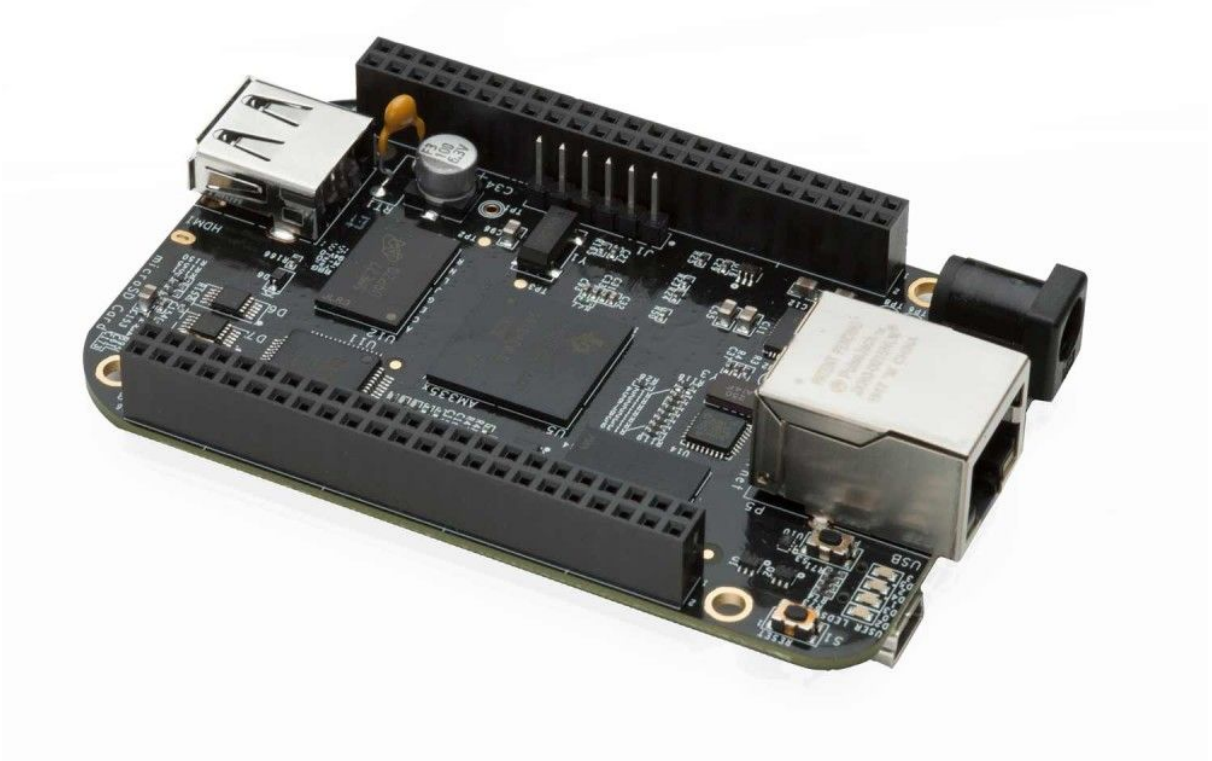
BeagleBone Black: Platform Bring Up with Upstream Components



Sam Protsenko

Rev B, 22 Jan 2019

Table Of Contents

Overview

This document is intended to give the user overall instructions on how to obtain, build and flash upstream software to the BBB board, with detailed explanation of all related features and components. It can be used as a good platform for training C/Embedded/Kernel developers and QA engineers.

We will focus on next components:

1. U-Boot: this is most often used bootloader in embedded systems and it's recommended for BBB. We will use upstream U-Boot, as it supports BBB.
2. Linux Kernel: it's a main part of Linux OS. We will use upstream kernel, as it supports BBB.
3. BusyBox rootfs: it's minimal, so it's the best choice for understanding the crucial OS parts. It can be useful in real work for testing, debugging and isolating the bugs, due to its minimalistic nature. It also often can be found in networking devices (routers, etc), and is used as initramfs in desktop GNU/Linux distributions.

Why use upstream software? Learning about upstream software will give you good understanding of community structure and will prepare you for upstreaming activities in future. It's also always good to know what is the origin of main software components (where are all other git trees are forked from).

Why use BeagleBone Black? BeagleBone Black board is a development board. It's open hardware, so you can find hardware design files (schematic and PCB layout) here:

1. http://beagleboard.org/hardware/design
2. https://github.com/beagleboard/BeagleBone-Black
3. https://www.elinux.org/Beagleboard:BeagleBoneBlack#Hardware_Files

Technical Reference Manual (TRM) and datasheet for SoC (AM335x) are also public and can be found here:

http://www.ti.com/product/AM3358/technicaldocuments

Because it's open hardware board, it's very convenient to use it for training. Even electronics bits could be studied on it. Also, it has good design (as opposed to e.g. Raspberry Pi), which resemble professional embedded devices design. The cost is also quite low for such device's level. This board is very flexible, so we can use it to teach C, Linux, Embedded, Kernel development, testing, etc. The only thing we can't use this board for is Android. For that purpose HiKey960 board is probably the best choice right now.

These instructions were tested on **Ubuntu 18.04 LTS**, but should work on any Debian-based distros. Differences (if any) should be minor, obvious and easy to fix. We recommend to use latest LTS Ubuntu, as it's up-to-date, easy to start and stable OS.

BeagleBone Rev C board is a target board (latest one, up to this point). We recommend you to use Rev C revision.

1 Preparing the Tools

1.1 Initial Ubuntu Setup

We will use `apt` command for handling packages. Other front-ends for APT exist (like `aptitude` or `apt-get`), but `apt` is a recommended command right now. You can read more details here:

https://debian-handbook.info/browse/stable/sect.apt-get.html

Update packages list:

```
$ sudo apt update
```

Please note that you should update packages list each time before installing new packages. No need to do that very often though, one time per day should be fine.

Install tools that will be needed further:

```
$ sudo apt install git vim vim-gtk tree
```

`git` will be used further to obtain Linux kernel sources and other software. `git` is the most popular version control system right now, especially in open-source world. So I'm encouraging you familiarize yourself with it.

We recommend to use `vim` as a standard editor. It's good to know how to use it (at least basics), because `vi` can be found on virtually every Linux distribution, and can be run without GUI (think about working in ssh session with some server). You can use any other editor, of course. `vim` also can be configured as IDE for kernel development.

1.2 ccache

`ccache` tool is used to speed up the compilation time of big C projects. First time you build the project, it's gonna take the same time as without using `ccache`, and all object files will be stored (cached). But when rebuilding it subsequently, object files for unchanged source code files will be taken from previous cache.

Using `ccache` can be helpful, because we may need to rebuild Linux kernel many times as a part of work process. Of course, incremental build (doing "`make`" without "`make distclean`") spare you a lot of time, as well as running make with "`-j4`" flag. But as you will see, complete rebuilding is also needed sometimes (like when using different branch, different configuration, or just when doing clean build).

Install ccache:

```
$ sudo apt install ccache
```

Set size to 5G:

```
$ ccache -M 5G
```

Check ccache stats:

```
$ ccache -s
```

If you want to clear the whole cache, you can use:

```
$ ccache -C
```

By default, ccache files are stored in `~/.ccache`. You can see cache tree (hashed objects) like this (will be populated after your first build using ccache):

```
$ tree ~/.ccache
```

The simplest way for using ccache is to add it to your `CROSS_COMPILE` environment variable:

```
$ export CROSS_COMPILE="ccache your-toolchain-prefix"
```

You can use special ccache environment variables, to specify some particular behavior. For example, using `CCACHE_DIR` variable, you can specify manually some custom path to ccache directory. It can be useful to have the separate ccache dir for some big project, so that you can manipulate that dir apart from the rest of the cache. See “`man ccache`” or details.

1.3 Serial Console

Serial console is basically a remote terminal for your board. Using serial console you can execute shell commands on your board from your PC or laptop, and see its output. To do so, you will need two things:

1. Special cable for serial console, to connect your board to the PC.
    - The board provides serial console connection via UART lines (like Tx/Rx lines):

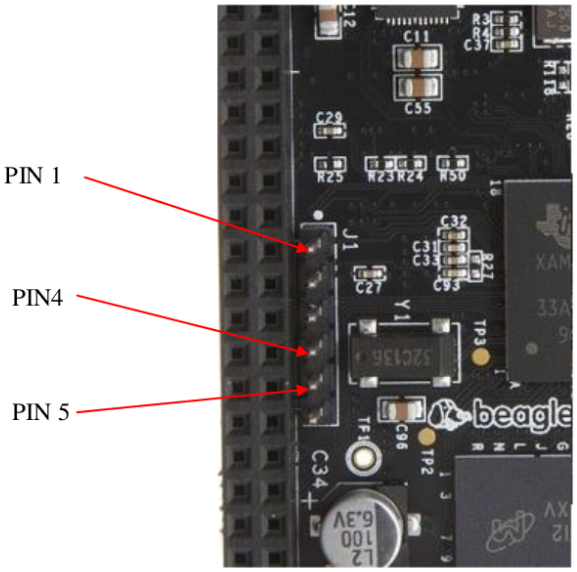


Fig. 1.1 - BBB Serial Port (J1 header)

Please notice that **PIN #1** is a **Ground** (PIN #1 is marked with big white dot on the board). Pin #4 is UART_RX, pin #5 is UART_TX.

- We will use TTL-232R-3V3 cable:

Fig. 1.2 - TTL-232R-3V3 cable

TTL-232R-3V3 is an FTDI-based cable, and it's a recommended choice to use with BeagleBone Black board. There is a cheaper option (PL2303-based cables), but those are not recommended, as some errors were reported for that cable.

Black wire is the ground (black is commonly used color for ground, and red is usually used for power line). Be sure to connect cable with black wire to PIN1.

2. Special software, to be able to manipulate the board shell via the cable.
    - We will use "`minicom`" tool, as it's most commonly used and is a recommended tool by Texas Instruments
    - There are other choices (like `picocom` and `kermit`), but let's stick to `minicom` for now (it has all functions needed for kernel developer)

So first of all let's go ahead and connect our board's serial port to the PC's USB port with our TTL-to-USB cable. Position the cable in such a way that the black wire of your cable is connected to PIN1 of serial port connector on your board:

Fig. 1.3 - TTL-to-USB cable connection

Now that cable is connected to the board and to the PC, you should see that the `/dev/ttyUSB0` file appeared (you can also check `dmesg` and `lsusb` output to be sure it's an FTDI cable, and not some other device):

```
$ ls -l /dev/ttyUSB*
```

You should see that `/dev/ttyUSB0` file is in `dialout` group. Let's add current user to the `dialout` group, so that we can use serial console without root privileges:

```
$ sudo usermod -a -G dialout $USER
```

Logout/login in your OS (Ubuntu).

Check if your user is in `dialout` group now:

```
$ id | grep dialout
```

Install `minicom` tool:

```
$ sudo apt install minicom
```

Now let's configure `minicom`:

```
$ sudo minicom -s
```

Select "Serial port setup" menu item and choose next settings:

- Serial Device: /dev/ttyUSB0
- Bps/Par/Bits: 115200 8N1
- Hardware Flow Control: **No**

Then select "Save setup as dfl" and "Exit from Minicom".

Launch `minicom` (from the regular user, not from root):

```
$ minicom
```

Or if you want color enabled:

```
$ minicom -c on
```

Notice that if you use transparent background for your terminal, it won't work correctly when minicom is running in color mode, due to some ncurses related issue.

In order to exit from minicom, press "Ctrl-A, Q". For minicom menu pop-up, press "Ctrl-A, Z". There are a lot of useful features in minicom menu, like:

- capturing the minicom output to file ("L"); useful for collecting the log files
- sending the break ("F"); it's needed for issuing Magic SysRq keys

Get yourself acquainted with those options, as you'll need it in further work.

1.4 Udev Rules for OTG USB

OTG USB connection on BBB can be used for two main purposes:

1. Flashing new software (images) to eMMC (e.g. using `dfu-util` or `fastboot` tool)
2. Using the board as a USB gadget (can be configured via ConfigFS in Linux). Some examples of useful USB gadgets are:
    - Mass Storage Gadget (your BBB board will act like USB flash drive)
    - rndis (you will have Ethernet network connection, but via USB cable)
    - ADB (Android protocol used for many things like copying files, shell, etc)
    - Actually, fastboot and DFU modes (see above) are also implemented via USB gadget

First of all, we will need to connect the board to PC using mini-USB to USB cable. Please connect the board to PC as shown on Figure 1.4.

Fig 1.4 - OTG USB cable connection

In order to be able to use USB OTG connection without root privileges, we need to specify udev rules accordingly. We know that BeagleBone Black (when in USB gadget mode) will appear with vendor ID = 0451 (Texas Instruments). Let's specify that OTG USB device file should have `plugdev` group identifier.

Create or edit `/etc/udev/rules.d/51-android.rules` file:

```
$ sudo vim /etc/udev/rules.d/51-android.rules
```

and add next text to it:

```
# Texas Instruments
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", MODE="0664", GROUP="plugdev"
```

Reload udev rules:

```
$ sudo udevadm control --reload
$ sudo udevadm trigger
```

Once udev rules are reloaded, corresponding device file in `/dev/bus/usb/` will be in plugdev group (which your user also belongs to). So now you should be able to read/write to OTG USB file from regular user. That means you can run flashing tools (like `dfu-util` or `fastboot`) from regular user, without root privileges.

1.5 Toolchains

Toolchain is a bunch of tools for cross-compiling programs for the architecture different than host architecture. For example, we will use toolchains to build programs for ARM device (*target*, i.e. your BeagleBone Black board) from x86_64 PC (*host*, i.e. your laptop/PC). Toolchain usually consists of next main components:

- C compiler, linker and related tools (gcc, ld, as, cpp)
- libc (and other related libraries)
- debugger (GDB)
- tools for working with cross-compiled binaries (readelf, objdump, etc.)

There are two kinds of toolchains:

1. Baremetal (EABI) toolchain: for building programs being run without OS
2. Linux toolchain: for building user-space programs being run under Linux OS

Toolchain executables usually have prefix, like “arm-eabi-” or “arm-linux-gnueabihf-”. So, instead of regular “gcc” program you will see something like “arm-eabi-gcc”, inside of toolchain’s `bin/` directory. The prefix itself contains so called *tuple*:

- the full form is: arch - vendor - OS - libc/ABI
- the shortened form: arch - OS - libc/ABI
- even more shortened form: arch - libc/ABI

where:

- **arch** is your target architecture
- **vendor** is the name of company which developed this toolchain
- **OS** is the name of OS where your cross-compiled program is supposed to run
- **libc/ABI** indicates which C library and ABI is provided by this toolchain.

To configure your shell environment for using the chosen toolchain, you would usually have to add toolchain’s `bin/` dir to your `$PATH` environment variable. Also, usually you will need to provide `$CROSS_COMPILE` variable with your toolchain’s prefix string (it will be further used in Makefile of compiled project to run correct tool, like `${CROSS_COMPILE}gcc`).

1.5.1 Baremetal toolchain

Baremetal toolchain is used to build “bare” programs, which are being run without Linux environment (no system calls, no libc). Most often it is used for building next software: firmwares, bootloaders, Linux kernel.

1. Download baremetal toolchain from here:

   https://releases.linaro.org/components/toolchain/binaries/7.3-2018.05/arm-eabi/

   This file should be downloaded:

   gcc-linaro-7.3.1-2018.05-x86_64_arm-eabi.tar.xz

   You can use `wget` command to download it, e.g. into `~/Downloads`. Or just use your web-browser.

2. Extract toolchain to /opt:

```
$ sudo tar xJvf gcc-linaro-7.3.1-2018.05-x86_64_arm-eabi.tar.xz -C /opt/
```

1.5.2 Linux toolchain

Linux toolchain is used to build user-space programs, which rely on Linux environment. It contains full version of GNU libc (glibc), which relies on syscall kernel interface. We will use this toolchain to build our rootfs (BusyBox) and user-space tools.

1. Download Linux toolchain (for building user-space programs):

   https://releases.linaro.org/components/toolchain/binaries/7.3-2018.05/arm-linux-gnueabihf/

   This file should be downloaded:

   gcc-linaro-7.3.1-2018.05-x86_64_arm-linux-gnueabihf.tar.xz

   You can use `wget` command to download it, e.g. into `~/Downloads`. Or just use your web-browser.

2. Extract toolchain to /opt:

```
$ sudo tar xJvf gcc-linaro-7.3.1-2018.05-x86_64_arm-linux-gnueabihf.tar.xz \
  -C /opt/
```

2 Obtaining and Building the Software

Building the software is a relatively easy task. Usually required steps are next:

1. Obtain the software using git (or download the tarball)
2. Checkout to desired branch or tag (we will use stable branches and release tags, so that the software is guaranteed to work)
3. Read `README` and `INSTALL` files for instructions (what the dependencies are, how to configure and how to build the software)
4. Install all the build dependencies (if any)
5. Configure shell environment for cross-compiling with your toolchain
6. Configure the software for build with options you desire
7. Build the software
8. Install the built software to chosen location (if needed)
9. Flash the built software to the device

Install host build dependencies needed for all projects we're gonna build:

```
$ sudo apt install make libncurses5-dev libssl-dev bc bison flex
```

Let's download all sources in `~/repos` dir:

```
$ mkdir ~/repos
```

If you'd like to use some different directory for sources, please replace `~/repos` in all further instructions with your path.

2.1 U-Boot

U-Boot is a bootloader, it is commonly used on embedded boards and Android devices.

Obtain U-Boot source code:

```
$ cd ~/repos
$ git clone git://git.denx.de/u-boot.git
$ cd u-boot
```

There are basically no stable branches in U-Boot, so we'll use the last release tag. Let's find out the latest release tags (but not release candidate tags):

```
$ git tag | grep -v rc | tail -5
```

Checkout to the latest release tag:

```
$ git checkout v2019.01
```

Also, please revert next commit, to fix a regression it introduces:

```
$ git revert --no-edit d30ba2315ae3
```

Configure toolchain environment in shell:

```
$ export PATH=/opt/gcc-linaro-7.3.1-2018.05-x86_64_arm-eabi/bin:$PATH
$ export CROSS_COMPILE='ccache arm-eabi-'
$ export ARCH=arm
```

Build U-Boot:

```
$ make am335x_boneblack_defconfig
$ make -j4
```

Output files:

- `MLO` (first-stage bootloader)
- `u-boot.img` (second-stage bootloader)

2.2 Kernel

Linux kernel is a main part of Linux OS and Android OS. It's universally used on Embedded systems.

You can find the link for git-cloning on Linux kernel site here:

https://www.kernel.org/

Obtain Linux kernel source code from repository with stable branches:

```
$ cd ~/repos
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
$ cd linux-stable
```

Checkout to the last **longterm** stable branch (can be found from "Releases" tab on kernel.org):

```
$ git checkout linux-4.19.y
```

Configure toolchain environment in shell:

```
$ export PATH=/opt/gcc-linaro-7.3.1-2018.05-x86_64_arm-eabi/bin:$PATH
$ export CROSS_COMPILE='ccache arm-eabi-'
$ export ARCH=arm
```

We will use `multi_v7_defconfig` as a base, which is a common config file for all ARMv7 boards. But we still need to enable some kernel options on top of it, for BBB. For that, let's create config fragment:

```
$ mkdir fragments
$ vim fragments/bbb.cfg
```

Then add next content and save the file:

```
# Use multi_v7_defconfig as a base for merge_config.sh
# --- USB ---
# Enable USB on BBB (AM335x)
CONFIG_USB_ANNOUNCE_NEW_DEVICES=y
CONFIG_USB_EHCI_ROOT_HUB_TT=y
CONFIG_AM335X_PHY_USB=y
CONFIG_USB_MUSB_TUSB6010=y
CONFIG_USB_MUSB_OMAP2PLUS=y
CONFIG_USB_MUSB_HDRC=y
CONFIG_USB_MUSB_DSPS=y
CONFIG_USB_MUSB_AM35X=y
CONFIG_USB_CONFIGFS=y
CONFIG_NOP_USB_XCEIV=y
# For USB keyboard and mouse
CONFIG_USB_HID=y
CONFIG_USB_HIDDEV=y
# For PL2303, FTDI, etc
CONFIG_USB_SERIAL=y
CONFIG_USB_SERIAL_PL2303=y
CONFIG_USB_SERIAL_GENERIC=y
CONFIG_USB_SERIAL_SIMPLE=y
CONFIG_USB_SERIAL_FTDI_SIO=y
# For USB mass storage devices (like flash USB stick)
CONFIG_USB_ULPI=y
CONFIG_USB_ULPI_BUS=y
# --- Networking ---
CONFIG_BRIDGE=y
# --- Device Tree Overlays (.dtbo support) ---
CONFIG_OF_OVERLAY=y
```

Generate `.config` file from `multi_v7_defconfig` + our fragment:

```
$ ./scripts/kconfig/merge_config.sh \
  arch/arm/configs/multi_v7_defconfig fragments/bbb.cfg
```

Build kernel image, device tree blob and all kernel modules:

```
$ make -j4 zImage modules am335x-boneblack.dtb
```

Output files:

- `arch/arm/boot/zImage`
- `arch/arm/boot/dts/am335x-boneblack.dtb`

2.3 BusyBox

2.3.1 Obtain and build

Let's build minimal Busybox rootfs.

Obtain sources:

```
$ cd ~/repos
$ git clone git://git.busybox.net/busybox
$ cd busybox
```

Checkout to latest stable branch:

```
$ git branch -a | grep stable | sort -V | tail -1
$ git checkout 1_29_stable
```

Setup build environment (use Linux toolchain):

```
$ export ARCH=arm
$ export PATH=/opt/gcc-linaro-7.3.1-2018.05-x86_64_arm-linux-gnueabihf/bin:$PATH
$ export CROSS_COMPILE="ccache arm-linux-gnueabihf-"
```

Configure BusyBox with all features (largest general-purpose configuration):

```
$ make defconfig
```

We will build BusyBox dynamically (`busybox` binary will be dynamically linked against libc). So there is no need to modify the configuration at this point. Although it may be useful to have BusyBox as one independent binary (statically linked), it may not work correctly with resolving network host names (libnss requires the binary to be dynamically linked against libc to work correctly).

Build BusyBox:

```
$ make -j4
```

Install to `_install/` dir:

```
$ make install
```

Create directories needed to populate rootfs with init files and missing nodes:

```
$ mkdir -p _install/{boot,dev,etc\/init.d,lib,proc,root,sys\/kernel\/debug,tmp}
```

2.3.2 Make init work

Create init script (`_install/etc/init.d/rcS`):

```
#!/bin/sh

mount -t sysfs none /sys
mount -t proc none /proc
mount -t debugfs none /sys/kernel/debug

echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

See `docs/mdev.txt` busybox documentation for details. We are installing mdev as a hotplug helper, so that the kernel is able to notify user-space (via that helper) about hotplug events.

Make it executable:

```
$ chmod +x _install/etc/init.d/rcS
```

Make a link to init system in root dir (so that kernel can run it):

```
$ ln -s bin/busybox _install/init
```

2.3.3 Populate /boot

Populate `/boot` rootfs directory with kernel files:

```
$ cd _install/boot
$ cp ~/repos/linux-stable/arch/arm/boot/zImage .
$ cp ~/repos/linux-stable/arch/arm/boot/dts/am335x-boneblack.dtb .
$ cp ~/repos/linux-stable/System.map .
$ cp ~/repos/linux-stable/.config ./config
$ cd -
```

`System.map` and `config` files are not necessary, but having them in rootfs may be helpful for development and debugging reasons. Also, doing "`make install`" and similar targets in kernel creates the similar structure.

2.3.4 Populate /lib

For this part, your shell environment should be configured for Linux toolchain.

Copy kernel modules to `lib/modules/$(uname -r)/` in our rootfs:

```
$ cd ~/repos/linux-stable
$ export INSTALL_MOD_PATH=~/repos/busybox/_install
$ export ARCH=arm
$ make modules_install
$ cd -
```

As BusyBox was dynamically linked, we need to copy system libraries from toolchain to `lib/` directory. The `busybox` binary depends on `libc.so`, `libm.so` and `ld-linux.so` (which is actually a dependency of `libc.so`). You can check it using this command:

```
$ ${CROSS_COMPILE}readelf -d _install/bin/busybox | grep NEEDED
```

But `busybox` also implicitly depends on `libnss.so` (by doing `dlopen()` call). So let's just install all toolchain libraries to met all dependencies:

```
$ cd _install/lib
$ libc_dir=$(${CROSS_COMPILE}gcc -print-sysroot)/lib
$ cp -a $libc_dir/*.so* .
$ cd -
```

2.3.5 Populate /etc

Prepare `mdev` configuration (support for module loading on hotplug):

```
$ echo '$MODALIAS=.* root:root 660 @modprobe "$MODALIAS"' > \
  _install/etc/mdev.conf
```

On sophisticated systems like Debian, hotplug events (and much more) are handled by device manager called `udev`. On BusyBox we have more simplified mdev tool for this. For details, please read udev article on Wikipedia, and `docs/mdev.txt` in BusyBox source code directory. If you are curious about hotplug uevents, please refer to Appendix A.

Create files needed for `/etc/mdev.conf` to work correctly:

```
$ echo 'root:x:0:' > _install/etc/group
$ echo 'root:x:0:0:root:/root:/bin/sh' > _install/etc/passwd
$ echo 'root::10933:0:99999:7:::' > _install/etc/shadow
```

Add `resolv.conf` file, which will be used to resolve network host names into IP addresses (e.g. when using tools like `ping`, `nslookup`, etc.):

```
$ echo "nameserver 8.8.8.8" > _install/etc/resolv.conf
```

Now the minimal `/etc` configuration is complete. If you want to learn about more comprehensive configuration files, look at next files:

- in BusyBox project: `examples/mdev_fat.conf`
- in BuildRoot project:
    - `package/busybox/mdev.conf`
    - `system/skeleton/etc/{group,passwd,shadow}`

Of course, for real-life tasks, aside from education/development, it's probably better to go with some comprehensive and ready-to-use rootfs, like Debian. In that case all those files (and much more) will be prepared there for you. But for educational purposes we want to use absolute minimum, to see what is essential, and not to be distracted by something fancy.

3 Flashing and Booting

3.1 QEMU Boot

We can test our kernel and rootfs without actual BeagleBone Black board, using QEMU. QEMU is a software emulator for various machines, like ARM boards. In other words, it's a virtual machine.

Install QEMU:

```
$ sudo apt install qemu-system-arm
```

Create CPIO archive of your rootfs and compress it using GZip:

```
$ cd ~/repos/busybox
$ cd _install
$ find . | cpio -o -H newc | gzip > ../rootfs.cpio.gz
$ cd ..
```

Our kernel image will work fine, basically because of next:

- we built our kernel with `multi_v7_defconfig` configuration, which allows the kernel to be run on any ARMv7 machines
- all device-specific information is supplied to kernel via device tree file (`am335x-boneblack.dtb` in our case); so kernel image itself (`zImage`) only contains the code common to all ARMv7 platforms

Before we run QEMU boot, there is some useful info about it:

- we'll be using "`virt`" machine, so that we don't need to provide any `dtb` files to QEMU
- no bootloader needed for QEMU boot, so we won't use U-Boot images just yet
- rootfs will be mounted as a RAM disk, so any changes to it during QEMU boot won't be reflected in `rootfs.cpio.gz`. It means that on next run you'll see all files "from scratch", doesn't matter if you create/delete/modify any files.

Run the kernel and rootfs in QEMU.

```
$ qemu-system-arm -kernel _install/boot/zImage -initrd rootfs.cpio.gz \
  -machine virt -nographic -m 512 \
  --append "root=/dev/ram0 rw console=ttyAMA0,115200 mem=512M"
```

You will see the kernel log from running machine. Kernel will unpack rootfs, then run /init process.

Once boot process of virtual machine is over, press Enter to see the command prompt. Run next commands, to be sure that's what you built, and to explore the system a bit:

```
/ # uname -a
/ # ls -l
/ # dmesg | grep init
/ # busybox --help | head -15
/ # poweroff
```

If QEMU is hanging after "poweroff", press "Ctrl-A, X" to exit. Or alternatively press "Ctrl-A, C" to get into QEMU monitor, and then run "q" command.

Another nice feature QEMU possesses is an ability to do some low-level debugging without actual JTAG device. It's quite easy to find out such instructions on Internet, using keywords like "qemu arm kernel kgdb".

3.2 SD Card Boot

Although eMMC is much faster and more convenient than SD card, booting from SD card can be still very useful in some cases. For example:

- To unbrick the device (when U-Boot on eMMC was incorrectly flashed and became not functional)
- When U-Boot is missing on eMMC or too old and can't be reflashed with conventional methods (e.g. if it's new device with factory-flashed bootloader)
- You don't want to mess up the eMMC setup
- You need to test or develop something related to SD cards

In those cases you'll need to format and prepare your SD card properly, flashing it with bootloader and rootfs files.

3.2.1 Formatting SD card

Install `sfdisk` tool (we will use it to partition the SD card):

```
$ sudo apt install util-linux dosfstools e2fsprogs
```

First, let's figure out which device file represents your SD card. Perform this command before and after inserting your SD card to your laptop (via adapter or using card reader):

```
$ sudo fdisk -l
```

Also, check the kernel log output after inserting your SD card:

```
$ sudo dmesg
```

Let's say you figured that your SD card device file is `/dev/mmcblk0`. We need to remove old partitions and create two new partitions using next scheme:

1. **Start offset** = 2048 sectors (1 MiB), **size** = 100 MiB, **type** = 0x0c (W95 FAT32), **bootable** = yes
2. **Start offset** = right after 1st partition, **size** = till the end of SD card, **type** = 0x83 (Linux), **bootable** = no

1 MiB offset in the first partition is needed to store MBR (first sector, which is 512 bytes), and due to partition alignment it should be 1 MiB. FAT type is required by ROM code (see AM335x TRM for details).

Insert SD card in your host (laptop/PC). Unmount partitions, if they were mounted automatically:

```
$ sudo umount /dev/mmcblk0p1
$ sudo umount /dev/mmcblk0p2
```

Erase old partition table:

```
$ sudo dd if=/dev/zero of=/dev/mmcblk0 bs=1M count=1
```

Create new partition table:

```
$ sudo sfdisk /dev/mmcblk0 << EOF
2048,100M,0x0c,*
,,L,-
EOF
```

Format both partitions to desired file system types:

```
$ sudo mkfs.vfat -F 32 -n "boot" /dev/mmcblk0p1
$ sudo mkfs.ext4 -F -L "rootfs" /dev/mmcblk0p2
```

Of course, `/dev/mmcblk0` should be replaced with correct SD card device file in all commands above.

Check that you have correct partitions layout on your SD card:

```
$ sudo fdisk -l
```

You should see something like this:

```
Device          Boot  Start       End  Size  Id  Type
/dev/mmcblk0p1  *      2048    206847  100M   c  W95 FAT32 (LBA)
/dev/mmcblk0p2       206848  31422463  14.9G 83  Linux
```

First partition will contain bootloader files. Second partition will contain rootfs files.

Now let's prepare and copy all needed files to SD card.

3.2.2 Create bootable SD card with BusyBox rootfs

Insert SD card in your host (laptop/PC). Unmount partitions, if they were mounted automatically:

```
$ sudo umount /dev/mmcblk0p1
$ sudo umount /dev/mmcblk0p2
```

Mount partitions:

```
$ sudo mkdir /mnt/{boot,rootfs}
$ sudo mount /dev/mmcblk0p1 /mnt/boot
$ sudo mount /dev/mmcblk0p2 /mnt/rootfs
```

Copy rootfs files to SD card ("rootfs" partition):

```
$ cd ~/repos/busybox
$ cd _install
$ sudo cp -R . /mnt/rootfs
```

Copy U-Boot files to SD card ("boot" partition):

```
$ cd ~/repos/u-boot
$ sudo cp MLO u-boot.img /mnt/boot
```

Unmount SD card:

```
$ sudo umount /mnt/boot
$ sudo umount /mnt/rootfs
```

3.2.3 Booting from SD card

Use next procedure to boot Linux from SD card:

- Unplug power cable and mini-USB cable from board (as mini-USB cable also provides power to board)
- Press and hold `USER/BOOT` button
- Plug power cable and mini-USB cable back to board
- Release `USER/BOOT` button

Note: just pressing `RESET` button (while holding `USER/BOOT` button) won't work, so don't rely on it, and reconnect the power (or USB cable) instead, as described above.

BusyBox will be loaded from SD card.

Let’s get into U-Boot shell now. If you are in BusyBox shell, just issue “reboot” command. Or just repeat SD boot procedure above. Please press SPACE when you see the message like this (when board just started to boot):

```
Press SPACE to abort autoboot in 2 seconds
```

Once you pressed SPACE, you will fall out into U-Boot shell, you’ll see U-Boot prompt:

```
=>
```

Run next commands to save default U-Boot environment to SD card:

```
=> env default -f -a
=> env save
```

Now let’s reboot to continue normal execution, so that kernel and BusyBox rootfs will be booted from SD card:

```
=> reset
```

3.3 eMMC Boot

3.3.1 Prepare ext4 rootfs image (for flashing using DFU)

In order to flash rootfs to eMMC via DFU we need to have ext4 image of our rootfs. Let’s create it from built BusyBox (from `_install/` directory):

```
$ cd ~/repos/busybox
```

Install `make_ext4fs` tool:

```
$ sudo apt update
$ sudo apt install android-tools-fsutils
```

Check `_install/` directory size:

```
$ du -h -d 0 _install/
```

In our case it’s 89 MiB. Let’s stick to 144 MiB image size (150,000,000 bytes), because we need some extra size for ext4 FS info and some free space where we can work.

Create ext4 image from `_install/` directory (BusyBox rootfs):

```
$ make_ext4fs -L rootfs -l 150000000 rootfs.ext4 _install
```

The name of created image file is `rootfs.ext4`.

3.3.2 Prepare sparse rootfs image (for flashing using fastboot)

For fastboot flashing we are going to use Android sparse image format. It was introduced to improve USB transfer speed by avoiding the transmission of zeroed areas of the image. See next link for details:

http://www.2net.co.uk/tutorial/android-sparse-image-format

We will use `rootfs.ext4` image (that we created in previous step) to create sparse image file. Some tools are needed to do so, let's install them first:

```
$ sudo apt install android-tools-fsutils
```

Now let's create sparse image from our ext4 image:

```
$ ext2simg rootfs.ext4 rootfs.img
```

Note that `ext2simg` tool is intended for converting ext4 images to sparse format. If you want to convert something other than ext4 to sparse, please use `img2simg` tool.

3.3.3 Flashing eMMC via USB using DFU

Install `dfu-util` tool:

```
$ sudo apt install dfu-util
```

Let's boot from SD card first. Get into U-Boot shell and execute all commands below.

Enter U-Boot shell and execute next commands to partition the eMMC (partition table is in `$partitions` U-Boot environment variable):

```
=> env default -f -a
=> env save
=> gpt write mmc 1 $partitions
```

Check partition list with next command:

```
=> part list mmc 1
```

You should see something like this:

```
Part   Start LBA    End LBA      Name

  1    0x00000300   0x000010ff   "bootloader"
  2    0x00001500   0x00393fde   "rootfs"
```

Check partitions start/end addresses (they are in sectors, in hex notation; 1 sector = 512 bytes).

Set and check DFU info for eMMC:

```
=> setenv dfu_alt_info $dfu_alt_info_emmc
=> env save
=> dfu 0 mmc 1 list
```

You should see next DFU partitions:

```
DFU alt settings list:
dev: eMMC alt: 0 name: rawemmc layout: RAW_ADDR
dev: eMMC alt: 1 name: boot layout: RAW_ADDR
dev: eMMC alt: 2 name: rootfs layout: RAW_ADDR
dev: eMMC alt: 3 name: MLO layout: FAT
dev: eMMC alt: 4 name: MLO.raw layout: RAW_ADDR
dev: eMMC alt: 5 name: u-boot.img.raw layout: RAW_ADDR
dev: eMMC alt: 6 name: u-env.raw layout: RAW_ADDR
dev: eMMC alt: 7 name: spl-os-args.raw layout: RAW_ADDR
dev: eMMC alt: 8 name: spl-os-image.raw layout: RAW_ADDR
dev: eMMC alt: 9 name: spl-os-args layout: FAT
dev: eMMC alt: 10 name: spl-os-image layout: FAT
dev: eMMC alt: 11 name: u-boot.img layout: FAT
dev: eMMC alt: 12 name: uEnv.txt layout: FAT
```

Enter DFU mode so that you can flash images to eMMC partitions over OTG USB cable (0 stands for USB controller number, 1 stands for MMC controller number):

```
=> dfu 0 mmc 1
```

After this command input will hang, and can be only interrupted with Ctrl+C. Also, if some warning message appears, please ignore it, it’s probably harmless.

On host machine: check that DFU device appeared on USB OTG:

```
$ sudo dmesg
```

You should see that OTG USB device (“USB download gadget”) was enumerated:

```
usb 3-2: new high-speed USB device number 8 using xhci_hcd
usb 3-2: New USB device found, idVendor=0451, idProduct=d022
usb 3-2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 3-2: Product: USB download gadget
usb 3-2: Manufacturer: Texas Instruments
```

Flash U-Boot images to eMMC via DFU:

```
$ cd ~/repos/u-boot
$ dfu-util -D MLO -a MLO.raw
$ dfu-util -D u-boot.img -a u-boot.img.raw
```

Flash rootfs image to eMMC via DFU:

```
$ cd ~/repos/busybox
$ dfu-util -D rootfs.ext4 -a rootfs
```

It’ll take a while, as DFU works only via USB EndPoint 0, which has some limitations, so USB transmission speed is rather low, when using DFU. Two possible ways to fix it are:

1. Use fastboot instead (see next section); it uses different EndPoint, so speed is better
2. Use DFU over TFTP (we don’t show it here)

Now that your eMMC is flashed with U-Boot and rootfs, please do next:

1. Power off the board
2. Remove SD card
3. Power on the board; the boot will start from eMMC
4. Press SPACE to get into U-Boot shell; it’s U-Boot booted from eMMC
5. **Repeat all U-Boot commands from this section**

This way you’ll have prepared U-Boot environment on eMMC for further development. Now that your U-Boot environment in eMMC is configured, you can put aside the SD card. You won’t need it anymore, except if you “brick” U-Boot on eMMC (then you can use SD card to restore it). But for regular development, booting from SD card is inconvenient (needs constant removing/insertion when you want to update bootloader, kernel or rootfs on it).

3.3.4 Flashing eMMC via USB using fastboot

DFU is a standard and recommended way for flashing images to eMMC. But it has one shortcoming: the transmission speed via DFU is rather small (because it’s bound to use EP0). So it’s not the best choice for flashing huge images, like rootfs. Better choice would be to use `fastboot` tool (it uses EP1 for the transfer, so it’s faster than DFU). Fastboot is an Android protocol for flashing images, but we can use it nevertheless for regular Linux images, as it’s supported in U-Boot.

Install `fastboot` tool:

```
$ sudo apt install fastboot
```

Enter fastboot mode in U-Boot shell (on USB controller #0, which is OTG USB):

```
=> fastboot 0
```

Check that BBB is visible from fastboot tool:

```
$ fastboot devices
```

If it's not there, you can also check `dmesg` output. The board in fastboot mode is configured as download USB gadget, so it should appear in `dmesg` output.

Flash rootfs image (in Android sparse image format) to `rootfs` eMMC partition:

```
$ fastboot flash rootfs rootfs.img
```

You can find other useful features in fastboot, like formatting the eMMC (`fastboot oem format`), rebooting the board (`fastboot reboot`), obtaining fastboot variables (`fastboot getvar`), etc. Please check the output of "`fastboot --help`" command on your host machine to see more details.

3.3.5 Booting from eMMC

Just power on the board (without SD card inserted) and wait for it to boot. Or (if you are in U-Boot shell), next command can be used to boot Linux from eMMC:

```
=> run bootcmd
```

3.4 TFTP Boot

TFTP boot is a case of booting the software obtained from network. User is requesting the software (kernel, dtb and rootfs) from host machine acting as TFTP server. The software is copied via Ethernet from host to device's RAM and then user boots it from there. Of course, device should be connected to host machine (or to the local network) using patch-cord. Rootfs is used as a RAM disk in this boot scheme (rather than mounting it from eMMC), so any changes to it will be lost on sequential boot. Taking all of that into the account, we can see pros/cons of such boot method, and possible use-cases when it can be convenient.

Pros:

- no tampering with eMMC (so we can keep our regular system on eMMC and do development/debugging on some other rootfs)
- fast speed (100 MBit/sec)

Cons:

- changes to RAM disk won't be reflected neither on used rootfs image, nor on eMMC rootfs partition (be sure it's what you want)
- some preceding host configuration is needed
- you need to do images download via TFTP for each new boot

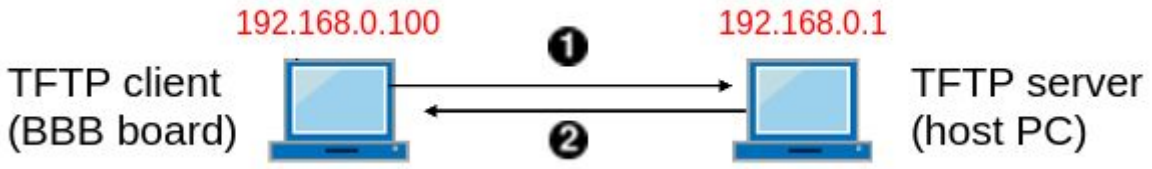


Fig 3.1 - TFTP boot with static IP

We will use static configuration of ethernet on BBB, because this way we don't need to bring up DHCP server, or use some external one.

From Fig 3.1 one can see how TFTP boot works:

1. Board requests (from U-Boot) images from server (zImage, dtb, ramdisk)
2. Server sends requested images to boards (via Ethernet)
3. Board obtains images in RAM
4. Board starts the kernel from RAM, mounting rootfs as RAM disk

3.4.1 Prepare ramdisk image

In order to run Linux using TFTP network boot, we'll need ramdisk in so called "legacy U-Boot image" format. In that case, you can even remove `/boot` directory in your rootfs, as zImage and dtb files will be obtained over Ethernet (discussed later).

Install the package with `mkimage` tool that will be used in this section:

```
$ sudo apt install u-boot-tools
```

Create CPIO archive of your rootfs and compress it using GZip:

```
$ cd ~/repos/busybox
$ cd _install
$ find . | cpio -o -H newc | gzip > ../rootfs.cpio.gz
$ cd ..
```

Now create legacy U-Boot image of your ramdisk:

```
$ mkimage                \
       -A arm            \
       -T ramdisk        \
       -C gzip           \
       -O linux          \
       -n "Ramdisk Image"  \
       -d rootfs.cpio.gz   \
       uRamdisk
```

Resulting file is `uRamdisk`. It will be transferred through the network very fast, because it's basically gzipped (compressed) rootfs, and therefore its size is quite small.

3.4.2 Host configuration

3.4.2.1 TFTP daemon configuration

There are 2 approaches on how to run tftpd daemon:

1. Make tftpd always running (by specifying `RUN_DAEMON="yes"` in tftpd-hpa config file)
2. Let super-server handle tftpd starting

We will use second option. "xinetd" is a super-server daemon. It listens for incoming requests over a network and launches the appropriate service for that request. In our case it will spawn tftpd service when it catches TFTP request over the network.

Install TFTP server and the super-server:

```
$ sudo apt install tftpd-hpa xinetd
```

Make sure that `/etc/default/tftpd-hpa` file (TFTP server config) looks like this:

```
TFTP_USERNAME="tftp"
TFTP_DIRECTORY="/srv/tftp"
TFTP_ADDRESS="0.0.0.0:69"
TFTP_OPTIONS="--secure"
```

Create or edit `/etc/xinetd.d/tftp` file and make it look like this:

```
service tftp
{
    protocol        = udp
    port            = 69
    socket_type     = dgram
    wait            = yes
    user            = root
    server          = /usr/sbin/in.tftpd
    server_args     = --secure --user tftp /srv/tftp
    disable         = no
}
```

Explanation:

`disable = no`	Enable spawning of TFTPD process
`user = root`	Start tftp daemon from root user. It's needed to listen privileged port 69 (as it's below 1024) and to make chroot for `/srv/tftp` directory
`server_args = --secure`	Use relative (rather than absolute) paths
`server_args = --user tftp`	Read files with this user's permissions
`server_args = /srv/tftp`	Path to TFTP files

See "`man tftpd`" for details.

Stop TFTP server and restart super-server:

```
$ sudo service tftpd-hpa stop
$ sudo service xinetd restart
```

Prepare TFTP files:

```
$ sudo mkdir -p /srv/tftp
$ sudo cp ~/repos/linux-stable/arch/arm/boot/zImage /srv/tftp
$ sudo cp ~/repos/linux-stable/arch/arm/boot/dts/am335x-boneblack.dtb /srv/tftp
$ sudo cp ~/repos/busybox/uRamdisk /srv/tftp
$ sudo chown -R tftp:tftp /srv/tftp
```

Note that you can exclude the whole `boot/` directory from that ramdisk, as we will use zImage and dtb files obtained over TFTP to boot the system.

3.4.2.2 Configuring the Ethernet interface

We won't setup DHCP server here, but rather will use static IP addresses for simplicity. Also, we're assuming that you have at least one not used Ethernet interface on your host machine (i.e. you have Internet connection on some another interface). If that's not the case, please add one (e.g. by inserting Ethernet card to your PC, or by using USB-to-Ethernet adapter if you use laptop).

First, let's identify the name of your Ethernet interface:

```
$ ip addr
```

It should be something like `eth0` or `enp0s3`. Further we will use `eth0` in instructions, but **you should use the one you found out from command above**.

Now let's set static IP address for Ethernet interface. Edit this file:

```
$ sudo vim /etc/network/interfaces
```

and add next lines to it:

```
# Ethernet interface for connection with dev board
auto eth0
iface eth0 inet static
    address 192.168.0.1
    netmask 255.255.255.0
```

Now issue next commands, for that configuration to come into effect:

```
$ sudo ip addr flush eth0
$ sudo service networking stop
$ sudo service networking start
```

Make sure your Ethernet interface has an address of 192.168.0.1:

```
$ ip addr show eth0
```

Now your Ethernet interface is configured and you should be able to ping your laptop from the board. Changes will be permanent even after next reboot. This instructions should be suitable for Ubuntu and all other Debian-based systems.

3.4.3 Run TFTP boot on device

NOTE: When copying multi-line command in U-Boot, be sure that it's not split up into several lines:

- either copy multi-line command to the editor and make it single line, and only then copy to U-Boot
- or type in the whole command manually (in that case watch out for typos)
- also, it may be helpful to enable line-wrapping in minicom, by pressing **Ctrl-A, W**.

Prepare U-Boot environment for TFTP boot from RAM:

```
=> setenv ipaddr 192.168.0.100
=> setenv serverip 192.168.0.1
=> setenv rdfile uRamdisk
=> setenv netloadrd 'tftp ${rdaddr} ${rdfile}'
=> setenv tftpramboot 'run findfdt; setenv autoload no; run netloadfdt; run
netloadimage; run netloadrd; run ramboot'
=> env save
```

Check that TFTP server is alive:

```
=> ping 192.168.0.1
```

Now you can use “tftpramboot” command (it’ll be also available on next boot, as it was stored to the environment). Run it to perform TFTP boot:

```
=> run tftpramboot
```

Notice next lines in kernel log:

```
[...] Kernel command line: ... root=/dev/ram0 ...
[...] Trying to unpack rootfs image as initramfs...
[...] Freeing initrd memory: ...
```

That’s how you will know your kernel was booted with ramdisk (not mounting rootfs from eMMC).

3.5 NFS Boot

NFS boot is a network boot with next procedure:

1. Obtain kernel and device tree files via TFTP (in U-Boot)
2. Run kernel specifying that rootfs must be used from NFS server
3. RootFS will be mounted via network from your host machine

So first of all, make sure you are able to do TFTP boot (described in previous section).

To make sure your kernel is able to mount RootFS via NFS, next options must be enabled as **built-in** (=y):

- `CONFIG_ROOT_NFS=y`: Support for mounting rootfs via NFS
- `CONFIG_TI_CPSW=y`: Driver for Ethernet controller on BeagleBone Black

Those options are enabled in our defconfig already, you can check it in `.config` file (in kernel source code).

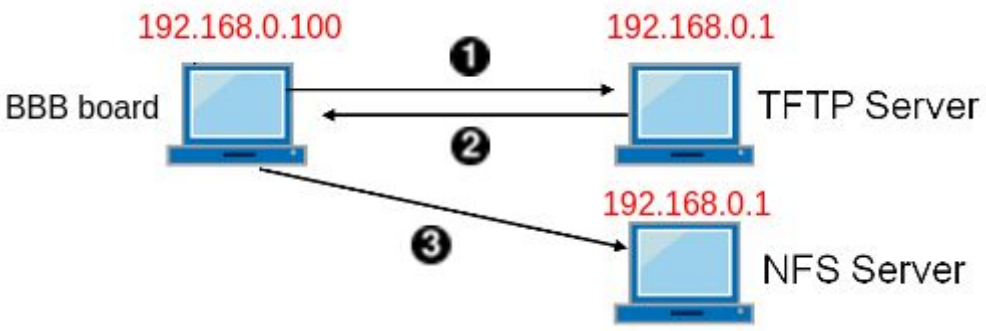


Fig 3.2 - NFS boot with static IP

3.5.1 Host configuration

Install NFS server:

```
$ sudo apt install nfs-kernel-server
```

Prepare RootFS for NFS access:

```
$ sudo mkdir -p /srv/nfs/busybox
$ sudo cp -r ~/repos/busybox/_install/. /srv/nfs/busybox
$ sudo chown -R root:root /srv/nfs
```

Notice that we will see the same owner and group for RootFS files on the board, as we have in `/srv/nfs/busybox` on host machine. And we want all RootFS files to be owned by root.

Update NFS exports file:

```
$ sudo vim /etc/exports
```

and add the next line there:

```
/srv/nfs/busybox 192.168.0.100/24(rw,sync,no_root_squash,no_subtree_check)
```

That means we want NFS server to provide `/srv/nfs/busybox` directory to NFS client with IP address of 192.168.0.100. Notice "`no_root_squash`" option: it allows us to create or change files with root ownership on board, and it will be reflected on NFS server. Refer to "`man exports`" for details.

Start NFS server and ensure that our RootFS is exported:

```
$ sudo service nfs-kernel-server start
$ sudo exportfs -a
```

Check export list for our NFS server:

```
$ sudo showmount -e
```

3.5.2 Run TFTP boot with NFS rootfs

Given that TFTP boot from previous section was successfully configured, we can now extend those instructions to make kernel use rootfs from NFS server. Let's configure TFTP boot with RootFS mounted from NFS in U-Boot for our case (static configuration without DHCP):

```
=> setenv ipaddr 192.168.0.100
=> setenv serverip 192.168.0.1
=> setenv rootpath /srv/nfs/busybox
=> setenv nfsopts nolock,nfsvers=3
=> setenv netboot 'run findfdt; setenv autoload no; run netloadimage; run
netloadfdt; run netargs; bootz ${loadaddr} - ${fdtaddr}'
=> setenv netargs 'setenv bootargs console=${console} ${optargs} root=/dev/nfs
nfsroot=${serverip}:${rootpath},${nfsopts} rw ip=${ipaddr}'
=> env save
=> run netboot
```

Be sure to specify "`nfsvers=3`", without this param the kernel will hang, unable to mount RootFS via NFS. See "`man nfs`" for details on `nfsopts` params.

All changes made in RootFS will be reflected on your host machine (in `/srv/nfs/busybox`).

When BBB boot is finished, run "`mount`" command to check that your RootFS is mounted from NFS server. You can also check kernel log for related information.

Appendix A: USB Hotplug Explanation

For USB Ethernet adapter we need r8152 module to be loaded. This can be done either via insmod/modprobe command, or it can be done automatically by hotplug mechanism. Hotplug works as follows:

- Kernel notifies user-space via uevent (see corresponding uevent file in sysfs, /sys/bus/usb/….); it has MODALIAS string set by kernel
- Hotplug helper is being triggered by kernel (/proc/sys/kernel/hotplug)
- We have set mdev as hotplug helper, so it's being called (without parameters)
- mdev runs mdev rule from /etc/mdev.conf, running modprobe command
- modprobe command receives modalias as parameter and looks into modules.alias file to find corresponding module file to load; when found, kernel module (in our case it's r8152.ko) is being loaded

We can see next dmesg messages (in minicom) when inserting USB Ethernet adapter:

- USB device is found:

```
usb 1-1: new high-speed USB device number 2 using musb-hdrc
usb 1-1: New USB device found, idVendor=2357, idProduct=0601
usb 1-1: New USB device strings: Mfr=1, Product=2, SerialNumber=6
usb 1-1: Product: USB 10/100/1000 LAN
usb 1-1: Manufacturer: TP-LINK
usb 1-1: SerialNumber: 000001000000
```

- Hotplug loads corresponding module automatically:

```
usbcore: registered new interface driver r8152
usb 1-1: reset high-speed USB device number 2 using musb-hdrc
r8152 1-1:1.0 eth1: v1.09.9
```

Appendix B: Power Management on BBB

The AM335x device contains a dedicated Cortex-M3 processor to handle the power management transitions.. But to make it work, we need to upload the firmware to it first. Read more here:

www.ti.com/lit/an/sprac74a/sprac74a.pdf

Clone the firmware and copy it to kernel source dir:

```
$ cd ~/repos
$ git clone git://git.ti.com/processor-firmware/ti-amx3-cm3-pm-firmware.git
$ cd ti-amx3-cm3-pm-firmware
$ git checkout ti2018.02
$ mkdir ~/repos/linux-stable/firmware
$ cp bin/am335x-pm-firmware.elf ~/repos/linux-stable/firmware
```

Build the kernel with next options enabled (embed FW to zImage):

```
# --- PM (see sprac74a.pdf) ---
# Embed firmware in kernel
CONFIG_EXTRA_FIRMWARE="am335x-pm-firmware.elf"
CONFIG_EXTRA_FIRMWARE_DIR="firmware"
# Firmware Loading from rootfs
#CONFIG_FW_LOADER_USER_HELPER=y
#CONFIG_FW_LOADER_USER_HELPER_FALLBACK=y
# AMx3 Power Config Options
CONFIG_MAILBOX=y
CONFIG_OMAP2PLUS_MBOX=y
CONFIG_REMOTEPROC=y
CONFIG_WKUP_M3_RPROC=y
CONFIG_WKUP_M3_IPC=y
CONFIG_SOC_TI=y
CONFIG_TI_EMIF_SRAM=y
CONFIG_AMX3_PM=y
# RTC
CONFIG_RTC_DRV_OMAP=y
```

Copy new zImage, dtb and modules to your rootfs, re-flash. Test suspend/resume:

```
# echo enabled > /sys/class/tty/ttyO0/power/wakeup
# echo mem > /sys/power/state
```

Press any key to resume from suspend.