

# What is Android?

Android is an open-source operating system developed by Google. It is based on a modified version of the **Linux Kernel\*** and other open-source software and is designed primarily for touchscreen mobile devices.

\*The Linux Kernel is an open-source **monolithic\*\*** Unix-like computer operating system kernel.

\*\*A **monolithic kernel** is an operating system architecture where the entire operating system is working in **kernel space**. The monolithic model differs from other operating system architectures in that it alone defines a high-level virtual interface over the computer hardware.

## What is the Android Source Code?

Android is an open-source software stack created for a wide array of devices with different form factors. The primary purpose of Android is to create an open software platform available for carriers, OEMs, and developers to make their innovative ideas a reality. The result is a full production quality consumer product that improves the mobile experience for users.

## Android Rom Contents

Basically, an Android ROM contains the following main things:

- Kernel
- Bootloader
- Recovery
- Radio
- Framework
- Apps
- Core
- Android Runtime etc

## Terminologies

- **Kernel** – A kernel is a critical component of an operating system. It can be seen as a sort of bridge between the applications and the actual hardware of a device. Android devices use the Linux Kernel, but it's not the exact same kernel other Linux based operating systems use. There's a lot of Android specific code built in. OEMs have to contribute as well because they need to develop hardware drivers for the kernel version.
- **Bootloader** – The bootloader is a piece of code that is executed before any operating system starts to run. Bootloaders basically package the instructions to boot operating system kernel and most of them have their own debugging and modification environment. Think of the bootloader as a security checkpoint for all those partitions.

- **Recovery** – Recovery is defined in simple terms as a source of backup. Whenever your phone’s firmware is corrupted, the recovery does the job in helping you to restore your faulty or buggy firmware into working condition.
- **Radio** – The lowest part of the software layer is Radio. This is the very first thing that runs, just after the bootloader. It controls all wireless communication like GSM antenna, GPS etc.

# Android ROM Development

## Setting Up Linux Environment

Pre-requirements for Android ROM Development:

- Linux Operating System (Ubuntu Recommended)
- Java Development Kit (JDK)
- Some Other Tools

Ubuntu Installation:

- Installing as Primary OS
- Installing a Virtual Operating System (Using Virtual Machine - Make sure you have JDK and VT-x in processor settings from BIOS installed and enabled before installing the Virtual Box and booting your Linux OS)
- Dual Booting using different partitions (Prior Experience and Knowledge Required)

## Setting Up Build Environment for Compiling ROMs

Basically what we’re doing is downloading the *raw* Android source code (AOSP) and modifying it to our liking. Because there are so many sources to choose from for different devices, this guide will simply reference the **master source** known as AOSP (Android Open Source Project).

Now the thing about the AOSP is that the pure source code *does not include* device-specific hardware proprieties. In layman’s terms, hardware like your camera and GPU will not work “out of the box” when developing with the AOSP. In fact, your device will not even boot without these hardware binaries.

If you’re developing for a Google-branded phone (Pixel, Nexus, etc) you can find the hardware binaries [directly from Google](#), and this guide will walk you through obtaining and building them into your ROM. However, if you’re developing a ROM for a brand-name phone (Sony, Samsung, etc)... well, bless your heart, because you’re in for a ride.

*Some* manufacturers have their own open-source projects or release development tools for would-be developers, whereas other manufacturers keep a tight lid on their proprietary codes. Here’s a brief list of open-source projects from the more popular manufacturers:

[Samsung Open Source Release Center](#)

[Sony Developer World](#)

[Lenovo Support](#)

[Huawei Open Source Release Center](#)

[Motorola Developers](#)

With that out of the way, let's continue under the assumption we are building a ROM for the most basic, vanilla Android experience, for a Google Pixel device. With this knowledge under your belt, you'll be able to branch out on your own and start developing customized versions of specific manufacturer's ROMs.

## Requirements for this Guide:

### [Android Open Source Project](#)

Pixel XL phone **or** an Android emulator for Linux

64-bit Linux Operating System – [Ubuntu](#) or [Linux Mint](#) is the most newbie-friendly distros, whereas [BBQLinux](#) was developed specifically with Android developers in mind.

Python

A beefy computer (compiling code takes a lot of memory and space!)

## Setting Up Your Build Environment

Let's begin by setting up the Android emulator on your Linux machine. Whether or not you have a Google Pixel XL device, it's always safest to try your new ROM on an Android emulator *before* flashing it to your device. My personal favorite is Genymotion, so I'll walk you through installing that particular emulator. However, you can also check out this guide "[Best Android Emulators for Windows in 2017](#)", as most of them also have Linux compatibility. Head over to the [Genymotion website](#), register an account, verify it through email, and download the executable to your Linux desktop. Now open a Linux terminal, and type:

```
Chmod +x genymotion-xxxxx.bin (replace XXXX with the version number in the filename)
./genymotion-xxxxxx.bin
```

Press **Y** to create the Genymotion directory. Now type in the terminal:

```
cd genymotion && ./genymotion
```

Now it will ask you to begin the installation process, so just keep clicking Next until you get to the Add Virtual Devices window. Select "Pixel XL" under Device Model option, and then complete the installation. You can test the virtual device out if you want, it will basically be like having a Pixel XL phone on your desktop.

Let's now set up Python:

```
$ apt-get install python
```

Now we need to setup the Java Development Kit on your Linux machine. Open the Linux terminal and type the following commands:

```
$ sudo apt-get update
```

```
$ sudo apt-get install OpenJDK-8-JDK
```

Now you will need to configure the Linux system to allow USB device access. Run the following code in the Linux terminal:

```
$ wget -S -O - http://source.android.com/source/51-android.txt | sed "s/<username>/$USER/" | sudo tee  
>/dev/null /etc/udev/rules.d/51-android.rules; sudo udevadm control --reload-rules
```

This will download the required 51-android.txt file that allows the aforementioned USB device access. Open the .txt file and modify it to include your Linux username, then place the .txt file in the following location:

```
/etc/udev/rules.d/51-android.rules
```

(as the **root user**). Now plug your device into your computer via USB for the new rules to automatically take effect.

## Downloading the Android Source

The AOSP is hosted on Git, so we're going to use a tool called Repo to communicate with Git. First, we need to set up a /bin folder in your Home directory. Type the following commands into the Linux terminal:

```
$ mkdir ~/bin  
$ PATH=~/bin:$PATH
```

Now we will download the Repo tool, so type into the Linux terminal:

```
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo  
$ chmod a+x ~/bin/repo
```

After Repo is installed, we must now create an empty directory to hold your work files. So type this into the Linux terminal:

```
$ mkdir WORKING_DIRECTORY  
$ cd WORKING_DIRECTORY
```

Now we'll configure Git with your name and email address – **use a Gmail address that you check regularly**, otherwise you will not be able to use the Gerrit code-review tool.

```
$ git config --global user.name "Your Name"  
$ git config --global user.email you@gmail.com
```

Now we'll tell Repo to pull the latest master manifest of AOSP from Git:

```
$ repo init -u https://android.googlesource.com/platform/manifest
```

If done successfully, you'll receive a message that Repo has been initialized in your working directory. You'll also find a ".repo" directory inside the client directory. So now we'll download the Android source tree with:

```
$ repo sync
```

## Building the Android Source

This is where the hardware binaries mentioned at the beginning of this guide come into play. Let's head over to the [AOSP drivers](#) page and download the Pixel XL binaries for Android 7.1.0 (NDE63P). You want to download both the vendor image and the hardware components. These come as compressed archives, so extract them to your desktop and run the self-extracting script from the root folder. Choose to install the binaries to the root of the `WORKING_DIRECTORY` we created earlier.

Now type into your Linux terminal:

```
$ make clobber  
$ source build/envsetup.sh
```

Now we'll choose the target to build, so type:

```
$ lunch aosp_marlin-userdebug  
$ setpaths  
$ make -j4
```

There, we have now "built" an Android ROM from source. So let's test it in the emulator, by typing into the terminal:

```
$ emulator
```

So play around in the emulator a bit. As you can see, a purely vanilla Android experience is quite minimal, and this is why manufacturers customize the AOSP to their needs. So you *could* flash this ROM we just built to your device if you wanted, but without adding any enhancements, a purely vanilla Android experience would be a very boring thing indeed. So what manufacturers will typically do with the AOSP is fork it, add their own proprietary binaries, customize the UI, add a boot logo, etc. Manufacturer's basically just paint over the stock Android ROM, and so that will be your next goal as well.

## Editing Your ROM

This guide will show you how to add flavor to the ROM by adding a custom boot animation and system themes. Get your gloves on, because things will get messy.

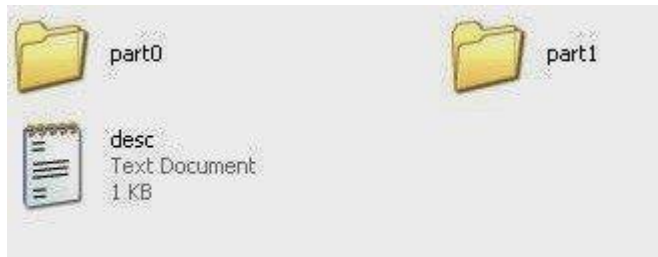
### How to Create a Boot Animation Tools Required:

A photo editor that saves in .PNG format (like GIMP or Photoshop)

Assuming you want to create your *very own custom* boot animation, remember that the image resolution should meet your device's resolution. So your images should be 720 x 1280 if you have a phone with that resolution, for example. Save them as 32-bit .PNG files. You need to save each frame of your images in corresponding steps:

00001.png  
00002.png  
00003.png

There is no limit to the number of frames you can have in your boot animation, but the best practice is to use a lesser amount of frames on a loop. CyanogenMod for example uses a looping animation of only a few frames, rather than one long continuous animation.



You need to create two folders on your desktop – name them **part0** and **part1**. If you’re creating something really fancy, you can create more **part#** folders. This is because the **part0** folder will be the animation’s “intro”, and **part1** will be the looping frames, and **part2** can be the outro, if you decide to do this. So think of it this way: your image fades *onto* the screen – these frames are saved in **part0**. Now your image spins around a few times – these frames are saved in **part1**. Now your image fades *out* of the screen – these frames are saved in **part2**. Makes sense, right?

Now what controls how your animation is played is a text file called “*desc.txt*”. The desc.txt is broken down like this:

```
720 1280 30
c 1 15 part0
c 0 0 part1
c 1 30 part2
```

A screenshot of a text editor window. The menu bar at the top includes 'File', 'Edit', 'Format', 'View', and 'Help'. The text area contains the following lines:

```
720 1280 30
c 1 15 part0
c 0 0 part1
c 1 30 part2
,
```

Here is what all of that means:

720 1280 30 = Resolution (width x height) + play at 30 frames-per-second. You can also do 60 or 10 FPS.

C means the animation will continue to play fully and *not abort*, even if the OS is loaded. You can optionally use P instead of C, which will abort the animation and go straight to the OS when its loaded, but this makes for an ugly boot-animation that never plays fully – unless you create an infinitely-looping animation.

1 is the loop count, meaning how many times the frames inside the part# folder will be played before moving to the next folder.

15 is how long each frame will “pause” before going to the next frame. 15 is 0.5 seconds, because 15 is half of 30.

Part# is obviously the folder being played.

Basically you want your *desc.txt* file to read like this:

```
[type] [loop count] [pause] [path]
```

Now, create a new .zip archive and name it bootanimation.zip, then drag your desc.txt and part# folders into this archive. If you followed Part 1 of this guide, you will have a WORKING\_DIRECTORY on your Linux machine. You need to copy your bootanimation.zip into the following folder:

```
out/target/product/<product>/system/media
```

Now the next time you build your ROM, your bootanimation.zip will be the default boot animation for your ROM.

## Set the Default Wallpaper

Navigate to this folder:

```
/frameworks/base/core/res/res/your-resolution
```

In there you will find a file “default\_wallpaper.jpg” – you can replace this with an image of the same resolution and filename, and when you build your ROM, it will be the default wallpaper.

## Add ROM Info to Settings > About

Navigate to ./packages/apps/Settings/res/xml/ in your build tree folder.

Now open device\_info\_settings.xml with GEdit and edit this information to your liking:

```
<!-- ROM name -->
<Preference android:key="rom_name"
android:enabled="false"
android:shouldDisableView="false"
android:title="ROM name"
android:summary="Appuals ROM Build Guide ROM"/>
<!-- ROM build version -->
<Preference android:key="rom_number"
android:enabled="false"
android:shouldDisableView="false"
android:title="ROM build number"
android:summary="7.0.1"/>
```

## Customize the Messenger App

Modifying a pre-existing app is much easier than replacing the core apps, so let’s perform a simple tweak on the default messaging app.

Navigate to ./packages/apps/Messaging/ and open BugleApplication.java with GEdit. We’re going to make a simple toast function, that is, the app will display a pop-up message when the app is opened. So inside the BugleApplication.java file, look for this bit of code:

```
import android.widget.Toast;
```

Look for the `onCreate()` function and just before the `Trace.endSection()`, add these lines:

```
Toast myToast = Toast.makeText(getApplicationContext(), "Appuals Rocks!",  
Toast.LENGTH_LONG);myToast.show();
```

Save the file and now the messenger app will display that toast message whenever the app is opened on your ROM!

## Edit the Build.Prop File

Navigate to the `/build/tools` folder in the Android source directory and edit the file `buildinfo.sh` with a text editor. It basically contains what will be output to the ROM's `build.prop` file when the ROM is compiled, for example you will see in `buildinfo.sh` things like:

```
echo "ro.build.date.utc=$BUILD_UTC_DATE"  
echo "ro.build.type=$TARGET_BUILD_TYPE"  
echo "ro.build.user=$USER"  
echo "ro.build.host=`hostname`"  
echo "ro.build.tags=$BUILD_VERSION_TAGS"  
echo "ro.product.model=$PRODUCT_MODEL"  
echo "ro.product.brand=$PRODUCT_BRAND"  
echo "ro.product.name=$PRODUCT_NAME"  
echo "ro.product.device=$TARGET_DEVICE"  
echo "ro.product.board=$TARGET_BOOTLOADER_BOARD_NAME"  
echo "ro.product.cpu.abi=$TARGET_CPU_ABI"
```

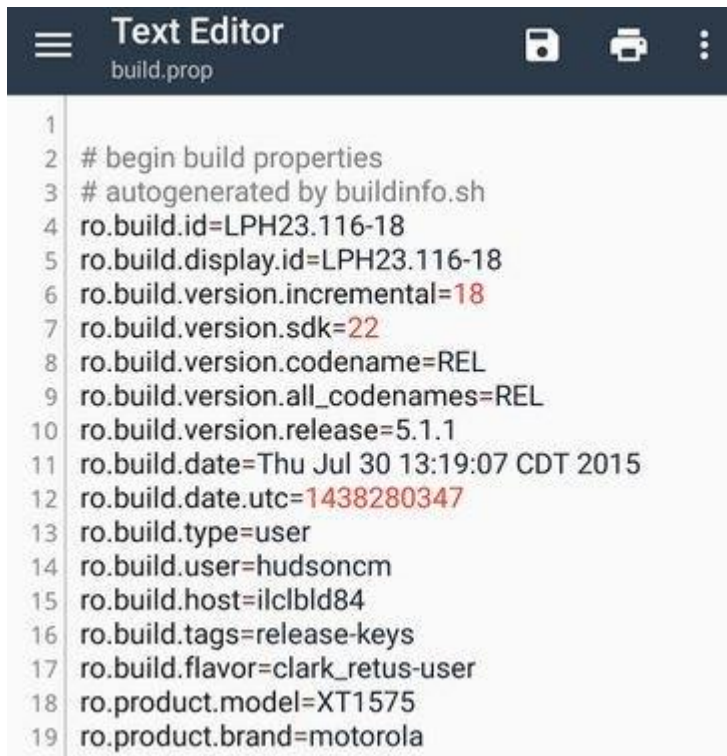
Anything you change in here will be copied over to `build.prop` when you compile the ROM.

## Essential Tweaks

Android devices use a universal `build.prop` that controls runtime flags when the Android system boots. Many of the properties are unique to each device, but there are a handful of `build.prop` tweaks that can be applied universally to virtually all Android phones. This guide will show you these tweaks, and how to edit the `build.prop` file.

**Warning:** *Tweaking your `build.prop` can brick your Android device. The tweaks I am sharing should work without problem on any Android device, but you should **always** have a backup of your data and stock ROM just incase something happens.*





```
1
2 # begin build properties
3 # autogenerated by buildinfo.sh
4 ro.build.id=LPH23.116-18
5 ro.build.display.id=LPH23.116-18
6 ro.build.version.incremental=18
7 ro.build.version.sdk=22
8 ro.build.version.codename=REL
9 ro.build.version.all_codenames=REL
10 ro.build.version.release=5.1.1
11 ro.build.date=Thu Jul 30 13:19:07 CDT 2015
12 ro.build.date.utc=1438280347
13 ro.build.type=user
14 ro.build.user=hudsoncm
15 ro.build.host=ilclbld84
16 ro.build.tags=release-keys
17 ro.build.flavor=clark_retus-user
18 ro.product.model=XT1575
19 ro.product.brand=motorola
```

## How to Edit Your Build.Prop

The fastest method is using a root explorer app (which requires a rooted phone) – ES File Explorer, FX, and Root Explorer are great apps. You will need to mount your /system as read/writable in your root explorer options. Navigate to your /system folder and find build.prop file, then open it with a text editor. Alternatively, you can download a build.prop tweaker through Google Play store. BuildProp Editor and ROM Toolbox are recommended apps.

The second method which does not require a rooted phone is through recovery mode. Not all manufacturers' stock recovery modes allow mounting of /system partition, so if this applies to you, search for how to install a custom recovery such as TWRP on your device.

If your stock recovery **does** allow /system mounting, you just need to connect your device to your PC via USB and open an ADB terminal, then pull your build.prop to your PC using this command:

```
adb pull /system/build.prop <save path, e.g. C:\MyBuildProp>
```

Now you can edit the file on your computer with a text editor, then push it back to your phone via ADB with:

```
adb push <file path> /system/build.prop
```

After you push the build.prop back to your device, you need to set its permissions. Continue to type the following into ADB terminal:

```
adb shell
chmod 644 /system/build.prop
adb reboot
```

# Essential Build.Prop Tweaks for All Android Devices

## More Efficient RAM Management

ro.HOME\_APP\_ADJ=1

## Improved audio/video recording quality

ro.media.enc.jpeg.quality=100  
ro.media.dec.jpeg.memcap=8000000  
ro.media.enc.hprof.vid.bps=8000000  
ro.media.capture.maxres=8m  
ro.media.panorama.defres=3264×1840  
ro.media.panorama.frameres=1280×720  
ro.camcorder.videoModes=true  
ro.media.enc.hprof.vid.fps=65

## Less video buffering on streaming services

media.stagefright.enable-player=true  
media.stagefright.enable-meta=true  
media.stagefright.enable-scan=true  
media.stagefright.enable-http=true  
media.stagefright.enable-rtsp=true  
media.stagefright.enable-record=false

## Faster Internet Speeds

net.tcp.bufferSize.default=4096,87380,256960, 4096, 16384,256960  
net.tcp.bufferSize.wifi=4096,87380,256960,409 6,163 84,256960  
net.tcp.bufferSize.umts=4096,8 7380,256960,4096,163 84,256960  
net.tcp.bufferSize.gprs=4096,8 7380,256960,4096,163 84,256960  
net.tcp.bufferSize.edge=4096,8 7380,256960,4096,163 84,256960

## Reduced Battery Consumption

ro.mot.eri.losalert.delay=1000 (could brake tethering.)  
ro.ril.power\_collapse=1  
pm.sleep\_mode=1  
wifi.suplicant\_scan\_interval=180  
ro.mot.eri.losalert.delay=1000

## 3G Network Tweaks

```
ro.ril.hep=0
ro.ril.hsxpa=2
ro.ril.gprsclass=12
ro.ril.enable.dtm=1
ro.ril.hsdpa.category=8
ro.ril.enable.a53=1
ro.ril.enable.3g.prefix=1
ro.ril.htcmaskw1.bitmask=4294967295
ro.ril.htcmaskw1=14449
ro.ril.hsupa.category=6
```

## Faster Phone Reboot

```
ro.config.hw_quickpoweron=true
```

## Change LCD Density

```
ro.sf.lcd.density=xxx <replace xxx with a numeric value>
```

## Enable VoLTE (Voice over LTE / HD call quality)

```
#ifdef VENDOR_EDIT
persist.dbg.ims_volte_enable=1
persist.dbg.volte_avail_ovr=1
persist.dbg.vt_avail_ovr=0
persist.data.iwlan.enable=true
persist.dbg.wfc_avail_ovr=0
#endif
```

## Rotating Launcher and Lock Screen

```
log.tag.launcher_force_rotate=VERBOSE
lockscreen.rot_override=true
```

## Audio Enhancements (Music and audio resampling quality, etc)

```
persist.audio.fluence.mode=endfire
persist.audio.vr.enable=true
persist.audio.handset.mic=digital
af.resampler.quality=255
mpq.audio.decode=true
```

# Building Your Own Kernel

Building your own kernel can be a rewarding experience as it will give you a greater degree of control over your Android device, from the CPU, RAM, GPU to even the battery.

This is a very hands-on process that involves a lot of compiling and console commands, but if you're familiar with Linux (or good at following directions) it should be no problem.

Please note this guide is for non-Mediatek devices. Appual's has a kernel compiling guide specific ally for Mediatek-based Android devices here: [How to Build Mediatek Android Kernel from Source](#)  
Other Appual's articles of interest include:

[How to Manually Theme Android System UI](#)

If you're building a **custom** kernel, you will just need to clone the kernel from Git with commands provided below. But if you're compiling a *stock* kernel, you need to know where to get the original kernel from source (for all sorts of reasons).

## Original Kernel Sources for Various Brands:

[Google](#)

[LG](#)

[Samsung](#)

[HTC](#)

[OnePlus](#)

[Motorola](#)

[Sony](#)

For downloading the kernel, either use a git clone *or* download the tarball file and extract it.  
Here is the git command:

```
git clone -b <branch_to_checkout> <url> <desired_folder_name>
```

—OR—

```
tar -xvf <filename>
```

So as an example, this would be the command to grab the latest Nexus 6P Nougat 3.10 kernel from Google:

```
git clone -b android-msm-angler-3.10-nougat-mr2  
https://android.googlesource.com/kernel/msm/ angler
```

This should clone the kernel / msm repo into an angler folder, and automatically checkout the android-msm-angler-3.10-nougat-mr2.

Now because most Android devices are ARM based, we'll need to use a compiler that will target ARM devices – this means a host/native compiler won't work, *unless* you're compiling on another ARM device. You have a few options here. You can either compile one yourself *if you know how*, using something like Crosstool-NG. Alternatively, you can download a prebuilt compiler – such as the one Google provides for [Arm 32-bit](#) and [Arm64](#).

Before downloading a prebuilt compiler, you need to know the exact architecture of your device, so use an app like CPU-Z to determine it.

One other popular toolchain would be [UberTC](#) – but for any kernels higher than 4.9, you'll need to patch them, and compiling with Google's toolchain first is best practice. In any case, once you've decided on the toolchain, you need to clone it.

```
git clone <url>
```

Now point the Makefile to your compiler, **running it from within the toolchain folder.**

```
export CROSS_COMPILE=$(pwd)/bin/<toolchain_prefix>-
```

Example:

```
export CROSS_COMPILE=$(pwd)/bin/aarch64-linux-android-
```

Now tell the Makefile your device architecture.

```
export ARCH=<arch> && export SUBARCH=<arch>
```

Example:

```
export ARCH=arm64 && export SUBARCH=arm64
```

Locate your proper defconfig by navigating to the arch/<arch>/configs folder within the kernel source (e.g. arch/arm64/configs).

Next locate the developer's proper config file for the kernel you are building. It should typically be in the form of <codename>\_defconfig or <kernel\_name>\_defconfig. The defconfig will instruct the compiler what options to include in the kernel.

Generic Qualcomm configs may also be found, these will usually be something like (msm-perf\_defconfig, msmcortex-perf\_defconfig).

## Building the Kernel

Code:

```
make clean  
make mrproper
```

```
make <defconfig_name>
make -j$(nproc -all)
```

When those commands are successful, you should have: an Image, Image-dtb, Image.gz, or Image.gz-dtb file at the end.

If those commands failed, you might need to specify the output directory when making a new CAF based kernel, like this:

```
mkdir -p out
make O=out clean
make O=out mrproper
make O=out <defconfig_name>
make O=out -j$(nproc -all)
```

If it still doesn't want to work, something is broken – check your headers or bring it up with the kernel developers. If the kernel was successfully compiled, you now need to flash it. There are two different ways of doing this – you can unpack and repack the bootimage using either Android Image Kitchen or AnyKernel2. There may also be some nuances based on specific devices – you'll need to ask your device developers for assistance if this is the case.

## Flashing the Kernel in Android Image Kitchen

Download [Android Image Kitchen](#)

Extract your Android device's boot image from the latest available image (whether stock or custom ROM).

Now unpack the image using this code:

```
unpackimg.sh <image_name>.img
```

Next locate the Image file, and replace it with your compiled kernel image – **rename it to what was in the boot image.**

Now run this code to repack the image:

```
repackimg.sh
```

Now you can flash the new boot image using fastboot, TWRP, etc.

## Flashing the Kernel in AnyKernel2

Download the latest [AnyKernel2](#)

Apply [this patch](#) to flush out all the demo files.

```
wget
```

```
https://github.com/nathanchance/AnyKernel2/commit/addb6ea860aab14f0ef684f6956d17418f95f29a.diff
```

```
patch -p1 < addb6ea860aab14f0ef684f6956d17418f95f29a.diff
rm addb6ea860aab14f0ef684f6956d17418f95f29a.diff
```

Now place your kernel image in the root of the file, and open the anykernel.sh to modify these values:

- **string:** your kernel name
- **name#:** List all of your device's codenames (from the /system/build.prop: ro.product.device, ro.build.product)
- **block:** Your boot image's path in your fstab. The fstab can be opened from the root of your device and it will look something like this:  
<https://android.googlesource.com/dev...r/fstab.angler>

The first column is the value you want to set block to.

Now re-zip the kernel, and flash it in AnyKernel2:

```
zip -r9 kernel.zip * -x README.md kernel.zip
```

Be warned that many kernels from CAF include a Python script that will trigger – werror, which basically causes your build to throw errors at the tiniest of things. So for higher GCC versions (which include more warnings), you'll typically need to make changes in the Makefile:

```
diff --git a/Makefile b/Makefile
index 1aaa760f255f..bfccd5594630 100644
--- a/Makefile
+++ b/Makefile
@@ -326,7 +326,7 @@ include $(srctree)/scripts/Kbuild.include
AS                                = $(CROSS_COMPILE)as
LD                                = $(CROSS_COMPILE)ld
-REAL_CC                          = $(CROSS_COMPILE)gcc
+CC                               = $(CROSS_COMPILE)gcc
CPP                              = $(CC) -E
AR                               = $(CROSS_COMPILE)ar
NM                               = $(CROSS_COMPILE)nm
@@ -340,10 +340,6 @@ DEPMOD                          = /sbin/depmod
PERL                             = perl
CHECK                            = sparse
-# Use the wrapper for the compiler. This wrapper scans for new
-# warnings and causes the build to stop upon encountering them.
-CC                              = $(srctree)/scripts/gcc-wrapper.py
$(REAL_CC)
-
CHECKFLAGS := -D__linux__ -Dlinux -D__STDC__ -Dunix -D__unix__ \
-Wbitwise -Wno-return-void $(CF)
CFLAGS_MODULE =
```

Using a higher GCC toolchain (5.x, 6.x, 7.x or even 8.x) will require you to nuke the GCC wrapper script as above and use a unified GCC header file (pick the following if you have an include/linux/compiler-gcc#.h file):

3.4/3.10: <https://git.kernel.org/pub/scm/linux...9bb8868d562a8a>

3.18: <https://git.kernel.org/pub/scm/linux...9f67d656b1ec2f>

Even if you get a lot of warnings, they're not necessary to fix (typically).  
Your kernel is built and ready to go...

## Updating Your Kernel

This guide will involve cherry-picking and merging commits from the latest Linux-stable kernel with your Android kernel before you compile it.

Upstreaming your Android kernel to the latest Linux stable has a lot of positive benefits, such as being up-to-date with the latest security commits and bugfixes – we'll explain some of the pros and cons later in this guide.

### What is Linux-Stable kernel?

mainline:	<b>4.18-rc8</b>	2018-08-05	<a href="#">[tarball]</a>
stable:	<b>4.17.14</b>	2018-08-09	<a href="#">[tarball]</a>
longterm:	<b>4.14.62</b>	2018-08-09	<a href="#">[tarball]</a>
longterm:	<b>4.9.119</b>	2018-08-09	<a href="#">[tarball]</a>
longterm:	<b>4.4.147</b>	2018-08-09	<a href="#">[tarball]</a>
longterm:	<b>3.18.118 [EOL]</b>	2018-08-09	<a href="#">[tarball]</a>
longterm:	<b>3.16.57</b>	2018-06-16	<a href="#">[tarball]</a>

Linux-stable as the name implies is the stable arm of the Linux kernel. The other arm is known as "mainline", which is the *master branch*. All of the Linux kernel development happens in the mainline, and generally follows this process:

1. Linus Torvalds will take a bunch of patches from his maintainers for two weeks.
2. After this two weeks, he releases an rc1 (e.g. 4.14-rc1) kernel.
3. For each week for the next 6-8 weeks, he will release another RC (e.g. 4.14-rc2, 4.14-rc3, etc) kernel, which contains ONLY bug and regression fixes.
4. Once it is deemed stable, it will be released as a tarball for download on [org](#) (e.g. 4.14).

### What are LTS kernels?

Every year, Greg will pick one kernel and maintain it for either two years (LTS) or six years (extended LTS). These are designed to have products that need stability (like Android phones or other IOT devices). The process is the exact same as above, it just happens for



a longer time. There are currently six LTS kernels (which can always be viewed on [the kernel.org releases page](https://kernel.org/releases/page/)):

- [4.14 \(LTS\)](#), maintained by Greg Kroah-Hartman
- [4.9 \(LTS\)](#), maintained by Greg Kroah-Hartman
- [4.4 \(eLTS\)](#), maintained by Greg Kroah-Hartman
- [4.1\(LTS\)](#), maintained by Sasha Levin
- [3.16 \(LTS\)](#), maintained by Ben Hutchings
- [3.2 \(LTS\)](#), maintained by Ben Hutchings

## What are the benefits of upstreaming my Android kernel to Linux Stable?

When important vulnerabilities are disclosed/fixed, the stable kernels are the first ones to get them. Thus, your Android kernel will be a lot safer against attacks, security flaws, and just bugs in general.

*The Linux stable includes fixes for a lot of drivers my Android device doesn't use, isn't this mostly unnecessary?*

Yes and no, depending on how you define "mostly". The Linux kernel may include a lot of code that goes unused in the Android system, but that doesn't guarantee there will be no conflicts from those files when merging new versions! Understand that virtually *nobody* builds every single part of the kernel, not even the most common Linux distros like Ubuntu or Mint. This doesn't mean you shouldn't be taking these fixes because there **ARE** fixes for drivers you **DO** run. Take arm/arm64 and ext4 for example, which are the most common Android architecture and file system respectively. In 4.4, from 4.4.78 (version of the latest Oreo CAF tag) to 4.4.121 (latest upstream tag), these are the following numbers for the commits of those systems:

```
nathan@flashbox ~/kernels/linux-stable (master) $ git log --format=%h v4.4.78..v4.4.121 | wc -l2285 nathan@flashbox ~/kernels/linux-stable (master) $ git log --format=%h v4.4.78..v4.4.121 arch/arm | wc -l158 nathan@flashbox ~/kernels/linux-stable (master) $ git log --format=%h v4.4.78..v4.4.121 arch/arm64 | wc -l22 nathan@flashbox ~/kernels/linux-stable (master) $ git log --format=%h v4.4.78..v4.4.121 fs/ext4 | wc -l118
```

The most time-consuming part is the initial bring up; once you are all the way up to date, it takes no time at all to merge in a new release, which usually contains no more than

100 commits. The benefits that this brings (more stability and better security for your users) should necessitate this process though.

## How to Merge Linux Stable Kernel into an Android Kernel

First you need to figure out what kernel version your Android device is running.

As trivial as this seems, it is necessary to know where you need to begin. Run the following command in your kernel tree:

```
make kernelversion
```

It will return back the version you're on. The first two numbers will be used to figure out the branch you need (e.g. linux-4.4.y for any 4.4 kernel) and the last number will be used to determine what version you need to start with merging (e.g. if you are on 4.4.21, you'll merge 4.4.22 next).

## Grab the latest kernel source from kernel.org

[kernel.org](https://kernel.org) houses the latest kernel source in [the linux-stable repository](#). At the bottom of that page, there will be three fetch links. In my experience, Google's mirror tends to be the fastest but your results may vary. Run the following commands:

```
git remote add linux-stable  
https://kernel.googlesource.com/pub/scm/linux/kernel/git/stable/linux-  
stable.gitgit fetch linux-stable
```

## Decide if you want to merge the entire kernel or cherry-pick the commits

Next, you will need to choose if you want to merge the commits or cherry-pick. Here's the pros and cons of each and when you may want to do them.

**NOTE:** If your kernel source is in the form of a tarball, you will most likely need to cherry-pick, otherwise you will get thousands of file conflicts because git is populating the history based purely on upstream, not what the OEM or CAF has changed. Just skip to step 4.

## Cherry-picking:

Pros:

- Easier to resolve conflicts as you know exactly what conflict is causing an issue.
- Easier to rebase as each commit is on its own.
- Easier to bisect if running into issues

Cons:

- It takes longer as each commit has to be individually picked.
- Little more difficult to tell if commit is from upstream on first glance

## Merge

**Pros:**

- It's faster as you do not have to wait for all of the clean patches to merge.
- It's easier to see when a commit is from upstream as you will not be the committer, the upstream maintainer will be.

**Cons:**

- Resolving conflicts can be a bit more difficult as you will need to look up which commit is causing the conflict using git log/git blame, it will not directly tell you.
- Rebasing is difficult as you cannot rebase a merge, it will offer to cherry-pick all of the commit individually. However, you should not be rebasing often, instead using git revert and git merge where possible.

I would recommend doing a cherry-pick to figure out any problem conflicts initially, doing a merge, then revert the problem commits afterwards so updating is easier (as merging is quicker after being up to date).

*Add the commits to your source, one version at a time*

The most important part of this process is the one version at a time part. There MAY be a problem patch in your upstream series, which could cause a problem with booting or break something like sound or charging (explained in the tips and tricks section). Doing incremental version changes is important for this reason, it's easier to find an issue in 50 commits than upwards of 2000 commits for some versions. I would only recommend doing a full merge once you know all of the problem commits and conflict resolutions.

## Cherry-picking

Format:

```
git cherry-pick <previous_tag>..<next_tag>
```

Example:

```
git cherry-pick v3.10.73..v3.10.74
```

## Merge

Format:

```
git merge <next_tag>
```

Example:

```
git merge v3.10.74
```

I recommend keeping track of the conflicts in merge commits by removing the # markers.

## How to Resolve Conflicts

We can't give a step-by-step guide for resolving every single conflict, as it involves a good knowledge of C language, but here's a few hints.

If you are merging, figure out what commit is causing the conflict. You can do this one of two ways:

1. `git log -p v$(make kernelversion)..<latest_tag>` to get the changes between your current version and the latest from upstream. The `-p` flag will give you the changes done by each commit so you can see.
  2. Run `git blame` on the file to get the hashes of each commit in the area. You can then run `git show --format=fuller` to see if the committer was from mainline/stable, Google, or CodeAurora.
- Figure out if you already have the commit. Some vendors like Google or CAF will attempt to look upstream for critical bugs, like the Dirty COW fix, and their backports could conflict with upstream's. You can run `git log -`

grep="<part\_of\_commit\_message>" and see if it returns anything. If it does, you can skip the commit (if cherry-picking using `git reset --hard && git cherry-pick --continue`) or ignore the conflicts (remove the <<<<< and everything between the ===== and >>>>>).

- Figure out if there has been a backport that is messing up resolution. Google and CAF like to backport certain patches that stable wouldn't. Stable will often need to adapt the resolution of the mainline commit to the absence of certain patches that Google opts to backport. You can look at the mainline commit by running `git show <hash>` (the mainline hash will be available in the commit message of the stable commit). If there is a backport messing it up, you can either discard the changes or you can use the mainline version (which is what you will usually need to do).
- Read what the commit is trying to do and see if the problem is already fixed. Sometimes CAF may fix a bug independent of upstream, meaning you can either overwrite their fix for upstream's or discard it, like above.

Otherwise, it may just be a result of a CAF/Google/OEM addition, in which case you just need to shuffle some things around.

Here is [a mirror of the linux-stable kernel.org repository](#) on GitHub, which can be easier for looking up commit lists and diffs for conflict resolution. I recommend going to the commit list view first and locating the problem commit to see the original diff to compare it to yours.

Example URL: <https://github.com/nathanchance/linux-stable/commits/linux-3.10.y/arch/arm64/mm/mmu.c>

You can also do it via the command line:

```
git log <current_version>..<version_being_added> <path>

git show <hash>
```

Solving resolutions is all about context. What you should ALWAYS do is make sure your final diff matches upstream's by running the following commands in two separate windows:

```
git diff HEAD

git diff v$(make kernelversion)..$(git tag --sort=-taggerdate -l v$(make kernelversion) | cut -d . -f 1,2)* | head -n1)
```

## Enable rerere

Git has a feature called rerere (stands for Reuse Recorded Resolution), meaning that when it detects a conflict, it will record how you resolved it so you can reuse it later. This is especially helpful for both chronic rebasers with both merging and cherry-picking as you will just need to run `git add . && git <merge|cherry-pick> --continue` when redoing the upstream bringup as the conflict will be resolved how you previously resolved it.

It can be enabled by running the following command in your kernel repo:

```
git config rerere.enabled true
```

## How to git bisect when running into a compiler or runtime error

Given that you will be adding a sizable number of commits, it's very possible for you to introduce a compiler or runtime error. Instead of just giving up, you can use git's built-in bisect tool to figure out the root cause of the issue! Ideally, you will be building and flashing every single kernel version as you add it so bisecting will take less time if needed but you can bisect 5000 commits without any issues.

What git bisect will do is take a range of commits, from where the issue is present to where it wasn't present, and then start halving the commit range, allowing you to build and test and let it know if it is good or not. It will continue this until it spits out the commit causing your issue. At that point, you can either fix it or revert it.

1. Start bisecting: `git bisect start`
2. Label the current revision as bad: `git bisect bad`
3. Label a revision as good: `git bisect good <sha1>`
4. Build with the new revision
5. Based on the result (if the issue is present or not), tell git: `git bisect good` OR `git bisect bad`
6. Rinse and repeat steps 4-5 until the problem commit is found!
7. Revert or fix the problem commit.

**NOTE:** Mergers will need to temporarily run `git rebase -i <good_sha>` to apply all the patches to your branch for proper bisecting, as bisecting with the merges in place will

often times checkout onto the upstream commits, meaning you have none of the Android specific commits. I can go into more depth on this upon request but trust me, it is needed. Once you have identified the problem commit, you can revert or rebase it into the merge.

## Do NOT squash upstream updates

A lot of new developers are tempted to do this as it is “cleaner” and “easier” to manage. This is terrible for a few reasons:

- Authorship is lost. It’s unfair to other developers to have their credit stripped for their work.
- Bisecting is impossible. If you squash a series of commits and something is an issue in that series, it’s impossible to tell what commit caused an issue in a squash.
- Future cherry-picks are harder. If you need to rebase with a squashed series, it is difficult/impossible to tell where a conflict results from.

## Subscribe to the Linux Kernel mailing list for timely updates

In order to get notified whenever there is an upstream update, subscribe to [the linux-kernel-announce list](#). This will allow you to get an email every time a new kernel is released so you can update and push as quick as possible.

# Building Mediatek Kernel

Many Android users enjoy installing customized kernels, which can offer a range of performance and battery life enhancing tweaks. But if you can’t find a kernel you like, or none are available for your device, sometimes you just have to build your own. This guide will focus on how to build a kernel from source for Mediatek devices.

Please be warned this guide is not for newbies, it is intended for people with an understanding of customizing Android ROMs, working in Linux terminals, and just overall a bit of working knowledge about what we’re doing.

## Requirements:

1. A Linux operating system

2. Some basic C knowledge and how to work with Makefiles
3. Android NDK

To begin, you're going to need to download the following packages for Linux:

- Python
- GNU Make
- JDK
- Git

```
sudo apt-get install git gnupg flex bison gperf build-essential zip curl  
libc6-dev libncurses5-dev:i386 x11proto-core-dev libx11-dev:i386  
libreadline6-dev:i386 libgl1-mesa-glx:i386 libgl1-mesa-dev g++-  
multilib mingw32 tofrodos python-markdown libxml2-utils xsltproc  
zlib1g-dev:i386 git-core lzop ccache gnupg flex bison gperf build-  
essential zip curl zlib1g-dev zlib1g-dev:i386 libc6-dev lib32ncurses5  
lib32z1 lib32bz2-1.0 lib32ncurses5-dev x11proto-core-dev libx11-  
dev:i386 libreadline6-dev:i386 lib32z-dev libgl1-mesa-glx:i386 libgl1-  
mesa-dev g++-multilib mingw32 tofrodos python-markdown libxml2-  
utils xsltproc readline-common libreadline6-dev libreadline6  
lib32readline-gplv2-dev libncurses5-dev lib32readline5 lib32readline6  
libreadline-dev libreadline6-dev:i386 libreadline6:i386 bzip2 libbz2-dev  
libbz2-1.0 libghc-bzlib-dev lib32bz2-dev libsdl1.2-dev libesd0-dev  
squashfs-tools pngcrush schedtool libwxgtk2.8-dev python gcc g++ cpp  
gcc-4.8 g++-4.8 && sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1  
/usr/lib/i386-linux-gnu/libGL.so
```

Now go to etc/udev/rules.d/51-android.rules:

```
# adb protocol on passion (Nexus One)  
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e12",  
MODE="0600", OWNER=""  
# fastboot protocol on passion (Nexus One)  
SUBSYSTEM=="usb", ATTR{idVendor}=="0bb4", ATTR{idProduct}=="0fff",  
MODE="0600", OWNER=""  
# adb protocol on crespo/crespo4g (Nexus S)
```



```
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e22",  
MODE="0600", OWNER=""  
# fastboot protocol on crespo/crespo4g (Nexus S)  
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e20",  
MODE="0600", OWNER=""  
# adb protocol on stingray/wingray (Xoom)  
SUBSYSTEM=="usb", ATTR{idVendor}=="22b8", ATTR{idProduct}=="70a9",  
MODE="0600", OWNER=""  
# fastboot protocol on stingray/wingray (Xoom)  
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="708c",  
MODE="0600", OWNER=""  
# adb protocol on maguro/toro (Galaxy Nexus)  
SUBSYSTEM=="usb", ATTR{idVendor}=="04e8", ATTR{idProduct}=="6860",  
MODE="0600", OWNER=""  
# fastboot protocol on maguro/toro (Galaxy Nexus)  
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e30",  
MODE="0600", OWNER=""  
# adb protocol on panda (PandaBoard)  
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d101",  
MODE="0600", OWNER=""  
# adb protocol on panda (PandaBoard ES)  
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="d002",  
MODE="0600", OWNER=""  
# fastboot protocol on panda (PandaBoard)  
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d022",  
MODE="0600", OWNER=""  
# usbboot protocol on panda (PandaBoard)  
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d00f",  
MODE="0600", OWNER=""  
# usbboot protocol on panda (PandaBoard ES)  
SUBSYSTEM=="usb", ATTR{idVendor}=="0451", ATTR{idProduct}=="d010",  
MODE="0600", OWNER=""  
# adb protocol on grouper/tilapia (Nexus 7)  
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e42",  
MODE="0600", OWNER=""
```

```
# fastboot protocol on grouper/tilapia (Nexus 7)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4e40",
MODE="0600", OWNER=""
# adb protocol on manta (Nexus 10)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4ee2",
MODE="0600", OWNER=""
# fastboot protocol on manta (Nexus 10)
SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", ATTR{idProduct}=="4ee0",
MODE="0600", OWNER=""
```

And in bash.rc:

```
export USE_CCACHE=1
```

Now finally:

```
sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/lib/i386-linux-gnu/libGL.so
```

So now we're ready to set up the build environment. In the terminal, type:

```
export TARGET_BUILD_VARIANT=user TARGET_PRODUCT=devicename
MTK_ROOT_CUSTOM=../mediatek/custom/ TARGET_KERNEL_V
```

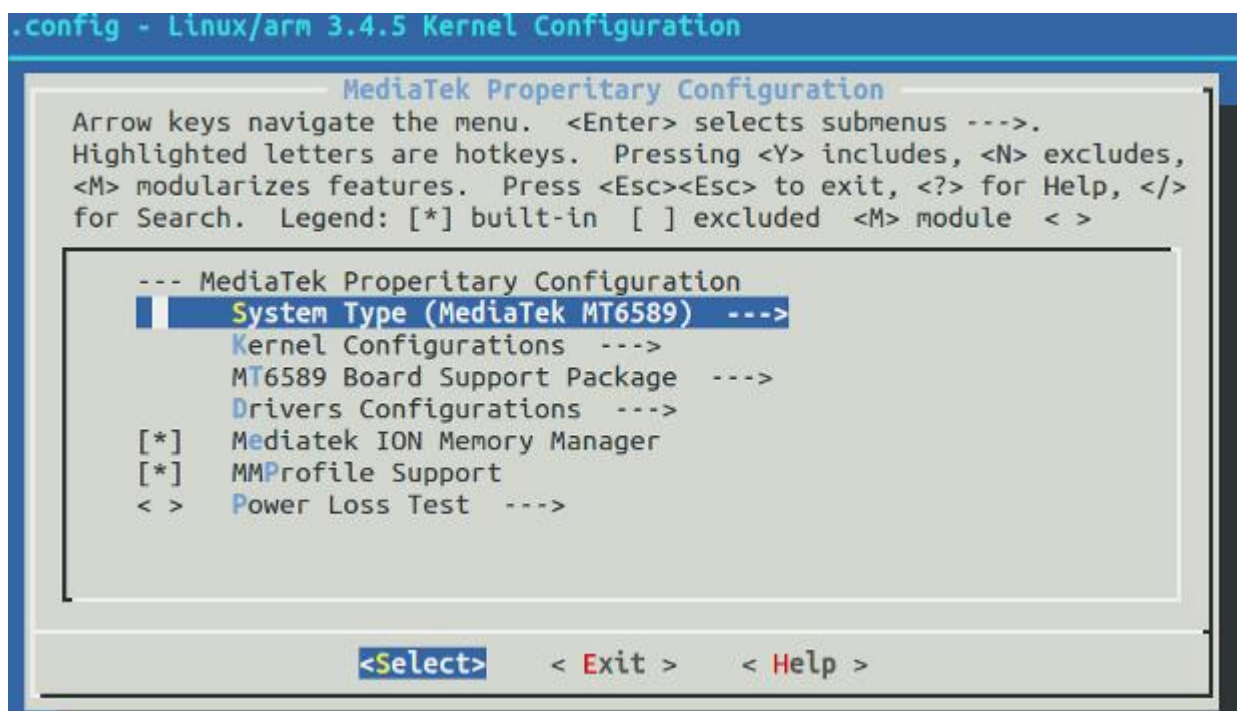
Here's what these commands are going to do:

- BUILD\_VARIANT: specifies what the kernel is going to be built for.
- TARGET\_PRODUCT/TARGET\_KERNEL\_PRODUCT: tells Linux which device specific files to use.
- MTK\_ROOT\_CUSTOM: specifies the directory of the mediatek/custom folder. remember this mide be in the same directory as the kernel source as well.  
PATH: sets your toolchain executables to your path.
- CROSS\_COMPILE : A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running. The toolchain facilitates this function

ARCH=arm, ARM is a family of instruction set architectures for computer processors based on a reduced instruction set computing (RISC) architecture developed by British company ARM Holdings. ARM is also used in Android.

So when we type 'export ARCH=arm' into the terminal, we're basically telling Linux that we're building for the ARM architecture.

So now we're ready to begin configuring the kernel. You need to be extremely careful, because the kernel is basically the controller for your phone. So just follow along carefully.



You'll most likely find the base config in the kernel\_source/mediatek/config/devicename/autoconfig/kconfig/platform.

We can use this base config and build it with different requirements, for example SELinux permissions enabled or disabled. You could always just build a base config from scratch, but I really don't recommend it.

So now let's type into the Linux terminal:

```
cd kernel_source
cp mediatek/config/devicename/autoconfig/kconfig/platform .config
make menuconfig
```

This is going to create a graphical interface that will allow you to add features to the kernel. For example, you can tweak the I/O Schedule, CPU Governors, GPU Frequency, etc.

Once you've tweaked your desired settings, you're ready to compile the kernel. So type into the Linux terminal:

```
make zImage
```

And it should return something like:

```
arch/arm/boot/zImage Ready
```