Halium docs Documentation

Release

Marius Gripsgard, Bhushan Shah, Luca Weiss, JBBgameich, Rada

Aug 23, 2017

Contents

This repository contains the following documents:

CHAPTER 1

Contents

Planning

Linux kernel + Android + Libhybris common basel

Introduction

Overall idea is, This stack includes,

- Linux kernel
- Android HAL
- Sensors
- Camera
- RILd
- Libhybris
- Android HAL interfaces like Audioflingerglue and droidmedia
- Build system and scripts
- GPS - AGPS from Mozilla
- Pulseaudio
- Media codecs
- oFono
- Way to distribute updates and images (way to install) (that's debatable)
- Systemd? (Upstart can handle user-level service initialization: See Ubuntu Touch)

This stack doesn't include the,

- Qt

- No Qt would be best, most platforms are very strict on the version they depend on
- Wayland
- KWin hwcomposer platform
- Plasma
- Unity
- Mir
- Lipstick
- Gecko
- Applications
- ....

Action Points

- See if there are any conflicting requirements and how best to resolve them. For eg. Ubuntu touch needs apparmor patches in kernel, while Sailfish doesn't.
- Decide what the stack will contain and make a short summary of the whole thing.
- Decide how the lxc/container should be setup (img file like ubuntu or in rootfs like sailfish)
- Decide what will be the default method to accrue android, build our own or take vendors and modify that (both would be best here in my opinion)
- What infrastructure we will need for this? What are options?
- Fork https://github.com/mickybart/gnulinux_support / make pull requests?
- Find all kernel features systemd requires, maybe fork mer-kernel-check to automatically check kernel configs
- Fork https://github.com/ubports/ubports-installer for cross platform installations
- Fork ubp-5.1 && ubp-7.0
- Common support for Multirom :) YES PLEASE I don't think Multirom is maintained anymore, but we can fork it can continue it ; There's already a fork with active developers: https://github.com/multirom-dev
- Please let's move away from android recovery update method (it does not fit us) (Boot from sdcard using efidroid at least for testing, so write images to sdcards?)
- Document all the things! YES! We need that badly

REVIEW ONCE THEN MOVE UP - bshah

What this base consists of? Or what is our "products"?

- AOSP source tree (LineageOS)
- Collection of Device Repos (similar to Cyanogenmod)
- Prebuilt and ready-to-integrate android images
- Reference packaging of libhybris and co. for the distributions / developers
- Reference binary images of this stuff
- Tool to flash the images

- Documentation on how to integrate the AOSP base with Linux system (this is most important.. Basically everything we create needs to be documented from start)
- Common config system?

What I would expect from this is a minimal booting android with libhybris, and all the libhybris tests pass

What is the lifecycle of port?

- Porting team decides upon target (or someone suggests it)
- We create android device and vendor tree or "fork" it from LineageOS/CM/AOSP/whatever
- Integrate our changes required for libhybris etc
- CI builds image and publishes them

Distributions (Plasma Mobile, UBports, Mer etc)

- Distributions picks up this new packaging if they are using our reference packaging for userspace (Linux) parts, otherwise they build their own packages.
- Along with rootfs generated from side of Distribution, they use the android IMG from the port lifecycle. (kernel depends on splitting / not splitting android and kernel)

The Stack - proposal

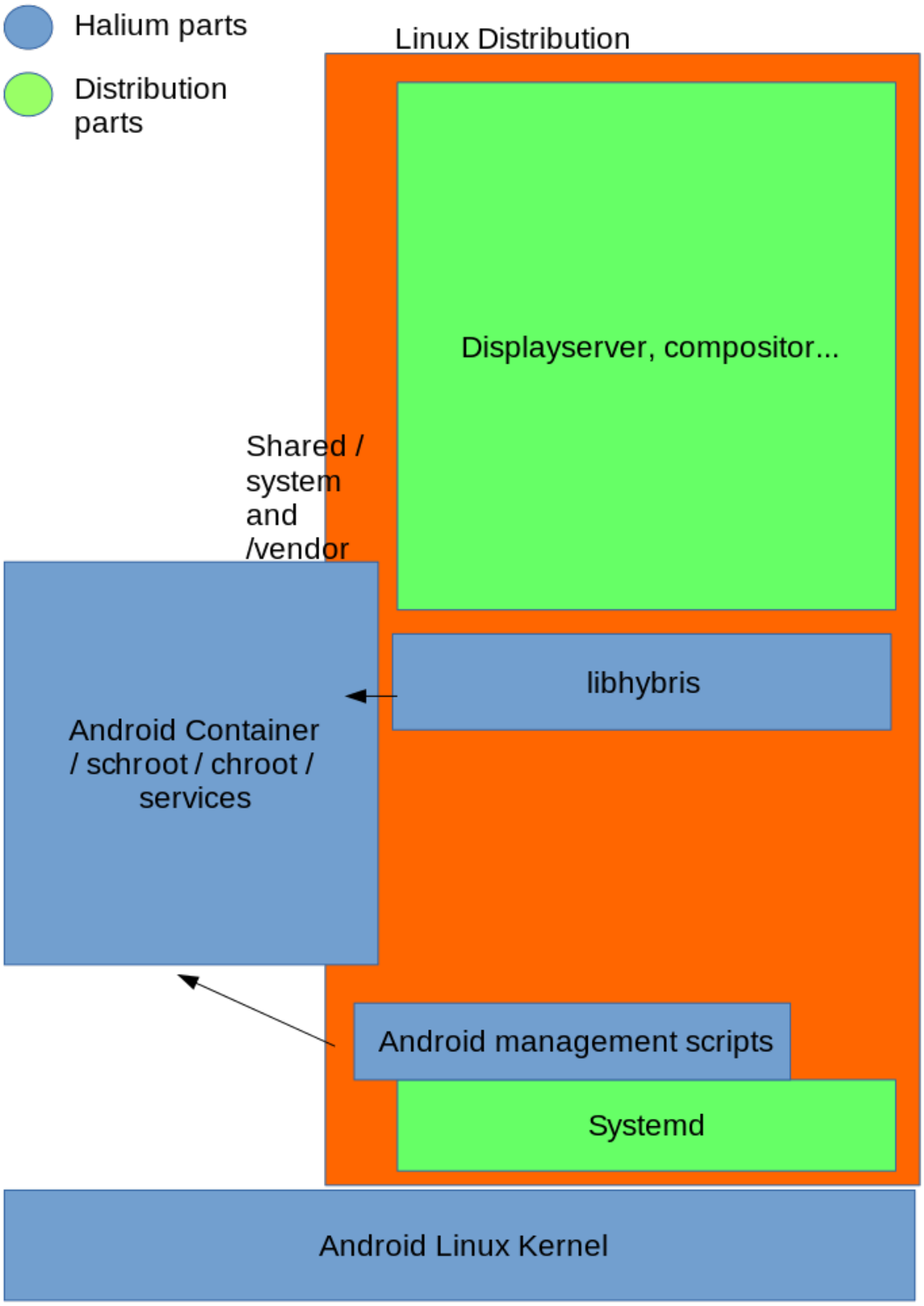
The system is running on a android manufacturer linux kernel, IF the device is mainlined, it can run mainline linux kernel

(Initramfs?)

Systemd start's up the system and the android HAL, which runs either in a schroot, container, or directly on the system.

Graphic output, camera and other sensors are managed by libhybris.

The linux distribution running with this stack can either use packages of libhybris etc. offered by us, if it's compatible, or build it's own


Halium parts
Distribution parts
Linux Distribution
Displayserver, compositor...
Shared / system and /vendor
libhybris
Android Container / schroot / chroot / services
Android management scripts
Systemd
Android Linux Kernel

Some random links

Mer-meeting takeaway

Jolla guys potentially interested but too early to commit to this. We need a proper proof-of-concept to show mer could run on top. Sailfish OS community devs of course anyone is welcome.

Jolla was concerned about how our things work with the ODMs way of things. This may not be a problem for community projects like ubports, plasma, et. al. but especially for Jolla since they're dealing straight with the ODMs. Something to take into account when we build our infrastructure. We should make it as flexible as possible so that this would work with them also.

Initial halium creator script (locusf rambling)

1. Plug in phone, run adb, which pulls in needed binaries from /system
2. builds halium kernel + the boot selection (still wondering what this could be)
3. boot selector then fetches/runs the actual os inside some runtime (container/switch_root up to debate)

Halium kernel means both the actual kernel + middleware needed in order to have a common libhybris base from the running android system

Development

Development plan/outline:

Goal: being able to `adb shell` / `telnet` / `SSH` into the `GNU/Linux` system where you can run tests related to `android` hardware enablement and they work out of the box.

Reference device(s): `Nexus 5`, `Oneplus one`, `Nexus 5X` (It will be interesting to see how easy it makes it for `Ubuntu Touch` once `Halium` works for this device).

Reference rootfs: `Ubuntu ARM` or `ArchLinux ARM` or `Debian ARM` or `Fedora ARM` (Requirement: `systemd` as main init system)

Initial development (stage 0, libhybris):

- [ ] Fork the Android source tree with libhybris patches
- [ ] Bring continuous integration system up to build the android image
- [ ] Start with the basic rootfs (ubuntu 16.04)
- [ ] Build and integrate the libhybris (upstream) to the base rootfs
- [ ] Create publishing infrastructure for the distribution of the android image and rootfs
- [ ] At this point it's fine if not all tests "pass" but they should at least run.

Hardware enablement (stage 1):

- [ ] After stage 0 we will have a system to work on where we are able to run the `libhybris` tests and get the logs required, in this stage we can pick random hardware component and get it working
- [ ] At end of this stage, Halium is "ready" for distributions

Device enablement (stage 2):

- [ ] During this stage we will have most of the infrastructure ready for including new devices, so this is the time to go wild ;-)

Tests: We need to write tests that can run in our reference rootfs to make it easier to spot problems, we need both direct tests that live in `android` and tests that use `libhybris`.

Automated tests: These tests will run run after each `CI` build to ensure the builds that comes out of our build servers are working. (@UBports has some tests we can use for this)

Future ideas

- [ ] Common upstream msm kernel for all devices
- [ ] Common sets of qcom drivers
- [ ] All qcom devices to caf
- [ ] Freedreno for mainlined devices
- [ ] Backport 5.1 and 6.0 kernel driver required by 5.1 and 6.0 blobs
- [ ] Upstream the Android N port of libhybris from UBports fork
- [ ] Common place to put device configs for userspace (not talking about android device configs here, but the config each OS uses to configure different parts)
- [ ] Create a *translator* that converts common configs to OS specific configs
- [ ] Possibly something to "update" kernel from distribution packaging. I remember @micky-bart/gnulinux_support has something around that.
- [ ] Create a sandbox env to test *only* the hal and libhybris

Distribution

Reference rootfs

To make development and porting easier, halium team provides the reference rootfs which is based on Ubuntu 16.04. It includes the following,

- systemd
- lxc
- lxc-android
- libhybris
- ofono

- ofono-scripts

This reference rootfs is built using the live-build tool, documentation on how you can build it locally using rootfs-builder is available at https://github.com/halium/rootfs-builder/tree/ubuntu

---

Deployment

There are various parts which needs to be deployed on the device,

- boot.img
- rootfs
- udev rules
- lxc container configuration
- system.img
- vendor.img (for newer devices)
- android ramdisk

boot.img

boot.img is based on hybris-boot, this boot.img is built inside the android tree.

boot.img have really simple function,

Mount the data partition, and the rootfs.img in it to /target

```
mount $DATA_PARTITION /data
mount /data/rootfs.img /target
```

And switch to the target rootfs and run the init (systemd)

```
exec switch_root /target $INIT --log-target=kmsg &> /target/init-stderrout
```

rootfs.img

rootfs is provided by the distribution, in the tar.gz format, installation tool will extract and put the rootfs in the ext2 or ext4 rootfs.img, this rootfs.img will be mounted by the hybris-boot later on.

Following are the minimal requirements of the such rootfs

- systemd as the main init
- lxc

Depending upon required functionality, you may need

- ofono (for calling and mobile data)
- libhybris
- NetworkManager
- Pulseaudio

udev rules

rootfs.img contains the udev rules which are device specific and is generated from the ueventd*.rc file from device tree.

TODO: document how to generate them

You can either,

a) copy them to /lib/udev/rules.d/ during the initial installation of rootfs b) Deploy all the supported udev rules in the /usr/lib/lxc-android/, and have a systemd service to copy the device udev rule before starting udev service.

lxc container configuration

This is most important bit, which starts the android init, this is required to start the android binary daemons etc.

There are two parts of the lxc container configuration

- config
- pre-start.sh

They are provided at : https://github.com/Halium/lxc-android/blob/master/var/lib/lxc/android/

Configuration file specifies the following options

- lxc.rootfs
- lxc.network.type
- lxc.devttydir
- lxc.tty
- lxc.pts
- lxc.arch
- lxc.cap.drop
- lxc.pivotdir (deprecated)
- lxc.hook.pre-start
- lxc.init_cmd
- lxc.aa_profile (optional, only for distribution using apparmor)
- lxc.autodev

pre-start hook is used to extract and configure the android rootfs before booting into it.

system.img and vendor.img

This is the android libraries and vendor binary blobs, this is generated by building the android tree, they are deployed to the userdata partition at installation time.

Android rootfs

Android rootfs is provided by the halium, this is generated by the android build system, android rootfs contains what is usually contained inside the initrd of android's boot.img, this is extracted by the android lxc container's pre-start hook before starting container. This is located at /system/boot/android-ramdisk.img.

Startup sequence

- Fastboot start the kernel and loads the initrd
- initrd will mount the userdata partition and rootfs.img from it
- After mounting rootfs.img it will start the systemd init from the rootfs
- Rootfs is expected to mount the /system, /vendor and other android mount points before local-fs.target
- After local-fs target, lxc container is started
- lxc pre-start hook will bind mount the mounted android partitions inside the android rootfs
- Once android container is started, host system will start the udev and other system daemons
- At this point scope of halium is over and userspace services like sddm, mir, lipstic etc can be started

Porting guide

Setup your environment

Setting up your environment is necessary before starting the porting process.

Using Debian (Stretch or newer) or Ubuntu (16.04 LTS or newer)

Step 1: Installing required packages

If you are on the `amd64` architecture (commonly referred to as 64 bit), enable the usage of the `i386` architecture:

```
sudo dpkg --add-architecture i386
```

Install the required dependencies:

```
sudo apt install git gnupg flex bison gperf build-essential \
  zip bzr curl libc6-dev libncurses5-dev:i386 x11proto-core-dev \
  libx11-dev:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 \
  libgl1-mesa-dev g++-multilib mingw-w64-i686-dev tofrodos \
  python-markdown libxml2-utils xsltproc zlib1g-dev:i386 schedtool \
  repo liblz4-tool bc
```

What is repo?

Repo is a tool written by the Android developers for working on Android source trees. It downloads large numbers of git projects into a repeatable directory structure, important for something as complicated as Android. To do this, Repo uses a manifest, which is simply an XML document that tells it what to get and where to put it. You can find a full reference for the repo command here here, but we'll mainly be using the repo init and repo sync commands (and repo diff and repo status for analyzing the changes made to the device-specific parts so that its easier later to commit them to a GitHub repo).

//TODO: Add instructions for installing build tools for other distros as well

Step 2: Create a new directory to download the Halium tree

```
mkdir halium && cd halium
```

This directory will be called BUILDDIR in the remaining part of the guide, when necessary, to avoid confusion.

If the target device has Android 7.1 or LineageOS 14.1 support, it's recommended to select `halium-7.1`:

```
repo init -u https://github.com/Halium/android -b halium-7.1
```

If your device does not have Android 7.1 or LineageOS 14.1 support but has support for Android 5.1 or CyanogenMod 12.1, select `halium-5.1`:

```
repo init -u https://github.com/Halium/android -b halium-5.1
```

//TODO: Add notes about halium-5.1 and halium-7.1

`halium-7.1` is based on LineageOS 14.1 `halium-5.1` is based on CyanogenMod 12.1

Step 3: Download the source code

Note: the repo tool takes some time to download the sources. You need a little patience here. Execute the following:

```
repo sync -c
```

Specifying `-c` only downloads one branch, which makes the download a lot smaller since it won't download all of the old branches, this will also result in a much faster sync time.

You can also specify `-j[num]` as it will fetch the files simultaneously. This defaults to 6 consecutive processes but bumping it to 10 makes a big difference on network with high bandwidth.

Prepare the Android tree

Put parts together

Now you need to put all the parts for your device together, and since our tree is based on LineageOS for `halium-7.1` or CyanogenMod for `halium-5.1` you can use the device files they provide.

Parts that are needed:

- Device tree (e.g. from LineageOS)
- Kernel source (e.g. also from LineageOS)

- Vendor tree (e.g. from TheMuppets)

LineageOS and TheMuppets have a lot of devices already prepared, but it is not mandatory to use only their sources. Basically any tree capable of producing a working (Android) ROM should suffice.

You might have to check `lineage.dependencies`/`cm.dependencies` that is included in every LineageOS/CyanogenMod device repo for additional repositories.

Create a local manifest

The repo tool will accept additional repositories to be synced on top of the ones defined by Halium already. Since you are porting for a new device, you also need to do this work. After you have a working configuration your local manifest.xml can go into an issue in the Halium/projectmanagement repo .

Create additional entries for the repo tool to download the required parts automatically:

```
cd <BUILDDIR>/.repo && mkdir local_manifests && cd local_manifests
```

Create a new file called `<VENDOR>_<CODENAME>.xml` and open it in your favorite editor.

For the time being, add entries to the created xml file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
<!-- Example for a device with one specific vendor repo, and 2 repos for the device,␣
↪a common and a specific one. At the end, a common kernel repo -->
  <project path="vendor/samsung" name="proprietary_vendor_samsung" remote="them" />
  <project name="android_device_samsung_smdk4412-common" path="device/samsung/
↪smdk4412-common" remote="los" />
  <project name="android_device_samsung_i9300" path="device/samsung/i9300" remote="los
↪" />
  <project name="android_kernel_samsung_smdk4412" path="kernel/samsung/smdk4412"␣
↪remote="los" />

  <!-- Example for additional repos gathered from the dependencies files in the␣
↪device or vendor repo -->
  <project name="android_hardware_samsung" path="hardware/samsung" remote="los" />
  <project name="android_external_stlport" path="external/stlport" remote="los" />

</manifest>
```

The remote properties “los” and “them” are shortcuts for the already defined remotes that Halium provides. They can be reviewed by looking into default.xml in the .repo/manifests directory (It is also symlinked to .repo/manifest.xml). Depending on which Halium branch you used, default.xml will be either set up for LineageOS 14.1 or CyanogenMod 12.1. You can create references to new remotes by adding the following snippet in front of any project definition:

```
  <remote name="new_remote"
        fetch="http://github.com/<path_to_project>"
        revision="refs/heads/<branchname>" />
```

It is preferred to add them to your local manifest, and leave the default.xml in its original state.

After your local manifest is finished, you need once again to call the repo tool to download the added parts:

```
repo sync -c
```

repo will detect most mistakes in your local manifest. Sometimes, if you misspelled an URL for example, you need to tell repo to overwrite already prepared local settings:

```
repo sync -c --force-sync
```

Modify the kernel configuration

Halium uses the systemd as the init system which requires various kernel config options to be enabled.

To check which config options needs to be enabled we use mer-kernel-check utility provided by mer-hybris.

```
git clone https://github.com/mer-hybris/mer-kernel-check
cd mer-kernel-check
./mer_verify_kernel_config <path to kernel configuration>
```

If you don't know the path to your kernel config run `grep "TARGET_KERNEL_CONFIG" device/<VENDOR>/<CODENAME>/BoardConfig.mk`. It should be in `arch/arm/configs/<CONFIG>` or `arch/arm64/configs/<CONFIG>` depending on the architecture of your device.

//TODO: Mention that the config parameters CONFIG_IKCONFIG and CONFIG_IKCONFIG_PROC need to be set to y, otherwise Halium wont boot (or add them to the check script

Include your device in fixup-mountpoints script

First check if the codename of your device is already included in the `<BUILDDIR>/halium/hybris-boot/fixup-mountpoints` script.

If it's not already included, you will need to add a few lines similar to following in the `fixup-mountpoints` script for all partitions that are mountable (i.e. have an `fstype` of `ext4`, `vfat`, `f2fs` or others). You can ignore the rest of the partitions.

To figure out the actual device node for the block device you can use following command:

```
readlink -f /dev/block/platform/msm_sdcc.1/by-name/<blockdevicename>
```

Building

Initialize

First we need to initialize the environment using the envsetup.sh tool:

```
source build/envsetup.sh
```

This will give you an output that looks like this:

```
including device/samsung/maguro/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/samsung/tuna/vendorsetup.sh
including device/samsung/manta/vendorsetup.sh
including device/samsung/crespo4g/vendorsetup.sh
including device/samsung/torospr/vendorsetup.sh
including device/generic/armv7-a-neon/vendorsetup.sh
including device/generic/x86/vendorsetup.sh
including device/oneplus/bacon/vendorsetup.sh
including device/lge/hammerhead/vendorsetup.sh
including device/lge/mako/vendorsetup.sh
including device/asus/tilapia/vendorsetup.sh
```

```
including device/asus/deb/vendorsetup.sh
including device/asus/flo/vendorsetup.sh
including device/asus/grouper/vendorsetup.sh
```

//TODO: This only works if either in halium-5.1 add_lunch_combo has been executed or in halium-7.1 breakfast has been used, see below comment

Choose you target

Now we need to choose the target to build using the lunch command:

```
lunch
```

The output of this command will look something like this:

```
You're building on Linux

Lunch menu... pick a combo:
 1. aosp_arm64-eng   4. aosp_mips-eng     7. cm_bacon-eng
 2. aosp_arm-eng     5. aosp_x86_64-eng   8. cm_bacon-user
 3. aosp_mips64-eng  6. aosp_x86-eng      9. cm_bacon-userdebug

Which would you like? [aosp_arm-eng]
```

Here you need to choose your device `cm_[your device]-userdebug`, example if you were to build the OnePlus One you would choose `cm_bacon-userdebug` or `9`.

//TODO: LineageOS recommends the breakfast tool instead of lunch.

Building the system.img and hybris-boot.img

Halium will use the mkbootimg tool for creating the boot image. In most cases it is not on the local harddisk, so it can be built by issuing

```
mka mkbootimg
```

To build the `system.img` and `hybris-boot.img` - required for Halium - use the following commands

```
mka hybris-boot
mka systemimage
```

If you use `make` and not `mka` it's recommended to set `-j[num]` to do parallel building, this reduces build time. *NOTE* this is not needed on halium-7.1 since it will calculate parallel building based on CPU performance.

Once you have `hybris-boot.img` and `system.img` built successfully you can move to testing this on your device.

//TODO: add testing instructions in this file

Booting the device

After you built `hybris-boot.img` it is time to test it out.

1. Connect your device with a USB cable to your desktop.

2. Start the device in bootloader (aka "fastboot") mode. This is usually acomplished by holding some combination of keys on the device for a few seconds. On some devices this is Volume-Down + Power. The Lineage OS wiki is a good resource to check for this.
3. Change to the directory where the boot images have been created with the command `cout`.
4. Flash the kernel image with: `fastboot flash boot hybris-boot.img`. You can use `fastboot boot hybris-boot.img` to try to boot the kernel image without flashing, but that is not working for some devices.
5. Next reboot into the recovery. You can use the latest version of TWRP recovery to deploy the rootfs and system.img. Please read the following guide for further information.
6. Once done you can reboot the device.

Debugging

Reading kernel logs

To find out what happened during an unsuccessful boot you can check the kernel log files. These can usually be retrieved from `/proc/last_kmsg` after rebooting the device into another, working system. A precondition for this is that you have a system that you can successfully boot, such as TWRP.

1. Boot your newly built image
2. Wait for it to fail
3. Reboot the device into the working system.
4. Retrieve the kernel log with `adb shell cat /proc/last_kmsg > ~/last_kmsg`
5. Read `~/last_kmsg` and find out what went wrong

Debugging via telnet

The hybris-boot image can offer you a telnet interface to access the system on the device even if it does not come up fully. To this end, the usb function of the device will be reconfigured to work as a network interface. While bringing this network interface up, the boot image will write a few debug messages. These debug messages are communicated via a clever hack of resetting the serial number of the usb connection. Once you see that the usb networking and telnet have been set up, you can configure the usb networking on your desktop and then telnet into the system.

The steps in detail are:

- Execute this command to watch the changes in the usb serial number: `while : ; do lsusb -v 2>/dev/null \ | grep -Ee 'iSerial +[0-9]+ +[^ ]' done \ | uniq`
- Boot your newly built image
- Watch the output of the lsusb command above. It will put out lines like this:

```
iSerial                 3 01234567
iSerial                 3 Mer Debug setting up (DONE_SWITCH=no)
iSerial                 3 Mer Debug telnet on port 23 on usb0 192.168.2.15 - also␣
↪running udhcpd
```

- Determine the name of the usb network device on your desktop: `dmesg | tail`. You're looking for a line similar to this:

```
[ 1234.123456] rndis_host 1-7:1.0 enp0s20f0u7: renamed from usb0
```

- In this example shown above, `enp0s20f0u7` is the usb network device name. Use this for the USBNETWORK below
- Check if the usb network device has a MAC address assigned.

```
$ ip address show dev USBNETWORK
6: USBNETWORK: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default␣
↪qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
```

If it shows the link/ether address `00:00:00:00:00:00` as shown above, you will have to manually assign the MAC address,

```
ip link set USBNETWORK address 02:01:02:03:04:08
```

You can set any MAC address you want, it just needs to be a valid MAC address.

- Configure usb networking:

```
sudo ip address add 192.168.2.1 dev USBNETWORK
ip address show dev USBNETWORK
sudo ip route  add 192.168.2.15 dev USBNETWORK
ping -c 2 192.168.2.15
```

- Connect with telnet: `telnet 192.168.2.15`

Now you have terminal access to the system running from initrd.

The `hybris-boot.img` brings up the telnet interface if it detects a problem. Alternatively, you can use the `hybris-recovery.img` image which will always start telnet.

Porting parts

Debugging the Android userspace

Debugging the Android userspace

Logcat

Logcat is a tool that reads

```
/system/bin/logcat
```

For radio logs

```
/system/bin/logcat -b radio
```

dmesg

Even though Android logs does not normally end up in dmesg, early initialization of Android and kernel output ends up here.

```
dmesg
```

Debug Libhybris crash

One of the main problems with the current Hybris based architecture, is the lack of symbols resolution and mapping once a crash happens at the Android layer. To workaround this we need to manuly import non stripped libaries

Graphics

Graphics is an essential part of Halium. Halium uses libhybris to make Android's bionic based hardware adaptations layer usable with glibc systems.

Tests

Here is some tests to test the graphics stack

```
test_hwcomposter
```

```
test_egl
```

```
test_egl_config
```

```
test_glesv2
```

//TODO: add tests using more heavy graphics applications using Wayland (mir)

Lights

Lights is one of the simple parts to get going in Halium.

Tests

```
test_lights
```

The device's light should start flashing now.

Vibrator

vibrator is one of the simple parts to get going in Halium, and is often used as to goto test to check if libhybris linker works as expected.

Tests

```
test_vibrator
```

The device should now vibrate

Wifi

Tests

Testing wifi is done using NetworkManager

```
nmcli d
```

Qcom devices

Wifi is fairly easy to get going on most Qcom based devices

```
echo 1 > /dev/wcnss_wlan
echo sta > /sys/module/wlan/parameters/fwpath
```

That should enable wifi

broadcom bcmdhd

It is recommended to modularize bcmdhd drivers since that will ensure use of HW MAC address.

In the kernel defconfigs, make sure these settings are set

```
CONFIG_MODULES=y
CONFIG_BCMDHD=m
```

Then add this to your devices init.rc file, it's recomended to set this in early stages to avoid race condition with network manager (`on post-fs-data` is a good place for this)

```
insmod /system/lib/modules/bcmdhd.ko
```

Deploying the rootfs and system.img

Once you have built the system.img from the android tree, you can download and install the rootfs using the `halium-install` script in a halium-scripts repository.

Currently latest rootfs is available at, bshah's personal server: Link

You can use the halium-install script as below, when device is connected in recovery mode

```
halium-install <path to rootfs tarball> <path to android system.img>
```

This will do the following,

- Convert the rootfs tarball into ext2 rootfs.img
- Convert system.img into the ext4 mountable image from android sparse image.
- Push the both images to /data partition

References

The hybris-boot image is based on work from the Sailfish OS and the Sailfish Hardware Adaptation Development Kit porting guide contains valuable tips.

supplementary hardware information

What to document here?

In this section, we want to gather information about

- the exact hardware built into our phones and tablets
- the software status for these devices, that is:
  - which OS is available in what version and to which degree of functionality
  - which kernel is available? What drivers are used for what part?
  - where are the sources for all this? What is their license?
- where can we reuse drivers/kernel patches/backports

Where to document this?

The documentation should preferably have a logical structure so that every specific group of devs we are trying to help here can access the info that is meant for them without having to read the other parts.

- Device Overview this is meant for linking to each devices sub page, show halium status (for porters) and list exact hardware parts (for kernel devs)
- Hardware Enablement this is more or less only for the kernel devs and should document mainlining/driver progress for all the hardware
- How 2 Document this is only for newcomers who want to know how to help with documenting here

Why is this important?

Halium is trying to reduce the workload for different OS teams by developing a common base. The deepest root in all of the OS systems, even the official ones, is the kernel, which is based upon the linux mainline kernel. For Halium to work, certain features need to be implemented into this kernel, for example the ability to use lxc.

Therefore it is of great interest to us to have a kernel for each device that has all the needed parts. The sense behind mainlining is that **you only care about the specific hardware you are working on**. All other features that are implemented by other people will just be handled mainline and if your code is mainline too, upon a kernel update, you can just use the new one with new features from other people, without having to reimplement your driver into this new kernel. So if e.g. lxc is updated or apparmor and we are working on an older kernel, at some point the rootfs we are working with will require this new version and we need to do all the implementation of our work to a new kernel.

By documenting things **that will never change** like the hardware built into a phone, the kernel version a driver was mainlined or general mainline status of hardware parts, we reduce the risk to document something that will change before it is ever looked at.

On the other hand, comparing the hardware of devices shows, that many parts are used across manufacturers, so many drivers may, at least partially, be cloned.

With documenting these things at a central point, we may make connections easier visible and by this enable kernel devs or newcomers to write/modify drivers and mainline them.

For our porters, we can gather info about available Android/Lineage/Cyanogenmod/... versions, where the sources and if available proprietary blobs are located and perhaps what challenges/problems may occur if any are known already.

Who is this for?

This sub section of the documentation is meant for

- **porters** as a first quick reference about the status for certain hardware parts (drivers) and whole devices (custom kernels & OSes)
- **community members** who want to help by documenting
- **kernel devs** as a reference which hardware is in what device, if there are open source drivers or where the official drivers are located