Android Training

Lab Book

bootlin
http://bootlin.com

March 21, 2018

About this document

Updates to this document can be found on http://bootlin.com/doc/training/android/.

This document was generated from LaTeX sources found on https://git.bootlin.com/training-materials.

More details about our training sessions can be found on http://bootlin.com/training.

Copying this document

Training setup

Download files and directories used in practical labs

Install lab data

For the different labs in this course, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
cd
wget http://bootlin.com/doc/training/android/android-labs.tar.xz
tar xvf android-labs.tar.xz
```

Lab data are now available in an `android-labs` directory in your home directory. For each lab there is a directory containing various data. This directory will also be used as working space for each lab, so that the files that you produce during each lab are kept separate.

You are now ready to start the real practical labs!

Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `chown -R myuser.myuser linux/`
- If you are using Gnome Terminal (the default terminal emulator in Ubuntu 16.04), you can use tabs to have multiple terminals in the same window. There's no more menu option to create a new tab, but you can get one by pressing the `[Ctrl] [Shift] [t]` keys.

Android source code

Download the source code for Android and all its components

During this labs, you will:

- Install all the development packages needed to fetch and compile Android
- Download the `repo` utility
- Use `repo` to download the source code for Android and for all its components

Setup

Look at the space available on your root partition using the `df -h` command. The first line should display the available storage size on your root partition. If you have more than 40GB, go to the `/opt` directory.

```
sudo mkdir android
sudo chown -R <user>:<group> android
cd android
```

If you don't have that much space available, go to the `$HOME/android-labs/source` directory.

Install needed packages

Install the packages needed to fetch Android's source code:

```
sudo apt-get install git curl
```

Fetch the source code

Android sources are made of many separate Git repositories, each containing a piece of software needed for the whole stack. This organization adds a lot of flexibility, allowing to easily add or remove a particular piece from the source tree, but also introduces a lot of overhead to manage all these repos.

To address this issue, Google created a tool called Repo. As Repo is just a python script, it has not made its way in the Ubuntu packages, and we need to download it from Google.

```
mkdir $HOME/bin
export PATH=$HOME/bin:$PATH
curl http://commondatastorage.googleapis.com/git-repo-downloads/repo > \
    $HOME/bin/repo
chmod a+x $HOME/bin/repo
```

We can now fetch the Android source code:

```
repo init -u https://android.googlesource.com/platform/manifest \
     -b android-4.3_r2.1
repo sync -c -j2
```

`repo sync -c` will synchronize the files for all the source repositories listed in the manifest. If you have a slow connection to the Internet, this could even take a few hours to complete. Fortunately, your instructor will take advantage of this waiting time to continue with the lectures.

The `-c` option makes it download only the branch we will be using, speeding up the download.

We are using the `-jX` option of `repo sync` to have `X` downloads in parallel. Otherwise, if the source repositories are fetched sequentially, a lot of time and available network bandwidth is wasted between downloads, processing the files downloaded by git.

In our case, we are using `2` because we all share the same internet connection, but you can definitely increase this number up to `8` when you are alone.

If this really takes too much time, your instructor may distribute a ready-made archive containing what `repo sync` would have downloaded.

To save time, do not wait for the `repo sync` command to complete. You can already jump to the next section.

Install packages needed at compile time

While `repo sync` runs, download the packages that you will need to compile Android and its components from source:

```
sudo apt-get install xsltproc gnupg flex bison gperf build-essential \
     zlib1g-dev libc6-dev lib32ncurses5-dev ccache unzip zip\
     x11proto-core-dev libx11-dev lib32readline6-dev lib32z1-dev \
     libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown \
     libxml2-utils
sudo apt-get clean
```

Again, if you have a slow connection to the Internet, installing these packages could take several tens of minutes. Anyway, we are getting back to the lectures.

Install the Java JDK

The Android build system requires the Oracle Java6 JDK, that is too old to be bundled in Ubuntu. We'll have to fetch it from another source, and install it.

To do so, we first need to add the repository that holds the JDK nowadays.

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
```

Now, we can install the latest JDK version with the command:

```
sudo apt-get install oracle-java6-installer
```

Follow the steps asked, and you should be all set!

Install the ccache

Jelly Bean takes a lot of time to compile. Fortunately, we can use a tool called `ccache` to speed it up. `ccache` caches the object files compiled for a given project, so that whenever you need them once again, you don't have to actually cache them.

Ask your trainer for a pre-filled ccache archive that we will use to speed up a lot the compilation. Once you have retrieved the archive, extract it in either the `/opt/` or `$HOME/android-labs/` directories.

First compilation

Get used to the build mechanism

During this lab, you will:

- Configure which system to build Android for
- Compile your first Android root filesystem

Setup

Stay in the Android source directory.

Set up ccache

To be able to use `ccache` properly, we need to apply a commit present in the AOSP master that will bump the version of ccache to a more recent and much more adapted version of `ccache`.

We will first need to create a new branch where we will store the commit doing such a change. To do that, we will obviously use `repo`, and especially the `start` sub-command, on the `prebuilts/misc` repository. We can call that branch `ccache` for example.

Now, go in the `prebuilts/misc` directory, and use the command `git cherry-pick 81012983`. This will apply the given commit in our current branch, and since we have created a new branch, `repo` will carry our commits, even if we do `repo sync`.

Build environment

Now that `repo sync` is over, we will compile an Android system for the emulator. This will include building the emulator itself.

First, run `source build/envsetup.sh`.

This file contains many useful shell functions and aliases, such as `croot` to go to the root directory of the Android source code or `lunch`, to select the build options.

Check your `PATH` environment variable:

```
echo $PATH
```

You will see that `build/envsetup.sh` hasn't modified your `PATH`. This will be done during the build job.

The target product for the emulator is *full*, and we want to have an engineering build. To do this, run `lunch aosp_arm-eng`

Now, to tell Android to use `ccache` whenever possible, we need to set a few environment variable.

```
export USE_CCACHE=1
export CCACHE_DIR=/opt/ccache
```

or, depending on your setup,

```
export CCACHE_DIR=$HOME/android-labs/ccache
```

And that's it!

Compile the root filesystem

The build system will use the proper setup to build this target. Before running `make`, first check the number of CPU cores in your workstation, as seen by the operating system (hyperthreading could make your OS see 4 cores instead of 2, for example).

```
cat /proc/cpuinfo
```

You can use `make -j` (`j` stands for *jobs* to instruct `make` to run multiple compile jobs in parallel, taking advantage of the multiple CPU cores, and making sure that I/Os and CPU cores are always at 100%. This will significantly reduce build time.

For example, if Linux sees 4 cores, you will get good performance by specifying the double number of parallel jobs:

```
make -j 8
```

While this command runs, make sure you are actually using ccache properly: in another terminal, set the `CCACHE_DIR` environment variable like we did, and use the command

```
watch -n1 -d prebuilts/misc/linux-x86/ccache/ccache -s
```

After a few minutes, you should see the number of cache hits increasing, while the number of cache misses remain very low. If that's not the case, call your instructor.

Go grab (several cups of) coffee!

Known issue In rare circumstances, some builds can fail when `make` is run with multiple jobs in parallel. If this happens, you can run `make` with no option after the failure, and this is likely to make the issue disappear.

Test your Android build

Now, look at the `PATH` environment variable again.

You can see that it contains:

- `$HOME/android-labs/source/out/host/linux-x86/bin/`. That's where new utilities have just been compiled.
- Prebuilt executables, in particular the ARM cross-compiling toolchain, which was already present in the repositories fetched by `repo`.

Look at the contents of the `out/target/product/generic` folder. That's where the system images have been built.

Run the `emulator` command. It will run the emulator with these generated images. Wait a few minutes for the emulator to load the system, and then check the build number in the `Settings` application in Android.

Note The `PATH` settings automatically added by the build process will be lost if you close your terminal. We will see how to restore this environment in the next lab.

Compile and boot an Android Kernel

Learn how to build an Android Kernel and test it

After this lab, you will be able to:

- Extract the kernel patchset from Android Kernel
- Compile and boot a kernel for the emulator

Setup

Go to the `$HOME/android-labs/` directory.

Compile a kernel for the Android emulator

The Android emulator uses QEMU to create a virtual ARMv7 SoC called Goldfish.

In the standard Android source code we fetched, the kernel source was not included. So the first thing to do is download the Android kernel sources stored in the git repository `https://android.googlesource.com/kernel/goldfish.git` using repo's local manifests. Store it into a folder called `kernel`.

Make repo download the kernel, and once done, using `git branch -a`, find the name of the most recent stable kernel version that supports the Goldfish platform. At the time of this writing, Linux 3.10 still don't seem to work with the emulator, and you will have to choose either the 3.4 or the 2.6.29 kernels. Since the latter is quite ancient, we'll use 3.4.

Edit your local manifest to use the revision you just identified.

Now that we have kernel sources, we now need a toolchain to compile the kernel. Fortunately, Android provides one. First, add the toolchain from the previous lab to your `PATH`.

```
PATH=$HOME/android-labs/source/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/bin:$PATH
```

Now, configure the kernel for the target platform

```
ARCH=arm make goldfish_armv7_defconfig
```

Then, configure the kernel using the tool of your choice to add a custom suffix to the kernel version and compile the kernel!

```
ARCH=arm CROSS_COMPILE=arm-eabi- make -j 4
```

A few minutes later, you have a kernel image in `arch/arm/boot`.

Boot your new kernel

To run the emulator again, this time with your new kernel image, you have to restore the environment that you had after building Android from sources. This is needed every time you reboot your workstation, and every time you work with a new terminal.

Go back to `$HOME/android-labs/source`.

All you need to do is source the environment and run the `lunch` command to specify which product you're working with[1]:

```
source build/envsetup.sh
lunch aosp_arm-eng
```

Now, test that you can still run the emulator as you did in the previous lab.

To run the emulator with your own kernel, you can use the `-kernel` option:

```
emulator -kernel $HOME/android-labs/kernel/arch/arm/boot/zImage
```

Once again, check the kernel version in the Android settings. Make sure that the kernel was built today.

Generate a patchset from the Android Kernel

Go back to the `$HOME/android-labs/kernel` directory.

To compare the Android kernel sources with the official kernel ones, we need to fetch the latter sources too, allowing us to see which commits differ. Git is great for that and offers everything we need.

First, add the `linux-stable` repository from Greg Kroah-Hartmann to our repository:

```
git remote add stable git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-
stable.git
```

Now, we need to fetch the changes from this remote repository:

```
git fetch stable
```

A while later, you should have all the changes in your own repository, browsable offline!

Now, open the main `Makefile` with your favorite editor and get the kernel version that is in it. This corresponds to the kernel version used by Android for the emulator. It should be 2.6.29 if you chose that version. Now, let's have a look at the set of commits between the mainline kernel sources and the Android kernel sources (`HEAD`):

```
git log v<kernel-version>..HEAD
```

Now, extract the corresponding set of patches[2]:

```
git format-patch v<kernel-version>..HEAD
```

In this set of patches, find the one that adds the wakelock API.

Congratuations! You now have all the patches to apply to a vanilla kernel version to obtain a full Android kernel.

In this lab, we showed useful Git commands, but didn't go into details, as this is not a Git course. Don't hesitate to ask your instructor for more details about Git.

[1]Remember that Android sources allow you to work with multiple products
[2]Git generates one patch per commit.

Supporting a new board

Learn how to make Android on new hardware

After this lab, you will be able to:

- Boot Android on a real hardware
- Troubleshoot simple problems on Android
- Generate a working build

Download the source code

Go to the directory holding the Android source code (either `/opt/android` or `$HOME/android-labs/source`).

While still using AOSP, we will use the configuration done in TI-flavored Android build named *rowboat*. This should make most of the port mostly ready to use, and only a few things here and there will be needed.

`repo` allows to manage your Android source tree in a smart way, downloading only the differences between the new manifest we would give it and the current code we have. In order to integrate these additions nicely, the first thing we need is to add rowboat's configuration repo to the manifest.

To do so, create a local manifest to add the repository `git://git.bootlin.com/android/training/vendor-ti-beagleboneblack`, at revision `rowboat-jb-4.3-fe`, and store it to `device/ti/beagleboneblack`

Once done, you can run `repo sync -c` again. It will download the new git tree we just added and add it to our source code. Note that you may have to use `repo sync --force-sync kernel` to force syncing over the previously fetched kernel sources.

Build the kernel

Obviously, the Beaglebone Black needs a kernel of its own. The way it's usually done in Android is by first compiling the kernel, and then configuring the build system to use that binary.

So, the first step is to download the kernel. Once again, we can use repo for that. Make it fetch the kernel from `https://gitlab.com/ozgurkeles/bbb-kernel`, at revision `rowboat-am335x-kernel-3.2`, in the `kernel` directory.

Once done, make `repo` download it, and we can start building our kernel. This time, you'll need to use the `am335x_evm_android` default configuration.

You can then start the compilation.

Once the compilation is over, we need to make the build system aware of the image we just compiled.

To do so, we need a few things. First we need to move the `zImage` we just compiled to our device folder.

Then, we need to tell Android that it will need to generate a `boot.img` with the brand new kernel we have. This is done through the `TARGET_NO_KERNEL` variable. Obviously, this will need to be disabled. You can find where this variable is defined using `grep`.

In order for Android to be able to properly generate the boot image, it also needs to know to which base address and which command line it needs. This is done by the `BOARD_KERNEL_BASE` and `BOARD_KERNEL_CMDLINE` variables. You can see that `BOARD_KERNEL_BASE` is already set, while `BOARD_KERNEL_CMDLINE` isn't. In order for Android to show its boot traces on the serial link, we need to set it to

```
console=ttyO0,115200n8 androidboot.console=ttyO0
```

You also need to change `BOARD_KERNEL_BASE` to `0x80008000`.

Finally, we need to inform the Android build system of where our actual image is. The `mkbootimg` tool is actually looking for a file called `kernel` in the build directory, and it's up to the device configuration to copy it there. Fortunately, the `PRODUCT_COPY_FILES` macro is doing just that, so we'll need to add our kernel copying to list of files to copy, with the syntax:

```
<path/to/source/image>:kernel
```

And we should be done for now.

Build Android for the BeagleBone Black

No, we have gained support for the BeagleBone Black we're using. To compile Android for it, we have to use lunch, in the same way we did previously:

```
source build/envsetup.sh
lunch beagleboneblack-eng
```

Make sure we are using `ccache`:

```
export USE_CCACHE=1
export CCACHE_DIR=$HOME/android-labs/ccache
```

or, depending on your setup,

```
export CCACHE_DIR=/opt/ccache
```

Then, we can start the compilation:

```
make -jX
```

Once again, you can expect this build job to take quite a long time (an hour or so) to run, even on a recent and rather fast laptop.

This job will build four images in `out/target/product/beagleboneblack`: `cache.img`, `boot.img` `ramdisk.img`, `system.img` and `userdata.img`.

We just generated bootable images that we will use on our device, we can now run them on our boards.

Setting up serial communication with the board

To see the board boot, we need to read the first boot messages issued on the board's serial port.

Your instructor will provide you with a special serial cable for the Beaglebone, that is basically a USB-to-serial adapter for the connection with your laptop.

When you plug in this adaptor, a serial port should appear on your workstation: `/dev/ttyUSB0`.

You can also see this device appear by looking at the output of the `dmesg` command.

To communicate with the board through the serial port, install a serial communication program, such as `picocom`[3]:

```
sudo apt-get install picocom
```

You also need to make your user belong to the `dialout` group to be allowed to write to the serial console:

```
sudo adduser $USER dialout
```

You need to log out and in again for the group change to be effective.

Run `picocom -b 115200 /dev/ttyUSB0`, to start serial communication on `/dev/ttyUSB0`, with a baudrate of 115200. If you wish to exit `picocom`, press `[Ctrl][a]` followed by `[Ctrl][x]`.

You should then be able to access the serial console.

Flashing and Booting the Images

To flash the images we previously generated, we will use `fastboot`.

The first thing you need to do is power up the board, and stop the bootloader automatic boot. To do so, press a key in the `picocom` terminal whenever the U-boot countdown shows up. You should then see the U-Boot prompt:

```
U-Boot>
```

You can now use U-Boot. Run the `help` command to see the available commands. To switch to fastboot mode, the only thing you need to do is type the command `fastboot` in this prompt.

The device will then wait for fastboot communications on its USB port.

Configure USB access on your machine

Your instructor will now give you a USB to mini-USB cable. Connect the mini-USB connector to your board, and the other connector to your computer.

We will use this connection for fastboot and, later on, adb.

If you execute any command right now, you are very likely to have an error, because fastboot can't even detect the device. This is because the USB device associated to the BeagleBone doesn't have the right permissions to let you open it.

We need to make sure that `udev` sets the right permissions so that we can access it as a user. To do so, create the file `/etc/udev/rules.d/51-android.rules` and copy the following line:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="<VendorId>", ATTR{idProduct}=="<ProductID>", MODE=
"0600", OWNER="<username>"
```

You can retrieve the vendor and product IDs to use when by using the `lsusb` command, and looking at the line with `Texas Instrument, Inc.`. You should see on this line the IDs to use, with the format `ID <VendorID>:<ProductID>`.

[3] `picocom` is one of the simplest utilities to access a serial console. `minicom` looks more featureful, but is also more complex to configure.

Now, unplug the USB cable and plug it again. You should now be able to use `fastboot` from your user account, with the command `fastboot devices` showing the board properly.

Flashing

On your workstation, first start by formatting the device

```
fastboot oem format
```

Then, you can flash each images independently by using the commands

```
fastboot flash <partition>
```

It will retrieve the matching images from the `out` folder, and will flash them using fastboot. Be sure you flash the `boot`, `cache`, `system` and `userdata` partitions.

Once this is done, reboot using `fastboot reboot`, and you should see Android booting.

However, you'll see that a process keeps crashing. If you use the `logcat` command, you'll see that the crashing process is `zygote`, and more specifically when starting a component called `SurfaceFlinger`.

This component is Android's window manager, and it keeps crashing because we haven't installed the GPU drivers yet.

Adding the GPU drivers

GPU drivers are most of the time implemented using a bunch of user space libraries and kernel modules implemented provided by the GPU vendor.

This is often a real pain to integrate, since it's tied to a particular kernel version. There's several options to integrate them, ranging from wrapping the Android build system invocations into a custom script that will build the driver every time you run it, until rewriting the whole driver Makefile to hook nicely into the build system.

The solution we're going to use is probably one of the easiest, while remaining one of the less evil.

The way we're going to proceed is, just like what we did for the kernel, we're going to build the libraries and modules, and store them in our device folder.

We're then going to add to the device configuration that the build system has to copy these files into the images at compilation time.

The first thing to do is to download the SGX drivers source code from rowboat. Modify your local manifest to download them from `https://git.bootlin.com/android/training/hardware-ti-sgx`, at revision `ti_sgx_sdk-ddk_1.10-jb-4.3-fe`, and store it in `hardware/ti/sgx`.

Once done, open up a new terminal, download it using `repo`. Now, set up the path to use the Android toolchain.

```
export PATH=$(pwd)/prebuilts/gcc/linux-x86/arm/arm-eabi-4.6/bin:$PATH
```

Then, go to the `sgx` folder, and compile it using the following commands:

```
export TARGET_PRODUCT=beagleboneblack
export ANDROID_ROOT_DIR=<path/to/android/source>

make OMAPES=4.x W=1
make OMAPES=4.x install
```

Now, if you look into the folder `device/ti/beagleboneblack/sgx`, you should see some folders with libraries and kernel modules that have been installed by the SGX Makefile.

If you have some spare time, you can take a look at the file `device-sgx.mk` in the beaglebone black device folder to see how we install everything.

Now, you can recompile your images, flash the boot and system images, and you should see the Android splash screen when rebooting!

Fix the blank screen

After the android logo, the screen will turn black. This is actually the backlight turning almost off.

You can find the controls for the backlight in `/sys/class/backlight`. Play with these controls until you find the values that are working.

To make these changes permanently, you will have to edit the kernel code. The used PWM is defined is the `am335xevm` board file, in the `arch/arm/mach-omap2/` folder. The PWM work by switching on and off at a fast pace the power supply to be able to adjust the average voltage delivered between 0 and the actual voltage of the line. Increase the period by a factor of 100.

Once you're done, rebuild the kernel, boot it, and you should be able to read the screen now!

Use ADB

Learn to use the Android Debug Bridge

After this lab, you will be able to use ADB to:

- Debug your system and applications
- Get a shell on a device
- Exchange files with a device
- Install new applications

Setup

Stay in the `$HOME/android-labs/source` directory.

Get ADB

ADB is usually distributed by Google as part of their SDK for Android.

If you were willing to use it, you would first need to go to `http://developer.android.com/sdk/` to get the SDK for Linux, and extract the archive that you downloaded.

Then, you would run the `tools/android` script, and in the `Packages` window, select `Android SDK Platform-tools` under `Tools`, unselect all the other packages, and click on the `Install 1 package...` button. Once done, you would have `adb` installed in `./platform-tools/adb`.

However, the Android source code also has an embedded SDK, so you already have the `adb` binary in the `out/host/linux-x86/bin/`.

To add this directory to your `PATH` after the build done in the previous lab, you can go through the same environment setup steps as earlier:

```
source build/envsetup.sh
lunch beagleboneblack-eng
```

Get logs from the device

ADB provides many useful features to work with an existing Android device. The first we will see is how to get the system logs from the whole system. To do this, just run `adb logcat`.

You will see the device logs on your terminal. This is a huge amount of information though, and it is difficult to find your way in all these lines.

The first thing we can do is download a little wrapper to `adb` to provide colored logs. You can find it here: `http://jsharkey.org/downloads/coloredlogcat.pytxt`. Once downloaded, just run it and you will see the logs colored and formatted to be easily readable.

ADB also provides filters to have a clearer output. These are formatted by the `Tag:Priority` syntax. For example, if you want all logs from `MyApp`, just run `adb logcat MyApp:*`.

Now try to save all logs from Dalvik to a file using only `adb`.

Get a shell on the device

Having a shell on the device can prove pretty useful. ADB provides such a feature, even though this embedded shell is quite minimal. To access this shell, just type `adb shell`.

You can also run a command directly on the device thanks to `adb shell`. To do this, just append the command to run at the end. Now, try to get all the mounted filesystems on the device.

Push/Pull files to a device

In the same way, ADB allows you to retrieve files from the connected device and send them to it, using the `push` and `pull` commands.

Building a library

Add a native library to the build system

After this lab, you will be able to

- Add an external library to the Android build system
- Compile it dynamically
- Add a component to a build

Build a shared library

The Android team already has put the libusb source code, but for some reason isn't actually using it or made a Makefile for it. Since we need a more recent version anyway, we will need to edit the manifest so that repo grabs a newer version instead.

Create a new local manifest to download the libusb from the git repository `git://github.com/libusb/libusb.git`. You'll need to use the version 1.0.9, that is in git `refs/tags/1.0.9`. You'll also need to remove both the existing `libusb` and `libusb_aah` projects from the manifests.

Then, do a `repo sync -c`, and everything should be downloaded. Note that you may have to use `repo sync --force-sync external/libusb`.

For this library, all the needed `.c` files are located in the `libusb` folder and its subfolders. The headers are located in the same folders. You shouldn't modify the `libusb` source code.

Remember that you have make functions in the Android build system that allows you to list all the source files in a directory. You can also use the `filter-out` make macro to remove files that match a given pattern.

You will find one missing header that you will need to generate. Indeed, the `config.h` generated by autoconf is not generated at all, because the Android build system ignores other build systems. You will have to generate it by yourself, by running the `configure` script[4] that you can find in the `libusb` source code. The git repository doesn't have this `configure` script however. This script is also usually generated through the `autogen.sh` script that you'll find in the sources.

To be able to use this script, we will need to install the packages `libtool` and `automake`. Once installed, you can generate the configure script using `./autogen.sh`. It should also generate the needed `config.h` header.

Along the building process, you might need to define values in the source code. We prefer to avoid modifying directly the source code, since it will end up in merging problems in the future if we ever want to upgrade that component. `gcc` accepts on its command line the `-D` argument, that you can use either to define a value (`-DFOOBAR=42`) or a macro (`-D'FOO(bar)=(bar * 42)'`).

If successful, the build system should go through the build process and you should have a directory generated in the `out` directory.

[4]You can have some hints about the available options available by passing the `--help` argument to `configure`

Integrate the library into the Android image

As you can see, your library has been compiled during the build process, but if you boot the generated image or look inside the folder, you can see that the shared object is not present.

Modify the appropriate files so that the library gets included in your image.

Add a native application to the build

Learn how to begin with the Android build system

After this lab, you will be able to:

- Add an external binary to a system
- Express dependencies on other components of the build system

Add the binary to the compiled image

Copy the `bin` folder from the `$HOME/android-labs/native-app` directory and put it into your device folder.

Just as for `libusb`, you now need to make an `Android.mk` file giving all the details needed by the build system to compile. But unlike `libusb`, this binary is an executable and depends on another piece of software.

Make it compile and be integrated in the generated images. Once you have the images, boot the board, plug a missile launcher and test the application.

You can then control the missile launcher once you have started the `mlbin` application, using the following commands on the standard input, following by the duration in seconds:

- L - go left
- R - go right
- U - go up
- D - go down
- F - actionnate trigger

For example, use `L 1 U 1 F 5` will turn left for one second, up for one second and fire a missile.

System Customization

Learn how to build a customized Android

After this lab, you will be able to:

- Use the product configuration system
- Change the default wallpaper
- Add extra properties to the system
- Use the product overlays

Set up a new product

From now on, we're going to use a product of our own, so that we don't have to modify the pre-existing `beagleboneblack` product to suit our needs. In order to do so, define a new product named *training*. This product will have all the attributes of the `beagleboneblack` product for now, plus the extra packages we will add along the labs.

Remember that you need to use `make installclean` when switching from one product to another.

The system should compile and boot flawlessly on the BeagleBone, with all the corrections we made earlier.

Change the default wallpaper

First, set up an empty overlay in your product directory.

The default wallpaper is located in `frameworks/base/core/res/res/drawable-nodpi/`. Use the overlay mechanism to replace the wallpaper by a custom one without modifying the original source code. Be careful, the beagleboneblack product already have an overlay and is selecting a live wallpaper. To be able to use `default_wallpaper.jpg`, you have to select the `ImageWallpaper` component, have a look at `frameworks/base/core/res/res/values/config.xml`. Also, be sure to wipe your `/data` partition to fall back to the default wallpaper.

Develop a JNI library

Learn how JNI works and how to integrate it in Android

After this lab, you will be able to

- Develop bindings from Java to C
- Integrate these bindings into the build system

Write the bindings

The code should be pretty close to the one you wrote in the native application lab, so you will find a skeleton of the folder to integrate into your product definition in the `$HOME/android-labs/jni` directory. Copy the `frameworks` folder you will find there into your device folder.

You will mostly have to modify function prototypes from your previous application to make it work with JNI.

As a reminder, JNI requires the function prototype to be like: `JNIEXPORT <jni type> JNICALL Java_<package>_<class>_<function_name>(JNIEnv *env, jobject this)`. Beware that the function name can't have any underscore in its name for JNI to function properly.

The package we are going to use is `com.fe.android.backend`, and the class name is `USBBackend`. We are going to need the functions `fire`, `stop`, `initUSB`, `freeUSB`, `moveDown`, `moveUp`, `moveLeft` and `moveRight`.

Integrate it in the build system

Now you can integrate it into the build system, by writing an `Android.mk` as usual.

The library should be called `liblauncher_jni`.

Testing the bindings

We should now have a system with the files `/system/lib/liblauncher_jni.so`, containing the JNI bindings and `/system/lib/libusb.so`.

Test what you did so far by using the `TestJNI.apk` application you'll find in the `android-labs` directory.

Install it using adb, and run it.

Fix the Problems

The permission model of Android is heavily based on user permissions. By default, your USB device won't be accessible to applications. We didn't see this behaviour because we ran the mlbin program as root. To test if the program runs properly, test it as a random user on the system.

To do so, you can use the command `su`, and you can test with the user `radio` for example.

When you start your tests, you will find that `libusb` cannot open the usb devices because of restricted permissions. This can be fixed through `ueventd.rc` files. Add a device-specific `ueventd` configuration file to your build to make the files under `/dev/bus/usb/` world accessible.

You should now see a succesful message in the logs after running it.

Going further

You will find that the binary we developed in the previous lab and the bindings share a lot of common code. This is not very convenient to solve bugs impacting this area of the code, since we have to make the fix at two different places.

If you have some time left at the end of this lab, use it to make this common code part of a shared library used by both the bindings and the binary.

Develop a Framework Component

Learn to integrate it in Android a Framework component

After this lab, you will be able to

- Modify the Android framework
- Use JNI bindings

Write and integrate the component in the build system

Aside from the `jni` folder, you'll find in the `frameworks` folder a `java` folder that contains a Java Interface, `MissileBackendImpl`. In the same folder, write the `USBBackend` class implementing this interface that uses your bindings. You have an example of such a class in the `DummyBackend.java` file.

Now you can integrate it into the build system, so that it generates a .jar library that is in our product, with the proper dependencies expressed.

You can find documentation about how to integrate device-specific parts of the framework in the `device/sample/frameworks` folder.

Testing the bindings

We should now have a system with the files `/system/framework/com.fe.android.backend.jar`, containing the Java classes, `/system/lib/liblauncher_jni.so`, containing the JNI bindings and `/system/lib/libusb.so`.

Test what you did using the Main class present in the Java source code by directly invoking Dalvik through the `app_process` command. You will have to provide both the classpath and the class name to make it work and should look like `CLASSPATH=path/to/java.jar app_process /system/bin com.fe.android.Main`

Once you have a solution that works, you can ask your instructor to give you a URL where Bootlin's solution is available, and compare it with what you implemented.

Write an application with the SDK

Learn the basic concepts of the Android SDK and how to use them

After this lab, you will be able to:

- Write an Android application
- Integrate an application in the Android build system

Write the application

Go to the `$HOME/android-labs/app` directory.

You will find the basics for the `MissileControl` app. This app is incomplete, parts of some activities are missing. However the UI part is already done, so create the code to launch the activity with the given layout and the needed hooks defined in the layout description. These hooks implement the behavior of every button: move the launcher, fire, etc. Every button should be self-explanatory, except for the USB/Emulation switch, which switches between USB mode (which uses the USBBackend you developed) and Emulation mode (which uses the `DummyBackend` class for debugging).

The whole application now uses 3 layers to work, the application itself, which is a perfectly standard Android application, relying on Java → C bindings integrated in the Android framework, which in turn relies on libusb that we included in the system libraries.

We can't have a real USB missile launcher for participants, so the `DummyBackend` class is provided to test that your application and the changes to the framework are functional.

You can still switch back-ends with the switch button in the application.

Integrate the application

Copy the `MissileControl` folder into the folder `device/<your>/<device>/apps` that you will create.

Once again, write down the `Android.mk` file to build the application in the Android image, and set the dependencies on the JNI libs so that the app is compiled and runs properly.

When you test your application, you will find that it crashes because it doesn't find the `.jar` file containing the Java bindings to libusb, while you used them successfully with Dalvik. This is because Android's security model refuses to load untrusted jar files. However, you can make Android accept a jar file by adding an xml file similar to `AndroidManifest.xml` into the `/system/etc/permissions/` folder. There are several examples of such a file in this folder, so adapt one to our case. After rebuilding, it should work fine.

Once you have completed this lab, you can ask your instructor to give you a URL where Bootlin's solution is available, and compare it with what you implemented.

Going Further

We have used direct bindings and calls to use our library. However, we have seen that with Android, we tend to use services in that case, for security reasons, as well as to keep a consistent state across the system. If you have any time left, implement a new service that will be used by your application and replace the direct function calls.