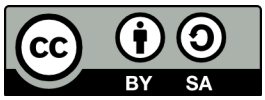
Hands-On Exercises for

Embedded Linux

v. 2022.10

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 2003-2022, Opersys inc.

These exercises created by: Karim Yaghmour

Originals at: www.opersys.com/training/

Delivered and/or customized by:

Software components versions

This page contains information that is required throughout the exercises. Keep this in mind when exercises ask you to get a certain image, apply a patch, configure for a given target, etc.

Hardware:

BeagleBone Black – requires FTDI cable to get serial

Class Drive:

Location to be provided in class

Files:

Prebuilt SD card image:	el-image-bb-191007.img.xz
Workspace:	bbone-black-3.1.0-211108.tar.bz2

Linux Kernel:

Version:	5.15.1
Patch:	NONE
Configuration for target:	multi_v7_defconfig
Kernel image:	arch/arm/boot/zImage
Device tree blob:	arch/arm/boot/dts/am335x-boneblack.dtb

Bootloader:

Package:	U-Boot
Version:	u-boot-2021.10
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

In this section you will learn how to:

- Set up some of the necessary packages for embedded Linux development on Ubuntu
- Configure your workstation to interact with an embedded device using the serial port
- Configure your workstation's network to interact with an embedded device

WARNING: Several exercises in this section and later require you to connect the SD card used to boot the BeagleBone Black to your host in order to program it. On every such occasion, it'll appear as yet another storage device on your host. In all such cases, it's very important to check which device your host's kernel sees this storage device as. You can use the "lsblk" command to help you in this regard. BE. **VERY**. CAREFUL. If you make a mistake, you risk destroying your host's distribution. There is no "undo" if you make such a mistake.

1. Use the following commands to install the following packages (Note that the "\" at the end of some lines is going to inform the shell that you want to continue typing and the shell will create a new line with ">" to allow you to continue typing. Do NOT copy-paste the ">" symbols, they will break the commands.) --- YOU PRESUMABLY ALREADY DID THIS:

```
$ sudo apt update
$ sudo apt install build-essential gawk wget git diffstat unzip \
> texinfo gcc-multilib chrpath socat nfs-kernel-server nfs-common
$ sudo apt install libc6-armel-cross libc6-dev-armel-cross \
> binutils-arm-linux-gnueabi  libncurses5-dev
$ sudo apt 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 over the serial port. 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 and extract it:

```
$ unxz el-image-bb-191007.img.xz
```

4. Use the "dd" command to put it on the SD card and boot with it – here's an example you'll need to adapt to your system (use the "lsblk" command to find out the device to write to)

```
$ sudo dd if=el-image-bb-191007.img of=/dev/sd[DEV] status=progress
```

You need to replace "[DEV]" with your actual storage device.

5. You can use the "sync" command on your host to guarantee that any changes you did previously (or writes) are committed to disk before you proceed – in this case this will hang on the console until the results of the previous "dd" command have been committed to storage:

```
$ sudo sync
```

TIP: We HIGHLY suggest you make it a habit to use "sync" right after any command you use to write directly to an SD card to ensure the contents are committed to the SD card **BEFORE** you remove it from your host. Linux uses filesystem cashing in RAM and when you execute some operation on a storage device, the prompt may return as if the command was completed. Under the hood, though, the kernel may still be writing from its RAM cash to the storage device and if you remove the device before it's done then you have a corrupted storage. By using "sync" you ensure the kernel has completed before you remove the device. This tip will NOT be mentioned again in further exercises, but we'll assume you have taken good note of it and will remember using it every time an operation requires placing the SD card in the host for direct programming. Failing to do this will give you strange results.

6. Connect the programmed SD card to your BeagleBone Black and boot it.

7. Use "picocom" to shell into the BeagleBone over the serial port.

8. Configure your VM to have full control over the USB to Ethernet adapter you will use to connect to the BeagleBone Black. This operation is VM software specific.

9. Connect the Ethernet cable between the BeagleBone Black and the Ethernet adapter on your host.

10. Create a local-LAN configuration for your Ethernet card to allow connection to the target using the Ethernet cable (host: 192.168.203.100; target: 192.168.203.79).

Example Ethernet card configuration:
IPADDR=192.168.203.100
GATEWAY=192.168.203.79
NETWORK=192.168.203.0
NETMASK=255.255.255.0
BROADCAST=192.168.203.255

You may need to fiddle around a bit for the fixed IP configuration to be accepted by the host Ubuntu.

11. Once your Ethernet is connected between the host and the BeagleBone, you should be able to configure your BeagleBone's interface manually – these are example commands to use on your BeagleBone Black:

```
# mount -t proc none /proc
# ifconfig eth0 up 192.168.203.79
```

Now you should be able to ping the Beaglebone from the host and vice versa.

Development Workspace

In this section you will learn how to:

- Create yourself a workspace for building and preparing images for an embedded Linux system.
- Use a script to setup environment variables for facilitating workspace commands
- Use the “fdisk” command to partition disk images
- Observe the kernel’s boot intricacies

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. Delete all partitions from the SD card so that we can manually recreate the content:

```
$ sudo dd if=/dev/zero of=/dev/sd[DEV] bs=512 count=1
```

4. Recreate the SD card partitions required for booting the beaglebone using fdisk and commit the result to disk:

```
$ echo -e "o\nn\np\n1\n\n+64M\na\nt\nc\nn\np\n\n\n\nw\n" | sudo fdisk /dev/sd[DEV]
```

5. Format the SD card vfat partition using “mkfs.vfat”:

```
$ sudo mkfs.vfat /dev/sd[DEV]
```

6. Put the images in your project’s “images/boot-part” directory on your bootable partition:

```
$ cd $PRJROOT
$ cp images/boot-part/* /media/[MYNAME]/[MYDEVICE]/
```

7. Boot with the newly reformatted card. You’ll need to hold the button marked as “S2” on the opposite side of the PCB from where the SD card slot is located to force the MLO/bootloader on the eMMC to load from the SD card. The kernel should panic when it finishes booting because it doesn’t have a root filesystem. That’s expected.

We will see how to force the BeagleBone Black to always boot from the SD card without pressing the S2 button later. It requires a bit of a nasty hack.

Kernel Basics

In this section you will learn how to:

- Work with the kernel's build system
- Configure a kernel for your device
- Build and install all kernel artifacts into your build workspace
- Copy built kernel artifacts to an SD card for booting on your embedded device

NOTE: p. 3 of this document contains the necessary information regarding many of the steps here – such as any necessary patch, which target to configure for, etc.

0. Apply any necessary kernel patch (see p. 3 for the patch)

1. Configure the kernel for our default target (see p. 3 for the target). See slide 137 for an example target-specific configuration command.

2. Start the kernel configuration menu ("make ... menuconfig") and make sure that support for config.gz in /proc is enabled ("Kernel .config support"). You can use the "/" key to search for options. You are looking for the configuration option called "IKCONFIG". Once this is set, exit the configuration menu and accept saving the configuration on exiting.

2. Build the kernel. See slide 140 for an example. The image generated should be listed on p.3 on this exercise document.

3. Install the kernel by copying the build artifacts to the proper location in your workspace. See slide 143 for an example.

4. Build the kernel modules. See slide 144 for an example.

5. Install the kernel modules. Also see slide 144.

6. Build the device trees. See slide 145 for an example.

7. Install the beaglebone device tree. Also see slide 145.

8. Overwrite the default kernel and DTB on the SD card with the one you have just built and boot with them:

1. Reconnect SD card to host
2. Look under the FAT partition and replace the kernel (zImage) and DTB (ends with .dtb) with the ones you just built
3. Unmount the SD card from the host
4. Use "sync" to commit the changes"

5. Insert the SD card into the BBB
6. Boot BBB with newly-modified SD card

9. Check that the kernel booting is your own in its early messages. The kernel version will show up at the top of the kernel boot messages. It should display the version you just built along with your build system ID (username/system) and build time.

Bootloader

In this section you will learn how to:

- Work with U-Boot's build system
- Configure and build U-Boot
- Install U-Boot build artefacts onto the embedded device's SD card
- Work with U-Boot's configuration system
- Configure your host and device for feeding kernels and DTBs at boot time using TFTP

NOTE: p. 3 of this document contains the necessary information regarding many of the steps here – such as any necessary patch, which target to configure for, etc.

0. Apply any necessary U-Boot patch (see p. 3).

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 (replace "/media/karim/..." with the relevant path to your VFAT partition on the SD card):

```
$ cd ${PRJROOT}/images
$ cp MLO /media/karim/….
$ cp u-boot.img /media/karim/….
$ cp uEnv.txt /media/karim/….
```

Don't forget to unmount the SD card and use "sync".

4. Transfer the SD card to your BBB and boot with U-Boot. This is a bit tricky on the BBB as the U-Boot you just copied to your. Here is the full story:

- By default BBB boots from eMMC MLO/U-boot — even if you press the S2 button
- To force BBB to boot from SD MLO/U-boot, we must "destroy" the MLO on the eMMC. See the following for details: https://groups.google.com/g/beagleboard/c/0wtIhOv2Djc
- Boot with either built-in eMMC U-Boot/kernel/filesystem or the ones from your SD card
- Either with the SD Card or the eMMC, destroying the image involves:
  ```
  /# dd if=/dev/zero of=/dev/mmcblk1 count=1 seek=1 bs=128k
  ```
- To restore – not for now, but just to show you what you could do later to undo the previous command:
  - If using SD card, first do this:
    ```
    / # mount -t proc none /proc
    / # mount -t sysfs none /sys
    ```

```
/ # mount -o remount,rw /
/ # mkdir mnt
/ # mount -t ext4 /dev/mmcblk1p1 /mnt
/ # cd /mnt
```

- If using eMMC, first do this:

```
# cd opt/backup/uboot
```

- Actual restore (booting from either SD card or eMMC):

```
# dd if=MLO of=/dev/mmcblk1 count=1 seek=1 bs=128k
# reboot
```

- All this to force BBB to boot from SD card
- Once you have force-removed the default MLO from the eMMC, you should be able to use your custom-built U-Boot to boot directly from SD card.
- Presumably you are now able to see something like this:

```
U-Boot 2021.10 (Nov 07 2021 - 15:41:11 -0500)

CPU  : AM335X-GP rev 2.1
Model: TI AM335x BeagleBone Black
DRAM:  512 MiB
WDT:   Started with servicing (60s timeout)
NAND:  0 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
Loading Environment from FAT... OK
Net:   eth2: ethernet@4a100000, eth3: usb_ether
Hit any key to stop autoboot:  0
471 bytes read in 2 ms (229.5 KiB/s)
...
```

5. New versions of U-Boot do not automatically load the uEnv.txt file from the SD card. uEnv.txt contains the U-Boot environment variables. To fix that, you can load the file manually:

```
=> load mmc 0:1 0x80800000 uEnv.txt
471 bytes read in 2 ms (229.5 KiB/s)
=> env import -t 0x80800000 471
=> run uenvcmd
```

You can also automate this so that it gets loaded automatically upon future reboots (note that there are linewraps in this output and that “0x80800000” and “471” have a space character in between):

```
=> setenv bootcmd load mmc 0:1 0x80800000 uEnv.txt\; env import -t 0x80800000
471 \; run uenvcmd
=> print bootcmd
bootcmd=load mmc 0:1 0x80800000 uEnv.txt; env import -t 0x80800000 471 ; run
uenvcmd
=> saveenv
Saving Environment to FAT... OK
```

6. Try the online help

7. Print U-Boot's environment variables. See slide 165.

8. Configure your host for serving DHCP requests. Look at slides 171-173. In case you're looking for your BBB's mac address:

```
=> printenv ethaddr
ethaddr=64:33:db:42:e6:05
```

Make sure you DHCPD configuration file uses **your** board's Ethernet address, not the default. Same with all other entries in that file. Make sure they are matching your setup.

9. Use "bootp" in U-Boot to get config from DHCP server. Look at slide 175.

10. Configure your host for serving TFTP requests. Look at slide 173 – on your host your TFTP server might be using /srv/tftp instead of the slides' /var/lib/tftpboot.

11. Use TFTP to download image to target. Look at slide 178.

12. Boot with image. Look at slide 179 – use this slightly different bootz command (no need to add "- $ftdadr"):

```
=> bootz $loadaddr
```

In this case, the kernel will look for its DTB right past its own self.

13. Create environment variables in U-Boot's environment to automate the loading of the kernel and the DTB to boot from them using tftp. A few hints:

- You'll need to set a few "setenv"
- Namely, you'll need to set "bootcmd", which is the default "boot" target started by U-Boot at boot time if the countdown is left to go to zero.
- Save it at the end with "saveenv"
- Example combined bootp/bootz:

```
=> setenv bootargs console=ttyO0,115200n8
=> bootp
link up on port 0, speed 100, full duplex
BOOTP broadcast 1
DHCP client bound to address 192.168.203.79 (9 ms)
Using ethernet@4a100000 device
TFTP from server 192.168.203.100; our IP address is 192.168.203.79
Filename 'zImage-DTB-5.3.2-191013.img'.
Load address: 0x82000000
Loading: #################################################  9.1 MiB
         4.3 MiB/s
done
Bytes transferred = 9490080 (90cea0 hex)
=> bootz 82000000
Kernel image @ 0x82000000 [ 0x000000 - 0x8ff200 ]

Starting kernel ...
```

Root filesystem

In this section you will learn how to:

- Create a root filesystem tree for use in an embedded Linux system
- Work with BusyBox's build system
- Configure, compile and install BusyBox
- Install a filesystem onto a storage device for booting an embedded Linux system
- Shell into a fully-booted custom embedded Linux system

1. Create essential rootfs directories. See slide 192.

2. Copy glibc libraries to target's rootfs and strip them. See slide 198. NOTE: contrary to the slides, the libraries are in /usr/arm-linux-gnueabi/lib/, not ${TARGET_PREFIX}/lib. Once you've copied the files, make sure they are set to be executable:

```
$ chmod +x ${PRJROOT}/rootfs/lib/*
```

3. Copy kernel modules to target's rootfs. See slide 203.

4. Make sure you have /dev, /proc and /sys in your rootfs.

5. Get BusyBox 1.34.1 from https://busybox.net/downloads/ and extract under sysapps/.

6. Configure BusyBox using the "make menuconfig" command. See slides 210 – 212.

6. Build and install BusyBox. See slide 212.

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. See slides 223 – 224.

8. Copy your rootfs into the ext4 partition on the device – you'll have to see what directory it's mounted to (it's like not "/home/karim/....").

```
$ cd ${PRJROOT}/rootfs
$ cp -a ${PRJROOT}/rootfs/* /media/karim/…
```

Don't forget to use "sync".

9. Boot the new SD card, you should now have a fully-functional embedded Linux system

Filesystem Types

In this section you will learn how to:

- Create a number of different filesystem image types
- Configure your host and device for using an NFS-mounted root filesystem

0. Install the MTD utilities for your host. See slide 245.

1. Build and install the cramfs utilities. You'll likely need to add the following at the top of "mkcramfs.c" to get the file to build:

```
#include <sys/sysmacros.h>
```

Note that the tool is fairly old and unmaintained. If it ultimately fails, skip to #3 below.

2. Create a CRAMFS image of your target's root filesystem. See slide 261.

3. Install the romfs utilities. See slide 262.

4. Create a ROMFS image of your target's root filesystem. See slide 262.

5. Install the squashfs utilities. See slide 263.

6. Create a squashfs image of your target's root filesystem. See slide 263.

7. Create a UBIFS image of your target's root filesystem. See slide 264.

8. Create a JFFS2 image of your target's root filesystem. See slide 265

9. Create a RAM disk image of your target's root filesystem. See slides 266 – 268. Note that you'll need to use a larger size than the default 8MB spelled out in the slides since that will not be sufficient given the size of the kernel modules.

10. Create an initramfs images of your target's root filesystem. See slide 269.

11. Compare the filesystem image sizes

12. 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. See slides 166, 174 and top of 180. In case of problems with NFS: have a look at the output from "sudo exportfs -v" for NFS entries. If yours isn't there, try "sudo exportfs -ra".

Device Drivers – Set #1

In this section you will learn how to:

- Create basic Linux device drivers
- Build device drivers outside the Linux kernel sources
- Implement a basic character device interface
- Use basic command line tools to interface with a character device driver

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
```

Here’s how to get started in your project – this just creates directories and empty files:

```
$ cd ${PRJROOT}
$ mkdir -p project/circ-char
$ cd project/circ-char
$ touch Makefile circular-char.c
```

Here’s a sample makefile to get started:

```
# If KERNELRELEASE is defined, we've been invoked from the
# kernel build system and can use its language.
ifneq ($(KERNELRELEASE),)
    obj-m := circular-char.o

# Otherwise we were called directly from the command
# line; invoke the kernel build system.
```

```
else
    KERNELDIR ?= $(PRJROOT)/kernel/linux-5.15.1
    PWD := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif

To build - even with empty C file:
make ARCH=arm CROSS_COMPILE=${TARGET}-
```

Tracing with ftrace

In this section you will learn how to:

- Configure your kernel to add tracing support
- Enable ftrace at runtime
- Use ftrace to monitor several types of events
- Add tracing support to a driver
- Use kprobes to add dynamic probes to the kernel

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. You’ll need to mount debugfs to use ftrace:

```
/# mount -t debugfs none /sys/kernel/debug
```

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). See slides 378 – 382

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

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. See slide #384.

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

In this section you will learn how to:

- Add an interrupt handler to a device driver

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

In this section you will learn how to:

- Package an embedded Linux system using Buildroot
- Work with Buildroot's configuration system
- Compile an embedded Linux target system using Buildroot
- Test a Buildroot-generated system on an embedded device

1. Download buildroot and extract Buildroot into a "buildroot/" directory in your workspace. See slide 501. The version we're using is https://buildroot.org/downloads/buildroot-2021.02.7.tar.bz2

2. Configure Buildroot for beagelbone and fix its configuration per the slides. See slides 501 – 502.

3. Build Buildroot. See slide 502.

4. Replace your custom-built rootfs with buildroot's own filesystem (remember NOT to copy the line-wrapped ">" character since it's added by the shell):

```
$ cd $PRJROOT
$ mv rootfs rootfs-orig
$ mkdir rootfs
$ sudo tar -C rootfs \
> -xf buildroot/buildroot-2021.02.7/output/images/rootfs.tar
```

5. Upon rebooting the BBB, the NFS server will serve your BBB the Builroot root filesystem.

6. 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

In this section you will learn how to:

- Configure debugging tools into Buildroot
- Use Unix filesystem I/O to interact with a serial device
- Cross-compile a user-space application for an embedded Linux system
- Use “gdbserver” to step through a user-space program
- Use strace/ltrace to trace through a user-space program
- Integrate user-space tracing events into ftrace

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

In this section you will learn how to:

- Work with device tree overlays

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 overriden";
      };
  };
};
```

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}

mmcargs=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 mmcargs; bootz ${loadaddr} - ${fdtaddr}
```

Tracing with eBPF/BCC

In this section, you'll learn how to get and use the eBPF/BCC tracing functionality.

1. For this section, we'll need a Debian distribution running on the BeagleBone Black:
   - Download the BeagleBone Black Debian image from:

https://debian.beagleboard.org/images/bone-debian-10.3-iot-armhf-2020-04-06-4gb.img.xz
   - Extract the image using "unxz"
   - Copy the image to your target:

```
$ sudo dd if=bone-debian-10.3-iot-armhf-2020-04-06-4gb.img of=/dev/sdb \
> status=progress
3773104640 bytes (3.8 GB, 3.5 GiB) copied, 1708 s, 2.2 MB/s[[1;7A
7372800+0 records in
7372800+0 records out
3774873600 bytes (3.8 GB, 3.5 GiB) copied, 1710.12 s, 2.2 MB/s
```

2. Update and install missing packages for Debian (shell prompt omitted for brevity and ease of copy-pasting):

```
echo "deb-src http://deb.debian.org/debian buster main" > /etc/apt/sources.list.d/buster.list
apt update
apt build-dep bpfcc
apt install clang
```

3. Clone, compile and install the BPF Compiler Collection from source – this will take a while since it's building on the BeagleBone itself:

```
cd
git clone https://github.com/iovisor/bcc.git
mkdir bcc/build; cd bcc/build
cmake ..
make
make install
```

4. The BCC tools are found under /usr/share/bcc/tools/. There are some very insightful utilities available in BCC. We strongly recommend you spend some time exploring them. Here are a handful we recommend:

- execsnoop
- filetop
- opensnoop
- biosnoop (may need some fixing for paths)
- tplist (may need some fixing for paths)
- softirqs

For the following, refer to the BCC reference guide: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md:

5. Create your own BCC command based on bcc-master/examples/hello_world.py that attached to the sched_switch trace event using TRACEPOINT_PROBE instead of the “kprobes__” semantic.

6. Using “syncsnoop.py” as an example, create your own BCC command that snoops on your driver’s read, write and ioctl commands. Try recording and reporting the PID of the caller in addition to a timestamp.

7. Use the previously-coded command to monitor what happens when you “cat” and “echo” to “/dev/circchar”.