1. Toolchain

Toolchain is a bunch of tools for cross-compiling programs for the architecture different than the 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 following 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 a 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 the path to toolchain's `bin/` directory 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 the compiled project to run the correct tool, like `${CROSS_COMPILE}gcc`).

1.1 Baremetal toolchain

Baremetal toolchain is used to build "bare" programs, which are being run without Linux environment (no system calls, no libc). It is usually used for building firmwares, bootloaders, Linux kernel. Most popular toolchains are GCC-based and clang-based. We will use GCC one, as it's recommended and only supported toolchain for Linux kernel and U-Boot. Previously GCC toolchains were developed by Linaro, but now the same toolchains (newer versions) are available at ARM site.

1. Download baremetal toolchain for Cortex-A processors (32-bit) from here:

   https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads

   This file should be downloaded:

   gcc-arm-8.3-2019.03-x86_64-arm-eabi.tar.xz

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

2. Extract this toolchain to `/opt`:

```
$ sudo tar xJvf gcc-arm-8.3-2019.03-x86_64-arm-eabi.tar.xz -C /opt/
```

1.2 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 Cortex-A processors (32-bit) from here:

   https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-a/downloads

   This file should be downloaded:

   gcc-arm-8.3-2019.03-x86_64-arm-linux-gnueabihf.tar.xz

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

2. Extract this toolchain to `/opt`:

```
$ sudo tar xJvf gcc-arm-8.3-2019.03-x86_64-arm-linux-gnueabihf.tar.xz \
  -C /opt/
```

2 Obtaining and Building the Software

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 https://gitlab.denx.de/u-boot/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 v2019.07
```

Also, please apply next patch series (it's still pending on review):

```
$ curl https://patchwork.ozlabs.org/series/130450/mbox/ | git am
```

Configure toolchain environment in shell:

```
$ export PATH=/opt/gcc-arm-8.3-2019.03-x86_64-arm-eabi/bin:$PATH
$ export CROSS_COMPILE='ccache arm-eabi-'
$ export ARCH=arm
```

NOTE: please **do not** put these commands to your `~/.bashrc`, to avoid possible issues.

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 this **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-arm-8.3-2019.03-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 the 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_31_stable
```

Setup build environment (use Linux toolchain):

```
$ export ARCH=arm
$ export PATH=/opt/gcc-arm-8.3-2019.03-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 a 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 meet 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 for you. But for educational purposes we want to use the absolute minimum, to only see the essential parts, and not to be distracted by something fancy.