How to Build your Own Android Based on AOSP

Michał Górny May 25, 2017

Mobile applications developing options allow developers to extend the functionality of stock operating systems provided by the major players in the mobile industry. However, sometimes it is not enough, especially if our requirements are larger than powers of the provided SDK to a specific platform. Thanks to Android openness, we can create the operating system that will be crafted according to our needs.

In the following tutorial, I will explain how the process of creating your customized version of Android system using AOSP (https://source.android.com) looks like in general. We will go through all the phases: starting from **setting up the environment**, through **building and uploading to a physical device**, and ending by **changing the system behavior and functionality**.

Android Open Source Project

Android Open Source Project is an *open software stack for a wide range of mobile devices and a corresponding open source project led by Google*. Simply put, it means that everyone can download Android sources and build their own customized version of an operating system.

We have to be aware that working with Android sources involves dealing with a huge amount of code. The Android Nougat 7.1.1 release has 518 repositories, 48 million lines of code in 27 languages. It's estimated that to develop such project in a year we would need 16075 developers, in one month - 192075 folks. Moreover, it would cost more than $2 billion dollars! But no worries—we don't want to reinvent the wheel. We simply want to use what the experienced developers have been creating for us for almost 10 years.

Building

In this tutorial, I will show how to build the customized system based on Android Nougat release on a Nexus 5X device.

Preparing a machine

First things first. You need to remember about few steps that need to be taken before you even start developing. There is a great documentation provided by Google that describes this process step by step. First of all, we need to choose a branch because some of the steps are determined by the version we plan to build. We will compile a version based on an *android-7.1.1_r24* tag. It corresponds to *N4F26T* release tag and supports the Nexus 5X device that will serve us to test a build.

Establishing a build environment

Currently, Android build supports two platforms - Linux and Mac OS. We need to install a bunch of required packages like JDK, Git, Python and more. Remember - this step is platform-specific so in order to go through, just follow an instruction provided here (https://source.android.com/source/initializing). I know it takes some time but fortunately, we need to do it only once.

Downloading the source

The Android sources are nothing different than a bunch of Git repositories hosted by Google. To operate on all of them Google provides a *Repo* tool that makes it easier to work with them in the context of Android platform. After the installation (https://source.android.com/source/downloading), we initialize the working directory. We chose a specific tag so to download them we need to call

```
repo init -u https://android.googlesource.com/platform/manifest -b android-7.1.1_r24
```

d then `repo sync`. Now we have a lot of time until almost 50 GB of sources download so a perfect moment to take a nice coffee break.

Preparing for a build

As sources are already downloaded, we can prepare the environment to start a build. The sources included in AOSP are not enough to build the whole Android. It requires additional libraries related to hardware to run on a specific device. For Nexus 5X these binaries can be downloaded from here (https://developers.google.com/android/drivers#bullheadn4f26t). Each binary is a self-extracting script in a compressed archive. They have to be extracted and run in the root of source tree. This installs all files in a vendor directory.

Setup environment

Android build system provides a lot of scripts and tools to make the build process easier and more convenient for a developer. To initialize directory we run a `. build/envsetup.sh` from root of source tree. Next, we need to choose a target for our build. We can select one of three variants: *user*, *userdebug* and *eng*. Each configuration is suited to various purposes. User build has limited access and is dedicated to production builds, *userdebug* and *eng* are for development purposes. They differ as far as an access to development and debugging tools are concerned. To set up the exact configuration for Nexus 5X device, run `lunch aosp_bullhead-eng` command.

Starting to build a code

Now comes the long-awaited moment of starting the build process. Android build system uses *make*. It allows building in parallel tasks when *-jN* argument is passed. *N* is a number of threads being used for the build. The command `make -j4` starts a build process and now we have to be patient because depending on the machine we are building on, it can take even a few hours. You can track all the console output during the build and in the end, you'll see something similar to

```
### make completed successfully (02:27:48 (hh:mm:ss)) ####
```

I achieved this result on a gaming laptop with Intel® Core™ i7-4720HQ CPU 2.60GHz x 8 and 32 GB RAM of memory on board. The same process on Macbook Pro with Intel® Core™ i5-4288U CPU 2.60GHz × 4 and 16 GB RAM takes 4 hours and 47 minutes. We can speed up our rebuild when we run build with *ccache* compilation tool. This works very well when we want to rebuild all sources again from the scratch. The duration of my rebuild was two times shorter and took 48 minutes on a gaming machine and few minutes over 2 hours on Macbook.

Polidea

Uploading a build to a device

A build is ready to be uploaded to a device. We can check and verify whether the Android starts and if we had produced the system we expected. Firstly, we need to make sure that we have *adb* and *fastboot* commands available. If you are an Android developer, you have most likely already installed them with the other platform tools delivered in Android SDK. Otherwise, you can build them with regular build system. Run `make adb fastboot` and that's it.

Booting into fastboot mode

This is a bootloader mode that allows flashing a device. There are two ways to run into such a state. Firstly, you can use the command `adb reboot bootloader`. Secondly, you can use the special key combination for Nexus 5X. Press and hold *Volume Down* then press and hold the *Power* button.

Unlocking the bootloader

Custom build can be uploaded only if a bootloader allows it. As a bootloader is locked by default, it has to be unlocked. Important! You have to be aware that it causes the removal of all user data. To unlock bootloader on Nexus 5X, go to the *Settings* app and enable developer options. This can be done when you tap seven times on *Build number* in *About phone* category. You should see a message confirming that fact and *Developer options* category added in an app. Open it and enable *OEM unlocking and USB debugging*. Restart a device into fastboot mode as it is described in the previous paragraph and run `fastboot oem unlock`. The bootloader can be re-locked using `fastboot oem lock` command.

Flashing

Now it's time to flash an entire Android system. This means that all produced images write to a proper partitions on a device. Run `fastboot flashall -w` command. After a successful process, the device will restart.

System customization

Finally we have our own version of Android. Now it's the time to implement some custom changes to a system.

Importing to IDE

What happens when we want to work with sources in IDE? Unfortunately, there is no officially designed IDE to work with Android Internals that fully integrates with a build system. But don't worry! Engineers made it possible to import sources very easily. In order to do that, we need to build *idegen* and run script by call following

```
make idegen && development/tools/idegen/idegen.sh
```

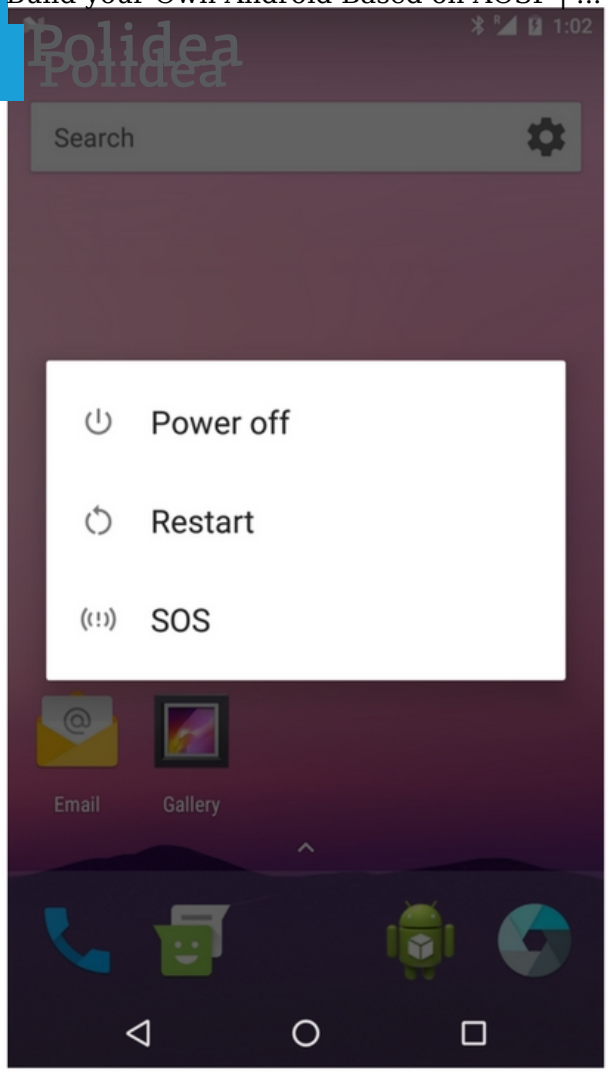
from the source root. This produces a few files including *android.ipr* that allows us to import sources to *Android Studio* or *IntelliJ*. Be patient once you launch IDE first time because sources need to be indexed and it's a quite long operation. When it finishes, we can easily browse, write and refactor code. To build code we use a terminal at the same time.

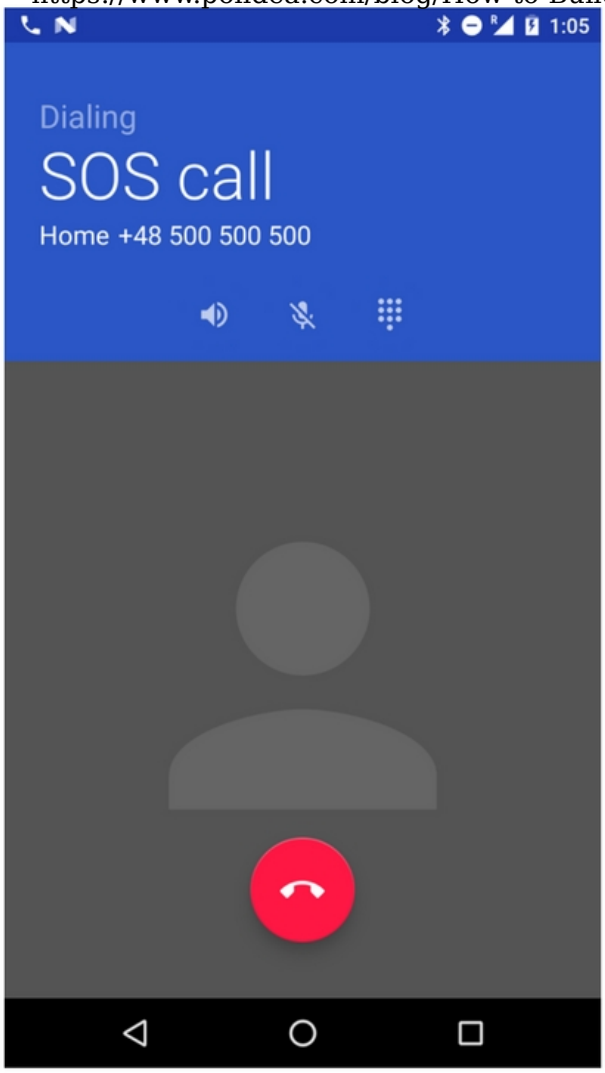
Developing

Now it's a time to invent a change that will single out our version from the stock Android. Let's imagine that we want to implement a smartphone for seniors. Usually, phones used by the elderly, have a feature that allows calling for help by pressing hardware button. We don't have such button in our Nexus 5X device, but we can implement this in software. This way, we will allow initializing a call to a predefined number in emergency situations.

SOS button feature

Emergency situations require quick actions. Making a phone call should be easy and quick. The change that we implement allows making a phone call directly from the menu that appears after long pressing the power button. Once user clicks SOS position, it will start to call a predefined number.





This feature requires a change to an Android Framework. Framework code is located under *frameworks/base/* path. So let's code.

SOSCallManager

First, we need to add a manager class that will handle triggering an action to start a call. This class sends a simple Intent including the uri of a phone number that is stored in the global settings of the device.

```
public class SOSCallManager {
    ...

    private void sendCallIntent() {
        Intent intent = new Intent(Intent.ACTION_CALL);
        intent.setData(getPhoneUri());
        intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        mContext.startActivity(intent);
    }

    private Uri getPhoneUri() {
        return Uri.fromParts("tel", getNumber(), null);
    }

    ...

    private String getNumber() {
        return Settings.Global.getString
                mContext.getContentResolver(), Settings.Global.SOS_CALL_NUMBER);
    }
```

Global actions

Long pressing the power button opens a menu with the *global actions*. Its content is based on a configuration in framework resources or specific overlay for a device. The config file is located in *frameworks/base/core/res/res/values/config.xml*. We add sos *item* to a resource named *config_globalActionsList*. *GlobalActions* class parses a config file and adds item if defined.

```
     Show the global actions dialog (creating if necessary)
     @param keyguardShowing True if keyguard is showing
 */
public void showDialog(boolean keyguardShowing, boolean isDeviceProvisioned) {
...

/**
 * Create the global actions dialog.
 * @return A new dialog.
 */
private GlobalActionsDialog createDialog() {

...

} else if (GLOBAL_ACTION_KEY_SOS_CALL.equals(actionKey)) {
    if (!TextUtils.isEmpty(Settings.Global.getString(
            mContext.getContentResolver(), Settings.Global.SOS_CALL_NUMBER))) {
        mItems.add(getSOSAction());
    }
}
```

An item in a power button dialog is an implementation of *Action* class. We can return an instance of *SinglePressAction* class with an overridden *onPress()* method that initializes a SOS call when users tap on it.

```
private Action getSOSAction() {
    return new SinglePressAction(com.android.internal.R.drawable.emergency_icon,
            R.string.global_action_sos_button) {
        @Override
        public void onPress() {
            mSOSCallManager.performSOSCall();
        }

        @Override
        public boolean showDuringKeyguard() {
            return true;
        }

        @Override
        public boolean showBeforeProvisioning() {
            return false;
        }
    };
}
```

The entire change of this feature in a framework can be found on Github (https://github.com/michalgorny/platform_frameworks_base/commit

723f8880524f4709cb310e556387bc61da7e122).

Polidea

Rebuilding components

Changes in the code are ready. It’s a time to recompile sources and verify the result on a device. We don’t need to follow the same procedure with compiling the entire system and flashing a device using *fastboot*. Instead, we can rebuild only particular components and replace them on a device. In order to do that call `make -j4 framework services`. If the platform is completely built, the names of the components passed are optional because the build system rebuilds only the components that have been changed. Now we have to upload what we built to a device. Before pushing files we put the *system* partition in a writable mode. By default, it is only readable. For this, we use `adb remount` command. Next call `adb sync system` that causes the files in a working directory to be synchronized with those on a device. In the end we restart a phone to load an uploaded version of rebuilt components during boot. We can restart a phone from power menu or using `adb reboot` command.

Changing a number

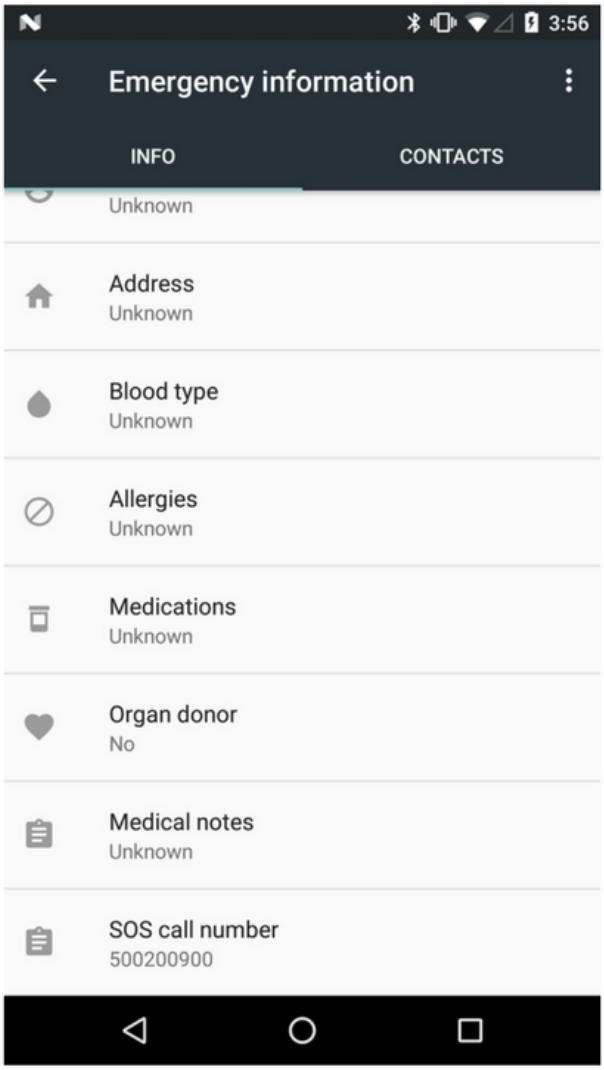
A number to perform SOS call is stored in global system preferences. Using *adb* we can simply change the value from a terminal by typing

```
adb shell settings put global sos_call_number <number>
```

I have to admit that it’s not the most convenient way to do so. For this purpose, we would add a possibility to edit in *Settings* application.

Emergency information

Android Nougat introduced *Emergency information*. This is a simple precaution that could prove very useful in emergency situations. The medical data can be filled out in *Users* section in *Settings* app. It seems to be a good place to add the next position with sos call number. The application that collects and provides medical data is stored in *packages/apps/EmergencyInfo/*. We add a few changes that will display another field with SOS call number. The user will be able to edit a system preference inside this application. The sources of this change are available on Github (https://github.com/michalgorny/platform_packages__apps_emergencyinfo/commit/812a49ebb9c734cff44216923a78cd7e38521b69).



Conclusion

Projects centered around the development of applications are a significant part of the mobile development world. Building your own version of Android system is not that popular and needs special cases to start considering customization. I strongly believe that taking part in such projects is a great opportunity to learn and it definitely contributes to Android developer's progress. The challenges you encounter are unique. As for me, I had the great chance to customize Android sources for a mobile payment device in project Albert (https://www.polidea.com/our-work/A_Step_Further_in_Payment_Solution_Revolution) - along with my team. It was a lot of fun! We're looking forward to the next Android Internals challenges!

Want to discuss your project idea? Contact us here! (/project)

Polidea

Michał Górny
Software Engineer

(https://github.com/michalgorny)

(https://github.com/Polidea)

(https://twitter.com/polidea)

Polidea

(https://www.facebook.com/Polidea.Software)

(https://www.linkedin.com/company/polidea)

(https://instagram.com/polidea/)

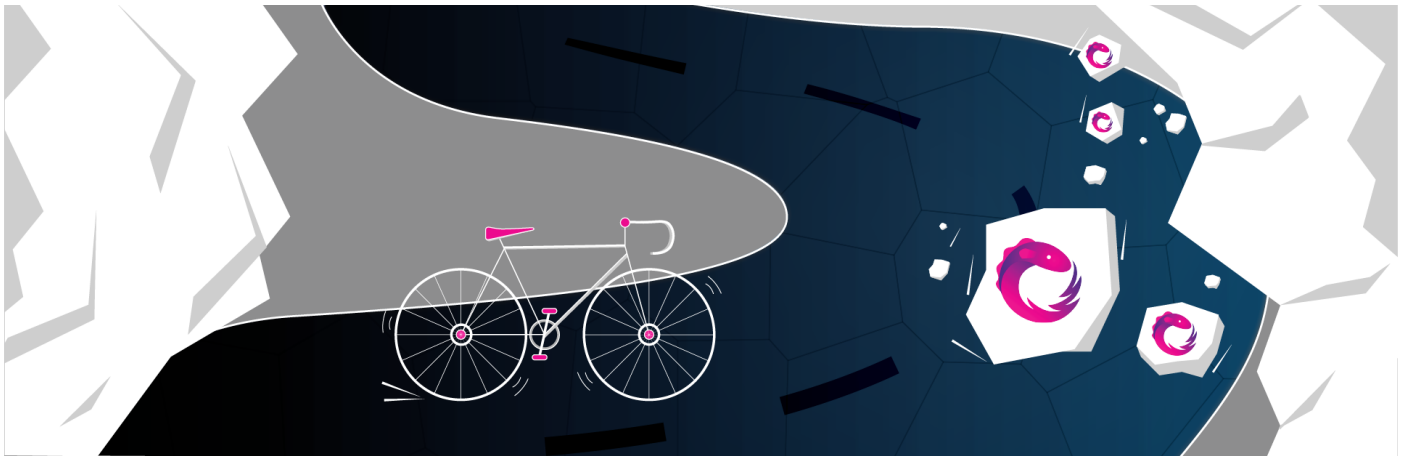
Polidea

POLIDEA'S NEWS

Subscribe to our monthly newsletter. Expect us to deliver sharp insights from the mobile app business world, unveil our latest released projects and share our events calendar, straight to your email box. Let’s keep in touch!

Enter your email Subscribe →

See also



RxSwift, polidea, development, guide, programming, Reactive Aug 23, 2017

8 Mistakes to Avoid while Using RxSwift—Part 1

First part of the series of articles addressing 8 mistakes to avoid while using RxSwift. Based on everyday practical use of RxSwift, this developer's guide will help you enjoy the benefits of reactive programming.

by Krzysztof Siejkowski read more →

(/blog/8-Mistakes-to-Avoid-while-Using-RxSwiftPart-1)