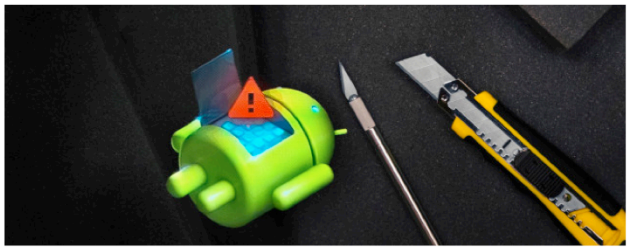

cuteboot

https://github.com/cuteboot

#cuteboot @ irc.freenode.net

A means to stuff your own QML UX on top of most Android devices [Today: a crude proof of concept, tomorrow, basis of products?]

By Carsten Munk (@stskeeps)
carsten.munk@gmail.com

This is the story about what happens if you do random experiments, because you one day wonder if things in your particular area of business aren't exactly as hard they seem. This is a talk about a proof of concept I and some others did called 'cuteboot', a means to stuff your own Qt QML UX, or other types of UI, on top of most Android devices. Originally, I wanted to give a live demo of this, as noted in the talk description but unfortunately murphy's law kicked in and I managed to bust up the USB port of my demo device, so you'll have to take my word that it does what it's supposed to – or will enable after the project has matured a bit.

About me

- Ex-CTO, R&D of Jolla
- Ex-maemo.org distmaster
- Project architect of Mer Core (basis of SailfishOS)
- Author of libhybris (used in SailfishOS, Asteroid, Ubuntu Touch, Luna, etc..)
- Available for Mobile Linux consulting

So quickly something about what I've been doing or am doing – I'm Ex-CTO, R&D of Jolla, a company that has made the Jolla phone and a very rare tablet and an alternative glibc-based mobile operating system called SailfishOS. SailfishOS was built on another project I made, called Mer Core, which is a collection of software packages making up a mobile operating system core. I also made a solution called 'libhybris' which enables glibc-based systems to leverage Android drivers w/ Wayland and other things that projects like SailfishOS, Asteroid, Ubuntu Touch etc are using as well. And as a last thing, I'm now independent, doing mobile linux consulting to finance my work into projects like this one on the side – and I'm available for contracting, if anybody is interested in what I could do for you w/ cuteboot or other things in the mobile space.

The philosophical problem

- The *"extended mind"* is an idea .. which holds that the reach of the mind need not end at the boundaries of skin and skull. Tools, instrument and other environmental props can under certain conditions also count as proper parts of our minds.
- What's running on the mobile devices we so rarely don't have on us or by our side?

Given my project history, you may wonder why on earth I'd embark on another semi-mobile OS like project Iread a particular philosophical paper by David Chalmers& Andy Clark titled "the extended mind", which propositions that the reach of the mind need not end at the boundaries of skin and skull, that tools, instruments and other props can under certain conditions also count as proper parts of our minds. My phone, personally, is like a extended memory and a digital augmentation to me. And these thoughts make me think that I'm and we're not in control of our minds – because we don't effectively control our devices. And this has haunted me ever since I read this paper.

While I've worked on solutions that enable us to build and have glibc-based systems on mobile devices, it's still way too hard, prone to showstoppers, compared to the speedy execution, despite the strong usage of ducttape, that essentially closed and Google-controlled devices are getting put into the market. That battle is uphill and very expensive, albeit not impossible – and right now, we're far from winning. We should instead of thinking about technology choices like glibc/dbus/wayland/etc versus bionic/binder/surfaceflinger/etc, think about our consumer, or even human freedom versus our lives and our data being controlled by big companies; where we can actually make a real impact on a much more equal footing. Is a battle of the technology of choice more important than that of the battle of freedom of our minds?

The opportunity

- Thousands of quite nice Android devices (ODM reference devices or on-market) out there with:
  - Open bootloader (so you can flash a boot image of your own choosing) & boot.img available
  - But often without any source codes available
  - Maybe sometimes kernel code but almost never full AOSP tree.
- Price rapidily going towards almost-zero. ODMs getting squeezed on margins. Every Android phone looks the same.
- What if we could leverage the existing Android supply chain and put our own UX experiences on them; freeing our minds?

But to get more technical, what exactly is the opportunity? The Android market has thousands upon thousands devices, on-market or available as reference devices available from manufacturers that at reasonably low prices can be produced with any system you'd want to put on it. As long as it derives somewhat from the existing software for the device and can be applied easily on top of typical AOSP-based source code tree, the barebones Android w/ hardware adaptation – be too distant and their manufacturing processes break down and your manufacturing price + time to market rises. For the whitelabel or aftermarket devices, many of them comes with open bootloaders and a boot img (kernel and initrd) available, but usually without any source code available, maybe sometimes a kernel source but almost never a reasonably full AOSP tree – so being content with having an Android device just because you could hack the AOSP tree is just not good enough (some exceptions apply like Nexus devices but even they don't run always run pure AOSP out of the box..) In addition to that, it's possible to work with manufacturers since their margins are getting hit the same and any kind of real differentation in phone space can matter -- Since every Android phone looks the same. What If we could leverage all this to put our own UX experiences on them, making us able to control our interaction and information flow, effectively freeing our minds? There's obvious other technical benefits, too, like untradtional products made from reference designs.

The typical solution – Make a ROM

- Build a Android source tree and patch it to match with the binary blobs and device settings
- Often with bad results, broken hardware support and other quirks and takes too long.
- Why? Because most non-published device vendor trees are patched right into hell with strange workarounds. Not to mention not always delivering right kernel source code.

Now, what is the usual way that people deal with this? They make a Android ROM. Take AOSP, patch and guesswork enough about the system on the device, add binary blobs on top until it works. However, It's often with bad results, broken hardware support and other quirks + the porting effort takes significant time. Why is this? Because most non-published device vendor trees are patched right into hell with strange workarounds. Not to mention not always delivering right kernel source code.

What if we could reuse the existing system on the device?

- Remove every APK from the installed system except for the non-visual middleware we want
- Boot up only essential services + handy things like SurfaceFlinger/RIL/AudioFlinger. Don't boot up the Android UX at all
- And instead run our own Qt/QML-based experience against SurfaceFlinger?
- Without rebuilding the system – the ultimate ROM?
- Turns out it's possible.

So what if instead of trying to build a ROM, we reuse the existing system on the device? Like for example, Removing every APK, booting up only essentially services, not booting up the Android UX. And instead running our own Qt/QML or otherwise experience against SurfaceFlinger, the Android compositor? What if we could make it the ultimate ROM that you could within 10-20 minutes adapt to a new device?

How to strip an Android bare?

- A bit of rm –f, adb logcat'ing and ps aux'ing ☺
- An boot.img contains a initrd w/ initialization scripts (init*rc) that control the startup process
  - Where you can disable the boot animation
  - And add your own things to start up (like a QML UX)
- And a default.prop in the boot.img where you can add things like:
  - config.disable_noncore=1, ro.secure=0 , ro.debuggable=1, ro.adb.secure=0, persist.sys.usb.config=mtp,adb
- And strip APKs from /system/app and /system/priv-app until only the services you want start up
- Gives nice 200mb RAM middleware that is fully functional (can probably be stripped down more RAM/storage wise)

So, how is this done in practice? First thing you'd want to do is to stop Android from booting up anything but it's minimal set. But how do you actually strip a Android system bare? It takes a bit of Jenga-style development, as in: guess a lot and keep removing things until everything falls apart – which you can witness when that happens in the logs over ADB and ps aux in adb shell. And some modification and patching of the boot image to customize the startup process. Android already provides enough knobs to tune to make this easy. In the end, you end up with a set of simple instructions to modify the state of an Android device – replace boot.img, delete a few things in the system directories, change a few setttings. And bam, you have a nice 200mb RAM middleware that's fully functional with telephony, wifi, 3g, bluetooth, nfc, composition, multimedia and camera that you can build on instead of trying to re-create on your own. 200mb RAM is before tuning the Dalvik/ART memory usage, which can probably be significantly reduced.

Getting a Qt UX on top

- Most of the support needed is already in Qt
  - Plus a few patches to make it more up to date with modern Android NDK w/o JNI
- A matter of using the right ./configure options and cross-compilers from AOSP and providing a bit of glue
- SurfaceFlinger plugin was available but didn't work properly with modern AOSP. Builds against an AOSP tree, not a NDK sysroot.
- Binder-accessed services AIDL are full of Parcelables (custom serialization routines written in Java, not very friendly for C++/QML)
- Adding a privileged APK ("Java bridge") requires full re-signing of framework.jar and other privileged apk's

We found when we started hacking at this proof of concept, that the most of the support to run Qt on top of 'bare' android was already in Qt, where we only had to patch it a bit to work with a bit more modern Android NDK. Then it was a matter of using the right configure options and the right cross compilers from AOSP along with a bit of Makefile glue, to get it building. Naturally, a bare Qt isn't terribly interesting, it'll work, build, but it can't show anything on the screen. SurfaceFlinger access, as in, manually creating windows w/o the involvement of the Dalvik VM is not something that's available as headers or even libraries in the Android NDK – so you end up having to build against a semi-built AOSP source tree which has the right headers. Because of the Qt QPA system and general pluginability of Qt we can make those parts hardware adaptation specific. The second problem is however that while that we have a lot of nice services running in the Android middleware, which theoretically is described in AIDL interface description files that we could build C++ bindings for… but in practice most of them have a lot of "Parcelabes" which means "i'll just write the serialization code in java manually" which is naturally a pain in the ass if we'd like to interface with those from C++. So the obvious choice is to make a java bridge that we communicate with over IPC, but the problem is that adding a privileged APK to an Android system when you don't have the platform key, which you usually don't, requires a re-cryptographic signing of the system framework jars and other privileged apks with a new platform key.

AIDL

```
interface IWifiManager
{
    int getSupportedFeatures();

    WifiActivityEnergyInfo reportActivityInfo();

    List<WifiConfiguration> getConfiguredNetworks();

    List<WifiConfiguration> getPrivilegedConfiguredNetworks();

    WifiConfiguration getMatchingWifiConfig(in ScanResult scanResult);

    int addOrUpdateNetwork(in WifiConfiguration config);

    boolean removeNetwork(int netId);
```

And just to help understand the power of having this middleware available, this is an example from the AIDL, interface descriptor for the WiFi Manager.. And you can imagine that being able to access those functions makes it quite easy to do a quick QML applications that does the things you want. Similar AIDLs are available for other very useful services.

To summarize, cuteboot brings

- A set of source code repositories and build scripts enabling:
  - Building Qt + Qt Declarative etc against Android NDK (device-independent sysroot) w/o JNI
  - Building hardware adaptation plugins (Qt Multimedia, SurfaceFlinger, etc) against specific (generic) AOSP version
  - With a simple 'build world' approach

Now you're naturally wondering what cuteboot itself brings to the game – we now have a middleware, but naturally, the system doesn't do a lot. So what we essentially bring with cuteboot is set of source code repositories and build scripts, not currently very good ones, enabling us to build Building Qt + Qt Declarative etc against a typical Android NDK sysroot w/o making to use JNI (and hence having to be an Android APK) and hardware adaptation plugins against the AOSP source code – which can be generic, and not have to exactly match the system running on the device. This is because we only use the header files effectively. Doing it in this manner makes us able to perhaps build a central (and signed) Qt/Qt Declarative build shared across many devices, and add hardware adaptations bits, so a main Qt-based UX can evolve independently.

Roadmap

- Java bridge w/ gRPC enabling easy access to Binder-accessed Java services (WiFi, etc)
- Installing into /system instead of 'cache' partition
- Installable with .zip file like a ROM
- Two-parts: NDK API level dependent, and AOSP version dependent

So, naturally, this is currently a proof of concept. But we want to take it further, and during February we'll start doing these things.. Actually make the Java bridge, make it work, installing cuteboot into /system instead of our custom cache partition like we do currently, and make it possible to easily install Cuteboot-based systems to any device like you'd install any other ROM, through .zip file based recoveries. If we're lucky, we get into a situation where we can do a central build for a project, and then a .zip file on top that adds the hardware adaptation. This could prove useful to have OS releases in two different paces.

Intended workflow

- Get a boot.img matching the software running in your device & patch it
- Build and install Qt & such into /system on device
- Build Java bridge, and access handy middleware functions over gRPC (Protobuf-based) w/ grpc++ or protobuf-qml
- Re-sign APKs and .jars on device w/ new platform key
- Build your dream

And the result would be that making the foundation for your device would simply be to get a boot img, patch it, build qt, build a java bridge, re-sign your APKs on your device.. And then spend more time building your dream experience. Could this be easier?

Buying guide for best hackability

- Devices with open bootloaders (and ideally open flashers)
- With available firmware images for download (so we get boot.img)
- With Android source tree downloadable for them (that actually can be built and flashed and is fully functional)
  - So you can patch the Android middleware for security, etc.
  - Like Fairphone 2.
- Where you can re-lock your bootloader and sign your boot images with own key (Evil Maid).

And naturally, you will be wondering, OK, so, I now have the ability to control the user experience of my device; but what about the fact I now have a huge amount of closed source on my device, that is, the Android system? The answer is to buy devices and support those companies that give devices with open bootloaders, flashers, available firmware images, a full Android source tree available for download + perhaps a tiny bit of vendor blobs, that can be made into a fully functional OS, so you can patch the Android middleware for security, etc. One such example that I know of, is the Fairphone 2. And as last thing, on my own personal wishlist, a device where you can unlock and re-lock your bootloader, add your own signing keys, to secure up your device from Evil Maid attacks.

One more thing: The Human Web

- The Human Web project seeks to empower everybody with the ability to:
  - Access, protect, control and share their identities and data
  - Securely communicate with machine and other human beings alike
  - Trade and participate in smart contracts
  - Collaborate and coordinate by themselves or together with others
  - Evolve and share their individual user experiences with others; like practices and ideas (memes) spread between people
  - Do all of the above without having to be tied to any middlemen
- #thehumanweb on irc.freenode.net

Naturally, you might wonder what use I personally would have for cuteboot and this is a bit of a moonshot project I call the Human Web project, a bit of continuation of the philosophical thoughts in the start of this talk. We're a bunch of people who are unhappy about the direction mobile devices are going and the unacceptable surveillance and centralization of services. We build on awesome technologies such as QML, IPFS, Ethereum, Tor, CRDTs, Containers/Sandboxing and so on to do this. What we seek to do is to empower everybody with the peer-to-peer ability to; … So if you're interested in how a very different take on user experience and decentralization could be like on our mobile devices and desktop alike, feel free to join #thehumanweb on irc.freenode.net and join in the discussions and possibly contribute to the open source project. All kinds of contributions welcome.

Thank you

- Any questions?
- Start at #cuteboot on irc.freenode.net for guidance
- github.com/cuteboot
- Android 4.4 and up preferred.
- @stskeeps on Twitter, Stskeeps on irc.freenode.net

Thanks for listening. If you'd like to play around with cuteboot, I suggest your first stop is #cuteboot on IRC, on freenode, to get guidance on how to proceed, and then go on from there; ideally helping documenting as you go on. Android 4.4 devices and up are preferred but older versions also possible.