Porting mainline Linux to mobile phones

FOSDEM 2022
Who am I?

- Luca Weiss (z3ntu)
- Mainlining phones since 2017 for fun
- postmarketOS core team member
- Also OpenRazer maintainer amongst other things
- Android Platform Engineer at Fairphone
  - but here on my own
The Android kernel situation

- Every device has their own kernel source tree
- Security issues need to be patched in each kernel
- Greg K-H: apply all "fixes", not just "security fixes"
- Lots of out-of-tree code
- Questionable code quality

https://source.android.com/devices/architecture/kernel/generic-kernel-image
The Android Mess

- **Linux kernel is licensed under GPL-2.0**
  - Every modified Linux kernel must be open-source as well
  - Some vendors don’t seem to care much… (but that’s another topic)

- **SoC vendors implement drivers in user space**
  - Gets around license requirements, potentially easier to hire people
  - Camera, Bluetooth amongst others
  - Most have kernel component but need user space counterpart
  - Qualcomm has some open-source parts **but** important parts are proprietary

- **Building 100% open-source operating system is basically impossible with that**
  - Ubuntu Touch reuses Android binaries through libhybris

- **Unless you get a proper kernel using standardized interfaces**
  - Be gone, proprietary bits!
Hardware description on x86

- Kernel needs to know what hardware is available
- On x86 ACPI and ACPI tables exist
  - Used to discover and configure hardware components
  - And power management bits
- Also USB & PCI enumeration
  - USB: Touchpad, SD card reader, webcam
  - PCI: Wifi card, ethernet controller, GPU
  - Both use vendor ID - product ID identifier
  - Various USB device classes (e.g. HID)

```c
static const struct pci_device_id rtl8169_pci_tbl[] = {
    // ...
    { PCI_VDEVICE(REALTEK, 0x8168) },
    // ...
    {}
};
```
Why it’s so difficult on (ARM) phones

- ARM doesn’t use ACPI
- Historically Linux used board files
  - C file compiled into the kernel describing the hardware
  - Kernel image can run on this specific board only
- Since around Linux 3.x device tree!
  - Originally used in SPARC computers
  - “Open Firmware” project
- Structured format to describe hardware
  - e.g. clocks, GPIOs, memory addresses and more!
- Loaded by (theoretically) generic kernel
  - Drivers get instantiated and used based on this information
  - Operating-system independent
Why mainline Linux

● Tremendous learning opportunity
  ○ Understand how hardware works from kernel perspective
  ○ Outside of formal education and jobs this is hard to acquire

● Run open-source software without proprietary user space
  ○ postmarketOS!
  ○ Mobian
  ○ Android (aospm)

● Code maintained at kernel.org
  ○ Fixes and features have a place to go

● Learn valuable engineering skills
  ○ Constant problem solving and debugging

● Brag about it online!
A devicetree file

.dts (devicetree source) => .dtb (devicetree binary)

```
#include "foobar.dtsi"

/dts-v1/
{
    #address-cells = <2>;
    #size-cells = <2>;

    soc: soc@0 {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges = <0 0 0 0 0x10 0>;
        dma-ranges = <0 0 0 0 0x10 0>;
        compatible = "simple-bus";

        rng@793000 {
            compatible = "qcom,prng-ee";
            reg = <0 0x00793000 0 0x1000>;
        }
    }
}
```
A devicetree node

Basic node has some or all of:

- Compatible string: `compatible = "qcom,prng-ee";`
- Memory address (with size): `reg = <0x00793000 0x1000>;`
- Clocks: `clocks = <&gcc GCC_PRNG_AHB_CLK>;`
- Pinctrl:
  - `pinctrl-names = "default";`
  - `pinctrl-0 = <&qup_uart2_default>;`
- GPIOs: `gpios = <&pm6350_gpios 2 GPIO_ACTIVE_LOW>;`
- Many other properties - check binding docs!
Let’s get it running with mainline!

- **Focusing on Qualcomm**
  - Probably best mainstream phone SoC vendor for mainline

- **Age of device matters**
  - New SoCs have ground work done by paid people
  - Old SoCs had time to mature to get features and iron out bugs

- **SoC bringup is more involved**
  - Pinctrl, clocks, basic devicetree parts
  - Cannot recommend as first target

- **Many SoCs exist, not all are supported**
  - Check postmarketOS wiki and Linux repo for existing work done
  - Much code exists in random repos - ask people!
Preparations

● Unlocked bootloader
  ○ obviously
● Stock firmware
  ○ In case something goes wrong
  ○ For charging the battery
● TWRP
  ○ Fast uncomplicated environment as reference
● Know the device features
  ○ e.g. notification LED, NFC, video out over USB-C
● Downstream kernel sources
  ○ Ready to build and modify
● Know your UART
  ○ If you can
Reconnaissance

- Input devices (touchscreen & physical buttons): /dev/input
- Display panel: /proc/cmdline
- Backlight: Check /sys for file brightness
- Fuel gauge: Check /sys for file capacity
- Internal storage & SD card
  - On which bus (SDHCl or UFS)
  - Also note down partition labels
- Vibration motor
- Wifi chipset/driver
- Bluetooth chipset/driver
Full devicetree

- Newer devices contain DTBO partition
- DTBO = device tree binary overlay
- Combines base dtb + dtbo to form full devicetree
- Extract from running device to get full dtb
  - `/sys/firmware/fdt` or `tar'ing /sys/firmware`
  - Use `dtc` to get dtb/dts
  - Also keep original .dts handy!
Booting

● Start with booting something simple - get boot.img packaging correct
  ○ lk2nd (if available)
  ○ Downstream kernel

● Get qcom,board-id / qcom,msm-id / dtbo correct

● Without UART you will need to guess some things
  ○ Possible to get simple-framebuffer / USB working without it (if you’re lucky)

● With UART you can check if kernel is booting at all

● Each bootloader has its own quirks
  ○ Learn to work with and around their limitations
  ○ Newer ones tend to be more problematic with dtbo, vendor_boot, etc. support
I²C example - downstream

An I²C bus (defined elsewhere)

```
$soc {
  i2c@f9924000 {
    ilitek@41{
      compatible = "ilitek,2120";
      reg = <0x41>;
      interrupt-parent = <&msmgpio>;
      interrupts = <28 0x2>;
      vcc_i2c-supply = <&pm8941_s3>;
      ilitek,name = "ilitek i2c";
      ilitek,reset-gpio = <&msmgpio 55 0x00>;
      ilitek,irq-gpio = <&msmgpio 28 0x02>;
      ilitek,vbus = "vcc_i2c";
      ilitek,power-enable-gpio = <&msmgpio 25 0>;
    }
  }
}
```

Ilitek ILI2120 touchscreen

I²C address 0x41

Interrupt on gpio 28

pm8941_s3 supplies power

Reset is gpio 55

Interrupt from above again

And another gpio for power
Mainline Linux has a driver for us!

Ilitek ILI210x/ILI2117/ILI2120/ILI251x touchscreen controller

Required properties:
- compatible:
  - ilitek,ili210x for ILI210x
  - ilitek,ili2117 for ILI2117
  - ilitek,ili2120 for ILI2120
  - ilitek,ili251x for ILI251x

- reg: The I2C address of the device

- interrupts: The sink for the touchscreen's IRQ output
  See ../interrupt-controller/interrupts.txt

Optional properties for main touchpad device:
- reset-gpios: GPIO specifier for the touchscreen's reset pin (active low)

Example:

```
touchscreen@41 {
    compatible = "ilitek,ili251x";
    reg = <0x41>;
    interrupt-parent = <&gpio4>;
    interrupts = <7 IRQ_TYPE_EDGE_FALLING>;
    reset-gpios = <&gpio5 21 GPIO_ACTIVE_LOW>;
};
```

Documentation/devicetree/bindings/input/ilitek,ili2xxx.txt
The node in mainline

```c
i2c@f9924000 {
  status = "ok";
  // Enable this i2c bus (most are status = "disabled");
}

touchscreen@41 {
  compatible = "ilitek,ili210x";
  reg = <0x41>;
  interrupt-parent = <&msmgpio>;
  interrupts = <28 IRQ_TYPE_EDGE_FALLING>;
  reset-gpios = <&msmgpio 55 GPIO_ACTIVE_LOW>;
};
```

- Same i2c bus as downstream
- Compatible for ILI2120 from doc
- Same i2c address as downstream
- Same interrupt as downstream (but with proper constant as 2nd arg)
- Same reset gpio as downstream (but again with proper constant)
Sometimes it’s more difficult...

- Bluetooth is defined without direct reference to UART transport

```c
bt_nitrous {
    compatible = "goog,nitrous";
    uart-port = <0>;
    power-gpio = <&msmgpio 34 0>;
    host-wake-gpio = <&msmgpio 48 0>;
    host-wake-polarity = <l>;
    dev-wake-gpio = <&msmgpio 61 0>;
    dev-wake-polarity = <l>;
};
```

```c
uart_0: serial@f991d000{
    compatible = "qcom,msm-huart-v14";
    reg = <0xf991d000 0x1000>, <0xf9904000 0x19000>;
    reg-names = "core_mem", "bam_mem";
};
```

In this case seems to refers to the UART below.
Or this Wifi example..

- Downstream dmesg is helpful - use grep!
- Registered in board file: arch/arm/mach-msm/board_wifi_bcm.c
- Determined “very likely” on: sdhc_3: sdhci@f9864900

```c
#define WLAN_POWER 35
#define WLAN_HOSTWAKE 46

static unsigned wlan_wakes_msm[] = {
    GPIO_CFG(WLAN_HOSTWAKE, 0, GPIO_CFG_INPUT,
             GPIO_CFG_NO_PULL, GPIO_CFG_2MA) }

/* for wifi power supply */
static unsigned wifi_config_power_on[] = {
    GPIO_CFG(WLAN_POWER, 0, GPIO_CFG_OUTPUT,
             GPIO_CFG_PULL_UP, GPIO_CFG_2MA) }
```
Getting more components supported

● Structured todo list might be useful
  ○ e.g. Kanban board

● Look through downstream dts what you might be missing
  ○ For non user facing components, like prng
  ○ Ignore SoC debug things - you probably cannot use or test them anyways

● Compared downstream vs. mainline to understand
  ○ Understand how downstream code gets “transformed” into mainline code
  ○ e.g. look at downstream MDP, compare with existing mainline MDP
  ○ Then attempt to add support for your variant
Tips

● Keep notes to reference
● Save your kconfig!
  ○ You can spend hours debugging what went wrong
  ○ I like to commit them into the repo (make savedefconfig)
● Kernel cmdline too
● There are useful cmdline options for bringup
  ○ clk_ignore_unused: doesn’t disable unused clocks
  ○ pd_ignore_unused: doesn’t disable unused power domains
● Know that i2c addresses are sometimes given as “8-bit” instead of “7-bit”
  ○ 8-bit: 0x50 read, 0x51 write => 7-bit: 0x28
● Devices with same SoC tend to be similar
  ○ Qualcomm provides a referenced design to OEMs
  ○ Some manufacturers diverge more than others
● Downstream binding documentation often exists
  ○ To understand obscure properties
Handy hardware utilities

- **USB meter**
  - Draining or charging the battery?
  - Or even fast charging?
- **Multiplexed UARTs**
  - USB-A breakout board
  - Headphone jack
Smartwatches!

Not just phones, some older watches use Snapdragon 400! Some newer ones use “Snapdragon Wear” SoCs
Conclusion

- Mainlining is neither **simple** nor **fast**
- Takes a lot of time and dedication
  - Understand these concepts is difficult
  - I’m still missing knowledge in some important areas
    - Interconnect
    - Audio
    - Cameras
    - IOMMU
- Reward: knowing you brought up mainline on some hardware!
  - And of course the eternal fame of your name in Linux git history
Further resources

- [wiki.postmarketos.org](http://wiki.postmarketos.org) - many pages with useful information
- [#mainline:postmarketos.org](https://mainline:postmarketos.org) - lots of helpful people
- Other SoC-specific Matrix/IRC channels
Thank you for watching!