORK=192.168.202.0  
NETMASK=255.255.255.0  
BROADCAST=192.168.202.255
```

Development Workspace:

1) Download the development workspace image and put it in your home directory.

2) Extract the development workspace

3) Make sure you are able to run the “devenv” setup script to set up your environment variables.

4) Repartition the SD card using the fdisk command for booting the beaglebone

5) Put the images in “images/boot-part” on your bootable partition.

6) Boot with the newly reformatted card – the kernel should panic because it doesn't have a root filesystem. That's expected.

Kernel Basics:

0) Apply any necessary kernel patch

1) Configure the kernel for the target. Make sure that support for config.gz in /proc is enabled

("Kernel .config support").

- 2) Build the kernel
- 3) Install the kernel
- 4) Build the kernel modules
- 5) Install the kernel modules
- 6) Build the device trees
- 7) Install the beaglebone device tree
- 8) Overwrite the default kernel and DTB on the SD card with the one you have just built and boot with them.
- 9) Check that the kernel booting is your own in its early messages.

Bootloader:

0) Apply any necessary U-Boot patch

1) Build U-Boot using this command:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- -j8 am335x_evm_defconfig  
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- -j8
```

2) Install U-Boot:

```
$ cp MLO ${PRJROOT}/images
```

```
$ cp u-boot.img ${PRJROOT}/images
```

3) Copy the u-boot images to the VFAT partition:

```
$ cd ${PRJROOT}/images
```

```
$ cp MLO /media/karim/....
```

```
$ cp u-boot.img /media/karim/....
```

```
$ cp uEnv.txt /media/karim/....
```

- 4) Boot with U-Boot and try out the online help
- 5) Print U-Boot's environment variables
- 6) Configure your host for serving DHCP requests
- 7) Use "bootp" in U-Boot to get config from DHCP server
- 8) Configure your host for serving TFTP requests
- 9) Use TFTP to download image to target
- 10) Boot with image
- 11) Create environment variables in U-Boot's environment to automate the loading of the kernel and the DTB to boot from them using tftp.

Root filesystem:

- 1) Create essential rootfs directories
- 2) Copy glibc libraries to target's rootfs and strip them. NOTE: contrary to the slides, the libraries are in /usr/arm-linux-gnueabi/lib/
- 3) Copy kernel modules to target's rootfs
- 4) Make sure you have /dev, /proc and /sys in your rootfs
- 5) Configure BusyBox using the “make menuconfig” command
- 6) Build and install BusyBox
- 7) Create initialization scripts for BusyBox init. Don't forget to set /etc/init.d/rcS to allow for it to execute (chmod 755 /etc/init.d/rcS). Contrary to the examples in the slides, do **not** put an entry in /etc/fstab for NFS, and do **not** put an entry for “custom-app” in /etc/inittab.
- 8) Copy your rootfs into the ext4 partition on the device – you'll have to see what directory it's mounted to.
\$ cd \${PRJROOT}/rootfs
\$ cp -a \${PRJROOT}/rootfs/* /media/karim/...
- 9) Boot the new SD card, you should now have a fully-functional embedded Linux system

Filesystem Types:

- 0) Install the MTD utilities for your host
- 1) Build and install the cramfs utilities
- 2) Create a CRAMFS image of your target's root filesystem
- 3) Install the romfs utilities
- 3) Create a ROMFS image of your target's root filesystem
- 4) Install the squashfs utilities
- 5) Create a squashfs image of your target's root filesystem
- 6) Create a UBIFS image of your target's root filesystem
- 7) Create a JFFS2 image of your target's root filesystem
- 8) Create a RAM disk image of your target's root filesystem
- 9) Create an initramfs images of your target's root filesystem
- 10) Compare the filesystem image sizes
- 11) Configure your host and your target so that the target's configuration is obtained at boot time via DHCP and the rootfs is mounted on NFS.

Device Drivers – Set #1:

Write a dynamically loadable device driver for the target that:

- a) Implements a character device with the `open()`, `release()`, `read()`, and `write()` functions.
- b) Uses the `misc_register()` functionality to register itself as a character device.
- c) Provides circular buffer functionality wherein a call to `write()` causes the input bytes to be written to a buffer and upon a `read()` causes the bytes in the buffer to be consumed and returned to the caller. Use a “read” pointer and a “write” pointer to walk around the buffer. A buffer of 400bytes is more than sufficient.
- d) Provides a `sysfs` interface to print the number of bytes in the buffer.

O'Reilly's “Linux Device Drivers, 3rd ed.” is a useful reference for this exercise. It is found online at: <http://lwn.net/Kernel/LDD3/>. There is a copy of a makefile to use for building your driver in LDD3. Using that makefile, you can use a command that has “ARCH=...” and “CROSS_COMPILE=...” to build your module.

For information regarding `misc_register()`, see: <http://www.linuxjournal.com/article/2920>

Once implemented, you should be able to test your driver by doing:

```
# echo foobar > /dev/circchar
# cat /dev/circchar
foobar
```

Tracing with ftrace:

1. Reconfigure the kernel to support ftrace, if it's not already enable:
 - a. The “function” and “function_graph” tracers. Make sure you enable `CONFIG_DYNAMIC_FTRACE`.
 - b. Modules and module unloading. You don't need to enable forced module unloading.
2. Rebuild your kernel, reflash the SD card and reboot. Or use network booting if you want.
3. Start ftrace using the “function” tracer and monitor its output. Careful: once you start ftrace by echoing “1” into `tracing_on`, it'll stay on until you stop it (i.e. echo “0” into same file).

4. Use ftrace to monitor the following:

- Scheduling change events
- CPU frequency scaling events
- Calls to `__kmalloc`
- Calls to the “brk” system call
- Calls to both `__kmalloc` and the “brk” system call
- All events occurring while the “system_server” process is scheduled
- All mcount calls occurring while the “system_server” process is scheduled

5. Modify your driver to add a static tracepoint to monitor each `read()` and `write()` operation to it. Use `trace_printk()` to achieve that, remember that you'll have to use `MODULE_LICENSE("GPL")` in order to have access to this symbol. Rebuild your driver and reload it on the device. Start ftrace tracing and make sure you can see the output in the ftrace output at runtime when you do a “cat” or “echo” as above.

6. Replace your `trace_printk()` statements in your driver with custom-defined `TRACE_EVENT()` events and monitor those events with ftrace.

7. Create a kprobe that catches all calls to the `open()` system call and logs them into ftrace using `trace_printk()`. Build the kprobe as a driver, load it on your device and verify that you can see the output in ftrace's traces.

Device Drivers – Set #2:

Write a device driver that registers an interrupt handler to IRQ 29 – same as the serial driver, it's a “shared” interrupt. Have your interrupt handler print out a message using `trace_printk()` every 10 interrupts it receives. Don't forget to use the `IRQF_SHARED` flag in your call to `request_irq()`, otherwise your request will fail. And don't forget to add a `free_irq()` in your `module_exit`. To generate interrupts, just make sure you causing communication over the serial link. You'll need to return a `IRQ_NONE` from your handler for interrupts to be properly processed.

Build tools and distributions:

1. Download buildroot and extract Buildroot into a “buildroot/” directory in your workspace
2. Configure Buildroot for beaglebone and fix its configuration per the slides
3. Build Buildroot
4. Flash the resulting image to the SD card of the Beaglebone and test it
5. Add buildroot’s uClibc-ng toolchain to your path by modifying your devenv script to add the “buildroot-VERSION/host/bin/” path as the primary path.

App. Development and Debugging:

1. Reconfigure Buildroot to include support for: strace, ltrace and trace-cmd. They are all under the “Target Packages”->debugging submenu.
2. Create a program that opens your circular character driver and writes and reads from it.
3. Use the example Makefile to build your program for the target. Make sure you add the debugging flag “-g”.
4. Load your program onto the target – or use your NFS root.
5. Use the gdb server to remotely step through your program. You will find the “gdbserver” binary in: /usr/bin
6. Use strace on the target to observe the behavior of a few processes. The “strace” binary is in the same location as the “gdbserver”.
7. Use ltrace on the target to do similar tracing.
8. Modify your user-space program to write to the trace_marker file and monitor your user-space process along with the kernel using ftrace.

Device Drivers – Set #3:

Write a device driver and corresponding device tree overlay that cause the driver to be taken into account by the device tree infrastructure.

To start, patch your arch/arm/boot/dts/am335x-bone.dts to add a “dtdemo” entry. Here’s an example from the BeagleBone Black DTS:

```
/ {
model = "TI AM335x BeagleBone Black";
compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";

+     dtdemo {
+         compatible = "ti,dtdemo";
+     };
```

```
};
```

Here's an example overlay:

```
    /dts-v1/;
/plugin/;

/ {
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";

    fragment@0 {
        target-path = "/dtdemo";
        __overlay__ {
            compatible = "ti,dtdemo";
            msg = "Demo module, this is overridden";
        };
    };
};
```

Here's how to build this overlay:

```
$ dtc -I dts -O dtb -o demo.dtbo demo.dts
```

Here's an example uEnv.txt to take the overlay into account:

```
kernel_image=zImage
fdt_image=am335x-boneblack.dtb
fdt_overlay=demo.dtbo
fdtovaddr=0x8800F000

console=ttyO0,115200n8
mmcroot=/dev/mmcblk0p2 ro
mmcrootfstype=ext4 rootwait

load_kernel_image=load mmc ${mmcdev}:${mmcpart} ${loadaddr} ${kernel_image}
load_fdt_image=load mmc ${mmcdev}:${mmcpart} ${fdtaddr} ${fdt_image}
load_fdt_overlay=load mmc ${mmcdev}:${mmcpart} ${fdtovaddr} ${fdt_overlay}; fdt
addr ${fdtaddr}; fdt resize; fdt apply ${fdtovaddr}

mmccargs=setenv bootargs console=${console} root=${mmcroot} rootfstype=${
mmcrootfstype} ${optargs} init=/sbin/init

uenvcmd=run load_kernel_image; run load_fdt_image; run load_fdt_overlay; run
mmccargs; bootz ${loadaddr} - ${fdtaddr}
```