BeagleBone Black: Platform Bring Up with Upstream Components
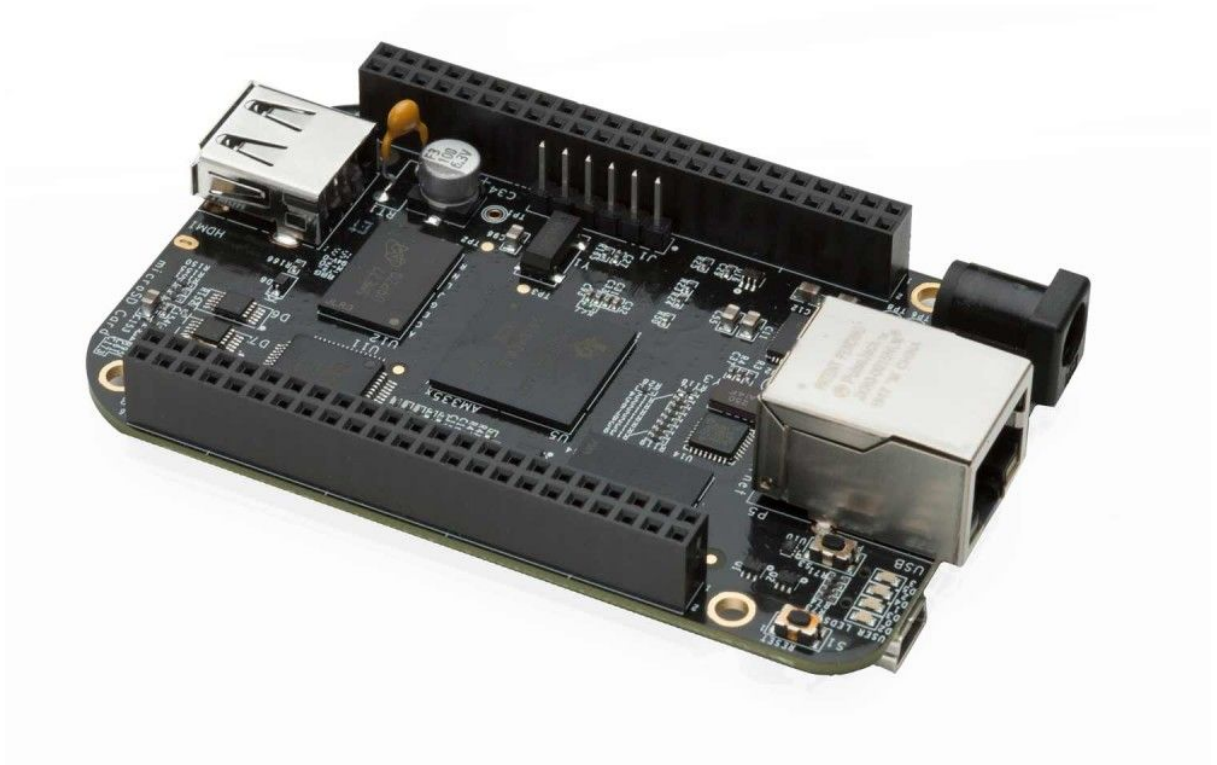
Sam Protsenko, 2018

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 PC GNU/Linux distributions.

Why use upstream software? Learning about upstream software will give you good understanding of community's 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 HiKey board probably should be used.

These instructions were tested on **Ubuntu 16.04 LTS**, but should work on any Debian-based distros. Differences (if any) should be minor, obvious and easy to fix, even by students. We recommend to use latest LTS Ubuntu, as it's up-to-date, easy to start and stable OS.

BeagleBone **Rev C** board was used (latest one, up to this point). We recommend you to use Rev C revision or newer. Most instructions should work on Rev B and Rev A5A, too.

1 Preparing the tools

1.1 Initial Ubuntu setup

Install tools that will be needed further:

```
$ sudo apt-get update
$ sudo apt-get install aptitude vim vim-gtk
```

`aptitude` is a recommended tool in Debian for dealing with APT package manager, so we will stick to using it. Please note that you should update packages list each time before installing new packages.

We recommend to use `vim` as a standard editor. It's good to know how to use it, because `vi` can be found on virtually every Linux distribution. You can use any other editor, of course.

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 consequently, 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 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 aptitude 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 Setup serial console

Serial console is basically a remote terminal for your board. This way 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):
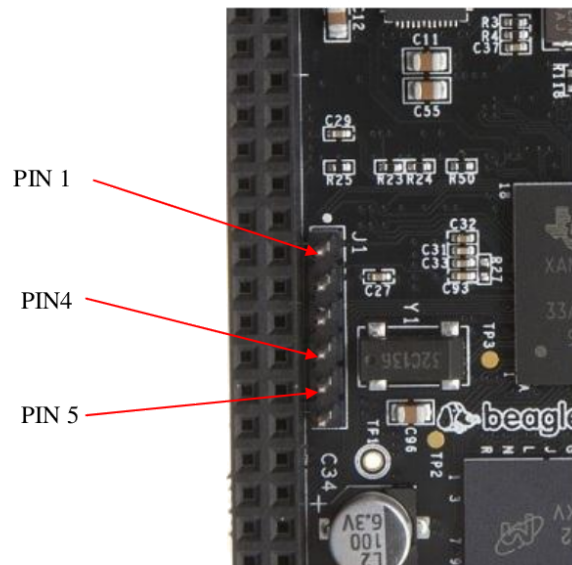

Fig. 1.1 - BBB Serial Port (J1 header)

Please notice that **PIN #1** is a **Ground** line (PIN 1 is marked with big white dot on the board).

○ 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 to use, 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 (as `picocom` and `kermit`), but let's stick with `minicom` for now

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 aptitude 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, sending the break (needed for issuing [Magic SysRq](#) keys), etc.

1.4 Udev rules for OTG USB

OTG USB connection can be used for two main purposes:
1. Flashing new software (images) to eMMC (using `dfu-util` or `fastboot` tool)
2. Using the board as a USB gadget (can be configured via ConfigFS in Linux)

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"
```

Restart udev:

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

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, too.

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*) from x86_64 PC (*host*). 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" file you will see something like "arm-eabi-gcc" file, 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 will 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/latest-7/arm-eabi/

   This file should be downloaded:

   gcc-linaro-7.2.1-2017.11-x86_64_arm-eabi.tar.xz

2. Extract toolchain to /opt:

   ```
   $ sudo tar xJvf gcc-linaro-7.2.1-2017.11-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/latest-7/arm-linux-gnueabihf/

   This file should be downloaded:

   gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabihf.tar.xz

2. Extract toolchain to /opt:

```
$ sudo tar xJvf gcc-linaro-7.2.1-2017.11-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 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 or prefix (if needed)
9. Flash the built software to the device

2.1 U-Boot

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

Obtain U-Boot source code:

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

Checkout to latest release tag (but not to some release candidate -rc tag):

```
$ git tag | tail
$ git checkout v2018.03
```

Configure toolchain environment in shell:

```
$ export PATH=/opt/gcc-linaro-7.2.1-2017.11-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:

11

- `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.

Install host build dependencies:

```
$ sudo aptitude install ncurses-dev libssl-dev bc bison flex
```

You can find clone link on Linux kernel site here:

https://www.kernel.org/

Obtain Linux kernel source code:

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

Checkout to stable branch 4.16:

```
$ git checkout linux-4.16.y
```

Configure toolchain environment in shell:

```
$ export PATH=/opt/gcc-linaro-7.2.1-2017.11-x86_64_arm-eabi/bin:$PATH
$ export CROSS_COMPILE='ccache arm-eabi-'
$ export ARCH=arm
```

We will use `multi_v7_defconfig` as a base, but we still need to enable some kernel options on top of it. 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

# Enable USB on BBB (AM335x)
CONFIG_AM335X_PHY_USB=y
CONFIG_NOP_USB_XCEIV=y
CONFIG_OMAP_USB2=y
CONFIG_USB=y
CONFIG_USB_ANNOUNCE_NEW_DEVICES=y
CONFIG_USB_CONFIGFS=y
CONFIG_USB_EHCI_HCD=y
CONFIG_USB_EHCI_ROOT_HUB_TT=y
CONFIG_USB_GADGET=y
CONFIG_USB_GPIO_VBUS=y
# For USB keyboard and mouse
CONFIG_USB_HID=y
CONFIG_USB_HIDDEV=y
CONFIG_USB_INVENTRA_DMA=y
CONFIG_USB_MUSB_AM35X=y
CONFIG_USB_MUSB_DSPS=y
CONFIG_USB_MUSB_HDRC=y
CONFIG_USB_MUSB_OMAP2PLUS=y
CONFIG_USB_MUSB_TUSB6010=y
CONFIG_USB_OHCI_HCD=y
CONFIG_USB_SERIAL_GENERIC=y
# For USB mass storage devices (like flash USB stick)
CONFIG_USB_STORAGE=y
CONFIG_USB_TI_CPPI41_DMA=y
CONFIG_USB_TUSB_OMAP_DMA=y
CONFIG_USB_ULPI=y
CONFIG_USB_ULPI_BUS=y

# Networking
CONFIG_BRIDGE=y
```

Generate kernel `.config` file from `multi_v7_defconfig` and 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:

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

Setup build environment (use Linux toolchain):

```
$ export ARCH=arm
$ export PATH=/opt/gcc-linaro-7.2.1-2017.11-x86_64_arm-linux-gnueabihf/bin:$PATH
$ export CROSS_COMPILE="ccache arm-linux-gnueabihf-"
```

Checkout to latest stable branch:

```
$ git branch -a | grep stable
$ git checkout 1_28_stable
```

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
```

Populating 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):

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

2.3.3 Populate /boot

Populate `/boot` rootfs directory with kernel files:

```
$ cd _install/boot
$ cp $WORK_DIR/linux/arch/arm/boot/zImage .
$ cp $WORK_DIR/linux/arch/arm/boot/dts/am335x-boneblack.dtb .
$ cp $WORK_DIR/linux/System.map .
$ cp $WORK_DIR/linux/.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

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

```
$ cd $WORK_DIR/linux
$ export INSTALL_MOD_PATH=$WORK_DIR/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`), which can be seen 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 module loading on hotplug):

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

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}`

3 Flashing and Booting

3.1 SD card

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.

TODO: Link SD card format script from TI Linux SDK here
for now this script can be used:
> https://wiki.linaro.org/Boards/BeagleBoard-X15#Format_SD_card

3.1.1 Create bootable SD card with BusyBox rootfs

Insert SD card. Unmount partitions, if they were mounted automatically (e.g. by Ubuntu):

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

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

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

Unmount SD card:

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

3.1.2 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) may not work, so don't rely on it, and reconnect the power instead, as described above.

BusyBox will be loaded from SD card.

3.2 eMMC

3.2.1 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):

Install `make_ext4fs` tool:

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

Check `_install/` directory size:

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

In our case it's 88 MiB. Let's stick to 96 MiB image size (100,000,000 bytes), because we need some extra size for ext4 FS info.

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

```
$ make_ext4fs -L rootfs -l 100000000 rootfs.ext4 _install
```

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

3.2.2 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 aptitude 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.

3.2.3 Flashing eMMC via USB using DFU

Install `dfu-util` tool:

```
$ sudo aptitude install dfu-util
```

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

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

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 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
```

Check on host machine 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:

```
$ 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:

```
$ dfu-util -D rootfs.ext4 -a rootfs
```

3.2.4 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 aptitude 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.

TODO: Provide comparison table for all flashing and booting options: speed, usual use-cases, pros/cons (dfu for bootloader, fastboot for rootfs)

3.2.5 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
```

3.3 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
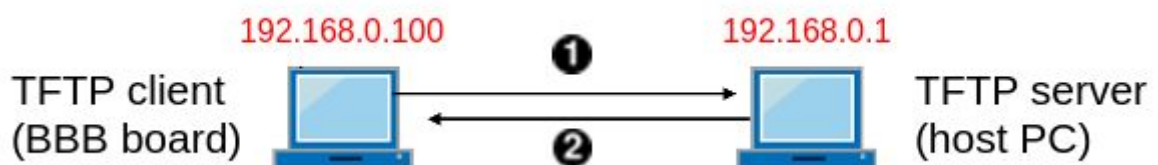- you need to do images download via TFTP for each new boot

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.

3.3.1 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 aptitude install u-boot-tools
```

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

```
$ cd _install
$ find . | cpio -o -H newc | gzip > ../ramdisk.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 ramdisk.cpio.gz   \
        uRamdisk
```

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

3.3.2 Host 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 aptitude 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 arch/arm/zImage /srv/tftp
$ sudo cp arch/arm/boot/dts/am335x-boneblack.dtb /srv/tftp
$ sudo cp 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.

We won't setup DHCP server here, but rather will use static IP addresses for simplicity. Configure your host machine Ethernet interface:

```
$ sudo ifconfig eth0 192.168.0.1 up
```

3.3.3 Run TFTP boot on device

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).

3.4 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
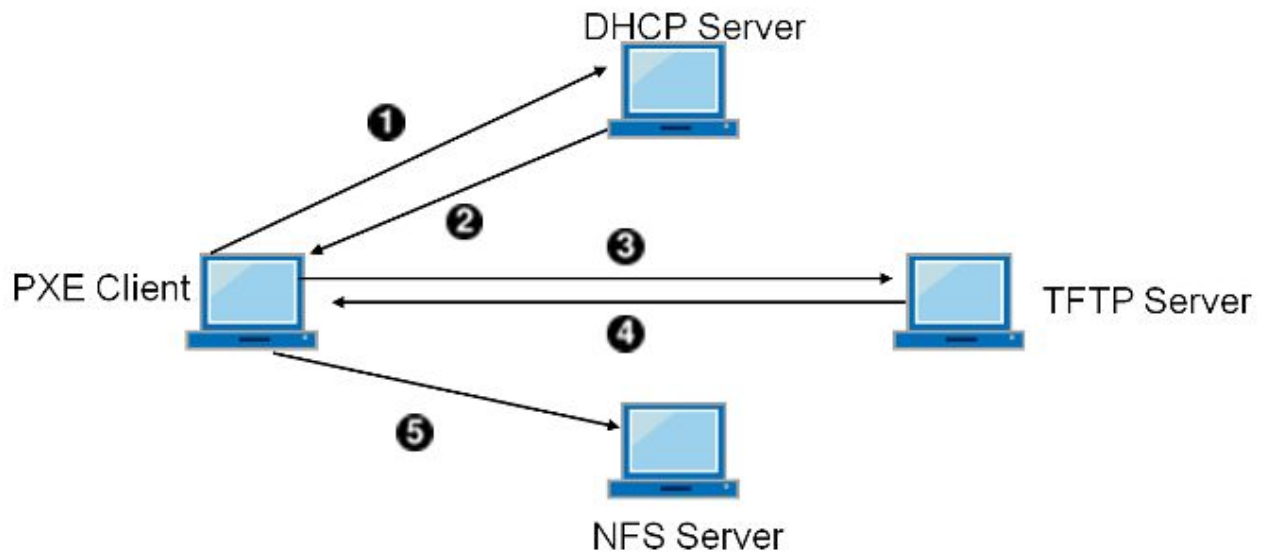
3.4.1 Host configuration

Install NFS server:

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

Prepare RootFS for NFS access:

```
$ sudo mkdir -p /srv/nfs/busybox
$ cp -r $WORK_DIR/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
```

3.4.2 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 'echo Booting from network ...; 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.

4 Working in BusyBox rootfs

4.1 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
```