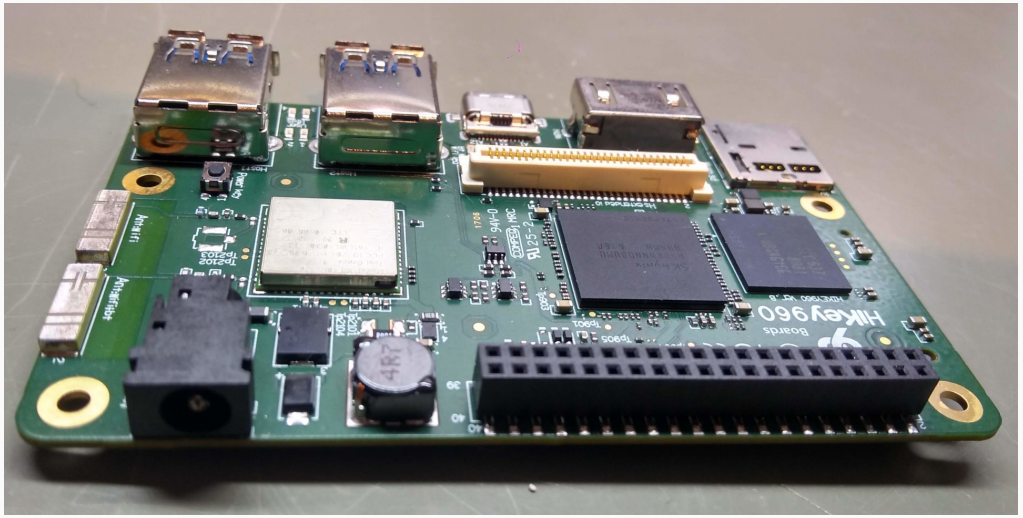
Linux auf ARMs

Hunz
hunz AT mailbox.org

6. April 2019

Einleitung & Hinweise

Motivation



- Warum eigenes System aufsetzen?
- System selber bauen vs. Blackbox
- Dinge verstehen
- schlanke, maßgeschneiderte Systeme
- Neue Features durch aktuelle Kernel
- Sicherheitsupdates
- Stromspartechniken, Performancegewinne

ARM Prozessoren/SoCs

- SoC: System On a Chip - CPU mit Peripherie
- hier: nur Systeme mit MMU (keine Mikrocontroller)

- ARMv7 Architektur: 32Bit
  → **armhf** (ARM hard floatingpoint)
  (früher: armel (ARM endian little))
- z.B. Cortex-A15, A9, A8, A7, A5

- ARMv8: 64Bit
  → **aarch64** (Userland) bzw. **arm64** (Linux Kernel)
- z.B. Cortex-A73, A72, A57, A55, A53, ...
- Ausnahme: Cortex-A32 (32Bit)

cross-compiling

- Kompilieren für andere Architektur (ARM)
- Erfordert cross-compiler:
  crossbuild-essential-armhf für 32Bit,
  crossbuild-essential-arm64 für 64Bit
- Nutzung mittels
  export CROSS_COMPILE=arm-linux-gnueabihf- bzw.
  export CROSS_COMPILE=aarch64-linux-gnu-
- Prefix für Compiler, Assembler, etc.
- Beispiel: normaler Compiler: gcc
  Crosscompiler: aarch64-linux-gnu-gcc
- **Buildsystem der Software muss CROSS_COMPILE unterstützen**
- z.B. Linux Kernel, U-Boot Bootloader

Hardware

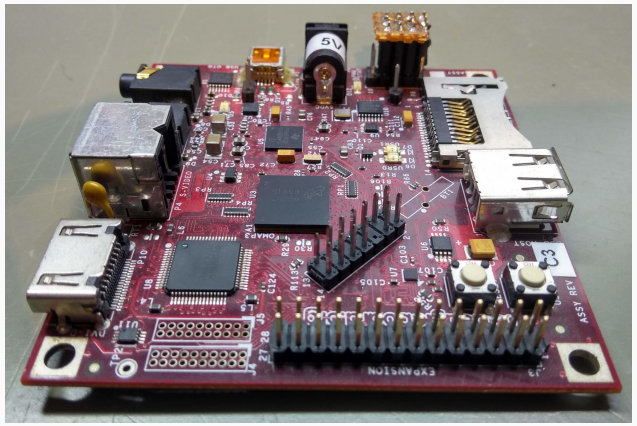
Der Raspberry Pi...

- **ist kacke** (aber billig...)

- kaum Doku verfügbar, kein richtiges Datenblatt
- Broadcom und Open Source ist keine Geschichte der großen Liebe
- Chip kann man nicht einzeln kaufen
- nur als Raspberry Boards verfügbar

Alternativen

- Microchip SoCs: Cortex-A5 https://www.acmesystems.it
- Texas Instruments: Cortex-A8 Beagle devices https://beagleboard.org/
- NXP - Wandboards: Cortex-A7/A9 https://www.wandboard.org/
- RockPi: Schneller Rockchip SoC https://rockpi.eu/



- Odroid N2: 4x Cortex-A73, 4x Cortex-A53 https://www.hardkernel.com/shop/odroid-n2-with-4gbyte-ram/
- Listen: https://www.cnx-software.com, https://www.hackerboards.com, https://en.wikipedia.org/wiki/Comparison_of_single-board_computers

Serielle Konsole

- bei Spielereien unverzichtbar
- für Zugriff auf Bootloader nötig
- Linuxshell ohne Netzwerk
- debugging
- Kernel-Ausgaben

- Terminal Programm: z.B. minicom

Serielle Konsole - Beispiel

```
U-Boot 2017.09-02089-g6bf7d40 (Jan 04 2019 - 20:14:05 +0800)

Model: Rockchip RK3399 Evaluation Board
DRAM:  3.9 GiB
Relocation Offset is: f5c0b000
PMIC:  RK808
pwm-regulator(vdd-center): init 950000 uV
vdd_center 950000 uV
regulator(vdd-center) init 950000 uV
MMC:   dwmmc@fe320000: 1, sdhci@fe330000: 0
Using default environment

Warn: can't find connect driver
Failed to found available display route
Warn: can't find connect driver
Failed to found available display route
In:    serial
Out:   serial
Err:   serial
Model: Rockchip RK3399 Evaluation Board
switch to partitions #0, OK
mmc1 is current device
```

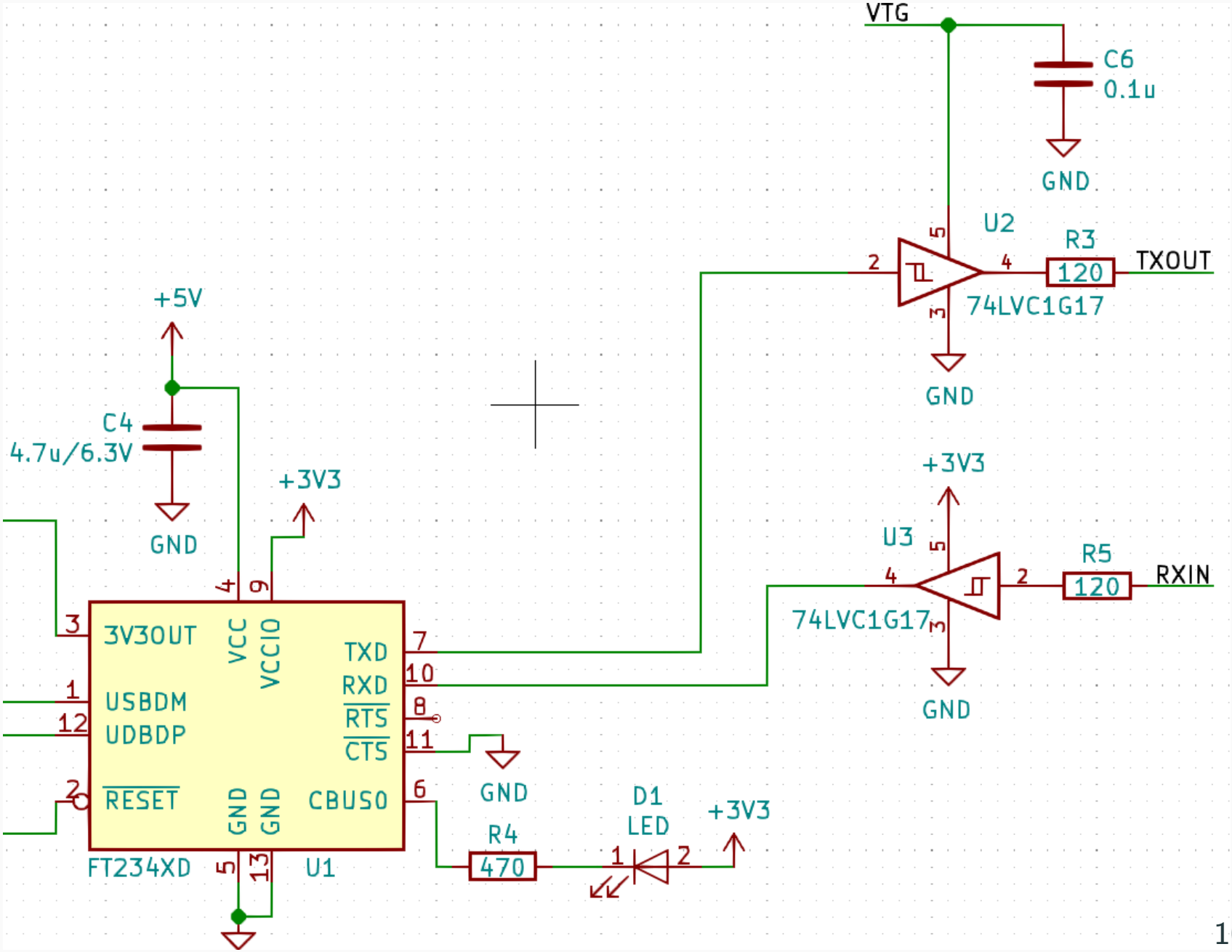
Serielle Konsole - Hardware

- Belegung für Boards findet man mit Google
- USB-to-serial devices

- klassisches RS-232: $\pm$9V Spannungspegel
- bei embedded devices: 3.3V oder 1.8V Pegel
- **falsche Spannung kann Gerät zerstören**

- buffered usb-to-serial devices benutzen wenn möglich
- **Eingang** für Spannung des Zielboards
- verhindert Schäden am Board ohne Spannungsversorgung

Serielle Konsole - buffered



Serielle Konsole mit unbuffered usb2serial

- Widerstände als Strombegrenzung in Rx/Tx (z.B. 330 Ohm)
- Board Tx → USB Rx nur verbinden wenn usb2serial Spannung hat
- USB Tx → Board Rx nur verbinden wenn Board Spannung hat

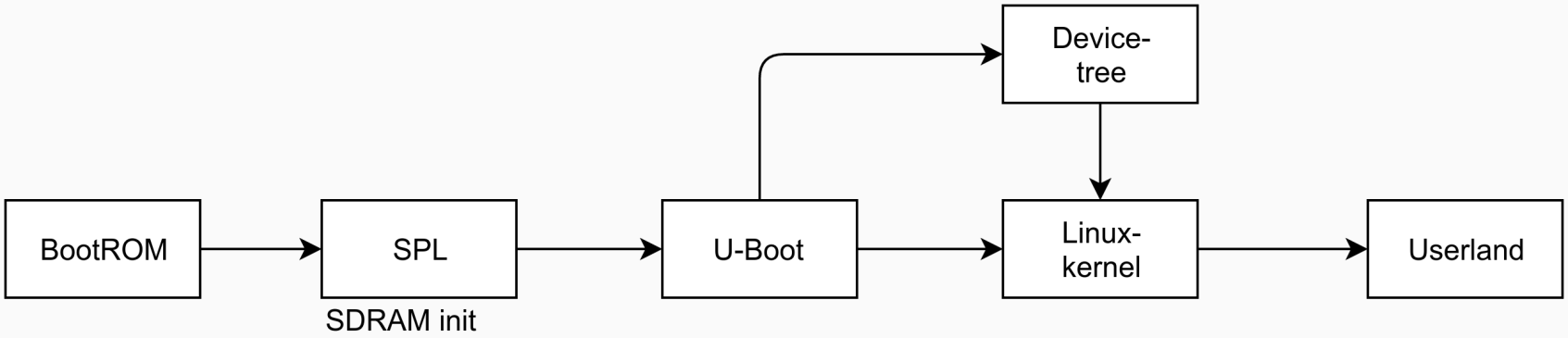
Beispiel:

- usb2serial an USB Port, Board GND an usb2serial GND
- Board Tx → USB Rx verbinden
- Board mit Spannung versorgen
- USB Tx → Board Rx verbinden

- vor powerdown Verbindungen rückwärts trennen

Der Bootprozess

Linux Bootprozess



1. Boot-ROM der CPU
2. SPL: secondary program loader / second stage loader
   (hier teilweise noch mehr Zwischenschritte)
3. third stage loader (U-Boot)
4. Linux Kernel
5. Linux Userland

Bootprozess: Boot-ROM

- Masken-ROM im SoC
- meist $<$64 kByte
- normalerweise keine UI (serielle Konsole) verfügbar
- Auswertung von Fuses
- Wahl des Bootmodus
- Code von Bootmedium laden & ausführen

Second stage loader

- U-Boot meist zu groß für internes SRAM
  → SDRAM wird benötigt
- Initialisierung nötig
  → primäre Aufgabe des 2nd stage loaders
- SPL läuft aus internem SRAM oder Cache
- exakte Initsequenz abhängig von Board und RAM-Baustein(en)
  → Boardspezifisch
- initialisiert ggf. weitere Peripherie
- lädt & startet U-Boot

Second stage loader - Ausgabe

```
channel 0 training pass!
channel 1 training pass!
change freq to 800MHz 1,0
ch 0 ddrconfig = 0x101, ddrsize = 0x2020
ch 1 ddrconfig = 0x101, ddrsize = 0x2020
pmugrf_os_reg[2] = 0x3AA1FAA1, stride = 0xD
OUT
Boot1: 2018-08-06, version: 1.15
CPUId = 0x0
ChipType = 0x10, 220
mmc: ERROR: SDHCI ERR:cmd:0x102,stat:0x18000
mmc: ERROR: Card did not respond to voltage select!
emmc reinit
mmc: ERROR: SDHCI ERR:cmd:0x102,stat:0x18000
mmc: ERROR: Card did not respond to voltage select!
emmc reinit
mmc: ERROR: SDHCI ERR:cmd:0x102,stat:0x18000
mmc: ERROR: Card did not respond to voltage select!
SdmmcInit=2 1
mmc0:cmd5,20
SdmmcInit=0 0
```

Second stage loader: Optionen

- Hersteller kann 2nd stage loader zur Verfügung stellen

- U-Boot Secondary Program Loader (SPL)
- im U-Boot Projekt enthalten

- kein 2nd stage loader,
  stattdessen Device Configuration Data (DCD)
- wird bei Freescale/NXP i.MX genutzt
- Array von Speicheraddressen und Werten
- enthält Registerwerte für SDRAM-Initsequenz
- wird vom Boot-ROM geladen und ausgewertet

U-Boot (Universal Bootloader)

- `http://www.denx.de/wiki/U-Boot/WebHome`
- sehr mächtiger Bootloader, open source (GPL v2 Lizenz)
- Initialisierung der Systemtakte
- Zugriff (read/write) auf nahezu alle bekannten Flash-Speicher
- Zugriff auf Busse: SPI, I2C, USB
- Netzwerkfähig (Boot-Image kann z.B. via BOOTP/DHCP + TFTP geladen werden)
- konfigurierbar, persistenter Konfigurationsspeicher, einfaches scripting
- lesen von einzelnen Dateien aus Dateisystemen
- Unterstützung für grafische Bootlogos (framebuffer)
- serielle Konsole zur Bedienung/Konfiguration

Linux Kernel

- Kernel kann meist alles ausser SDRAM selbst initialisieren
- SDRAM wird aus Platzgründen benötigt
- serielle Konsole mit debug-Ausgaben verfügbar
- meist Nutzung von komprimiertem Kernel mit eingebautem Entpacker ((b)zImage)

- Unterschied PC/SoC: kein ACPI, Peripherie nicht über PCI-Bus angebunden
- keine automatische Peripherie-Erkennung
- Kernel benötigt Peripherie-Liste mit Speicheraddressen
  $\rightarrow$ **device-tree**
- später mehr

U-Boot Bootloader

U-Boot

- Git repository: `http://git.denx.de/?p=u-boot.git`
- klonen mit **git clone git://git.denx.de/u-boot.git**

- verfügbare board-configs im **configs**-Unterverzeichnis
- code für boards liegt im **board/hersteller/boardname**-Unterverzeichnis
- Dokumentation in **doc/**

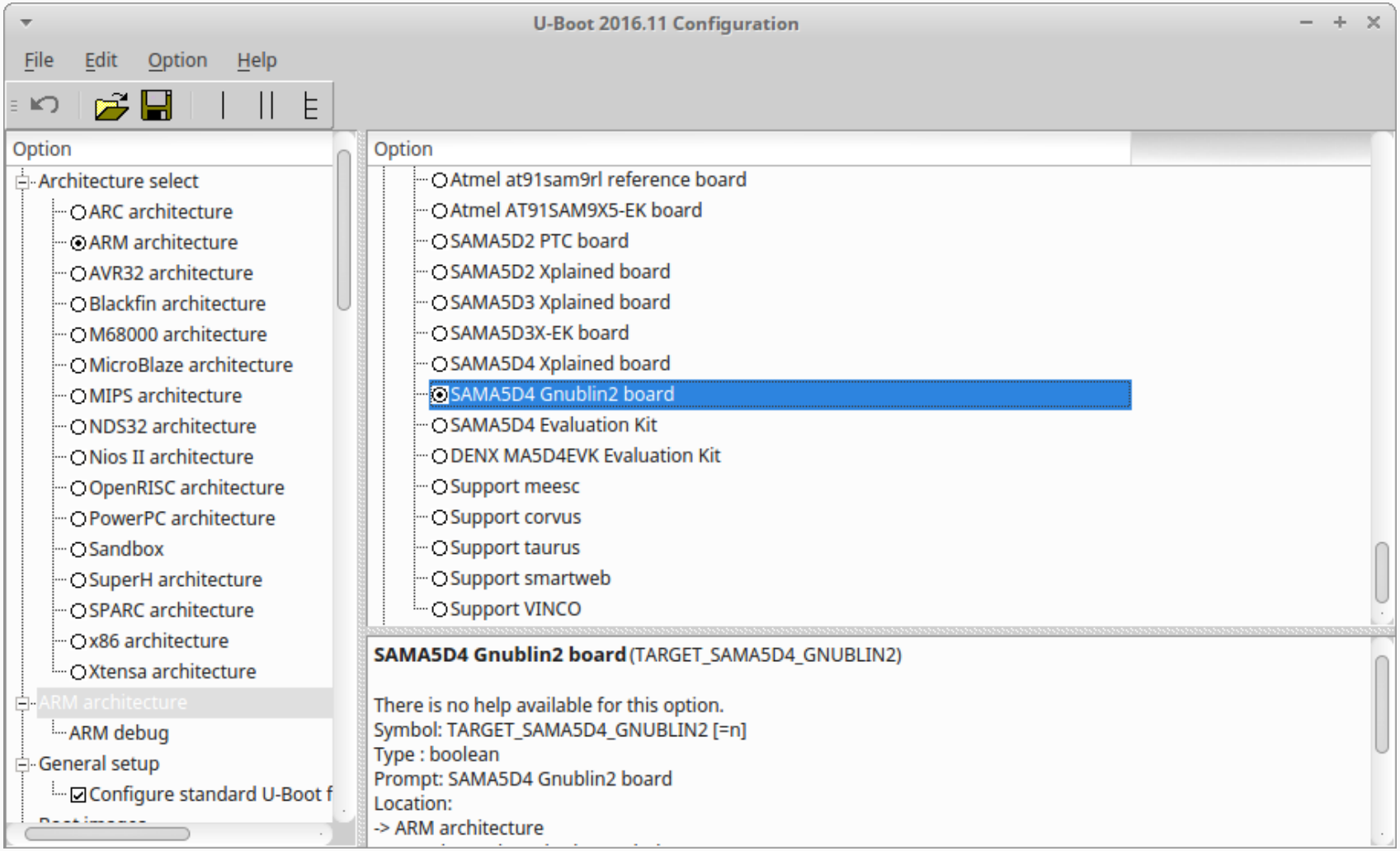
- Ähnliche Struktur wie Linuxkernel

U-Boot konfigurieren & kompilieren

- cross-compiler im Environment hinterlegen:
  export CROSS_COMPILE=arm-linux-gnueabihf-
  (hier: 32Bit ARM)
- Board-config laden
- z.B. **make odroid-xu3_defconfig**
- danach Konfiguration überarbeiten mit
- **make xconfig** oder **make menuconfig**
- make

- **Installation von U-Boot hängt vom Board ab**
  Beispiel: boot.bin in erster FAT-Partition der SD-Karte
- U-Boot SPL auch immer abhängig vom Board
  $\rightarrow$ Doku konsultieren

xconfig



xconfig wie beim Linux Kernel

U-Boot Environment

- U-Boot Konfiguration lässt sich zur Laufzeit teilweise verändern
- z.B. verschiedene Ethernet-Addresse, Boot-Medien, Update-Prozedur, etc.
- geschieht über **Environment**
- kann in einem nichtflüchtigen Speicher abgelegt werden (Flash, EEPROM)
- lesen/schreiben des Environments von U-Boot aus möglich
- default-Environment wird in U-Boot mit einkompiliert
- lässt sich über Board-configfiles anpassen
- fallback wenn anderes Environment nicht valide ist
- Anzeige mit **printenv**

U-Boot - Environment mit Scripting

```
=> printenv
arch=arm
baudrate=1500000
board=rockpro64_rk3399
board_name=rockpro64_rk3399
boot_a_script=load ${devtype} ${devnum}:${distro_bootpart} ${scriptaddr} ${prefi
x}${script}; source ${scriptaddr}
boot_efi_binary=load ${devtype} ${devnum}:${distro_bootpart} ${kernel_addr_r} ef
i/boot/bootaa64.efi; if fdt addr ${fdt_addr_r}; then bootefi ${kernel_addr_r} ${
fdt_addr_r};else bootefi ${kernel_addr_r} ${fdtcontroladdr};fi
boot_extlinux=sysboot ${devtype} ${devnum}:${distro_bootpart} any ${scriptaddr}
${prefix}extlinux/extlinux.conf
boot_net_usb_start=usb start
boot_prefixes=/ /boot/
boot_script_dhcp=boot.scr.uimg
boot_scripts=boot.scr.uimg boot.scr
boot_targets=mmc0 mmc1 usb0 pxe dhcp
bootcmd=run distro_bootcmd
bootcmd_dhcp=run boot_net_usb_start; if dhcp ${scriptaddr} ${boot_script_dhcp};
then source ${scriptaddr}; fi;setenv efi_fdtfile ${fdtfile}; setenv efi_old_vci
${bootp_vci};setenv efi_old_arch ${bootp_arch};setenv bootp_vci PXEClient:Arch:0
0011:UNDI:003000;setenv bootp_arch 0xb;if dhcp ${kernel_addr_r}; then tftpboot $
{fdt_addr_r} dtb/${efi_fdtfile};if fdt addr ${fdt_addr_r}; then bootefi ${kernel
_addr_r} ${fdt_addr_r}; else bootefi ${kernel_addr_r} ${fdtcontroladdr};fi;fi;se
tenv bootp_vci ${efi_old_vci};setenv bootp_arch ${efi_old_arch};setenv efi_fdtfi
le;setenv efi_old_arch;setenv efi_old_vci;
```

U-Boot Environment Einträge

- **bootcmd**: Boot-Kommando
- **dtb_name**: devicetree Dateiname
- **bootargs**: Boot-Argumente für Linuxkernel

U-Boot Kommandos

- Hilfe: **help**
- Kommandos für verschiedene subsysteme (z.B. MMC, USB)
- **boot**-Kommando führt **bootcmd**-Eintrag aus environment aus
- kernel kann auch manuell von medium ins RAM geladen werden
- ausführen des kernels mit **bootm/bootz**
- hilfreich z.B. bei unbeschriebenem integriertem flash
- **fatload**: Datei aus FAT-Dateisystem ins RAM laden (z.B. Kernel)
- auch Laden eines Kernels über UART oder USB möglich
- U-Boot kann benutzt werden um Flash-Speicher initial zu befüllen

Netzwerkunterstützung in U-Boot

- U-Boot kann Kernel via Netzwerk laden und booten
- Kommandos: **dhcp**, **bootp**, **tftpboot**
- zum Entwicklen nützlich
- nutzt BOOTP+TFTP
- BOOTP: Vorgänger von DHCP - automatische IP-Konfiguration
- auch Übergabe von Pfad für Boot-Image möglich
- Datei wird dann via TFTP geladen
- TFTP: Trivial File Transfer Protocol

Linux Kernel

Linux Kernel

- verschiedene trees verfügbar: `https://www.kernel.org/`
- mainline kann als Paket ohne git geladen werden
- Übersicht über weiteres trees: `https://git.kernel.org/cgit/`

- **arm-soc** tree für ARM-basierte SoCs
- `https://git.kernel.org/cgit/linux/kernel/git/arm/arm-soc.git/`
- sammelt Änderungen für ARM SoCs vor Übernahme in mainline kernel
- aktueller, teilweise Unterstützung für neue features, neue Treiber

- herstellerspezifische repositories sind häufig veraltet

32 Bit Kernel

- **export CROSS_COMPILE=arm-linux-gnueabihf-**
- zusätzlich **export ARCH=arm**

- Konfigurationsvorlagen liegen in **arch/arm/configs/**
- initiale Konfiguration mit **make exynos_defconfig** für Exynos-Boards (z.B. Odroid XU4)
- generische Konfiguration für alle Boards mit diesem SoC

- danach kernel-config mit **make xconfig** oder **make menuconfig**
- Konfiguration wird in **.config** Datei abgespeichert
- Konfiguration von altem Kernel kann mit **make oldconfig** übernommen werden

- kernel kompilieren mit **make -j4 zImage** für U-Boot
- andere Bootloader benötigen ggf. anderen Imagetyp

64 Bit Kernel

- **export CROSS_COMPILE=aarch64-linux-gnu-**
- zusätzlich **export ARCH=arm64**

- Konfigurationsvorlagen liegen in **arch/arm/configs/**
- initiale Konfiguration mit **make defconfig**
- derzeit nur eine Konfiguration für alle ARM64 SoCs

- xconfig, menuconfig, oldconfig usw. kein Unterschied

- kernel kompilieren mit **make -j4 Image** (kein zImage)

Linux Kernelmodule

- Kompilieren der Module mit **make -j4 modules**
- zum Installieren der Module Hilfsverzeichnis anlegen
- z.B. **mkdir ../module**
- dann **make INSTALL_MOD_PATH=../module modules_install**
- installiert module in Hilfsverzeichnis
- Module müssen später ins rootfs kopiert werden

Linux Kernel mit serieller Konsole

```
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd034]
[    0.000000] Linux version 5.1.0-rc2 (hunz@streator) (gcc version 8.2.0 (Ubunt
u 8.2.0-7ubuntu1)) #1 SMP Mon Mar 25 10:00:24 CET 2019
[    0.000000] Machine model: Radxa ROCK Pi 4
[    0.000000] printk: debug: skip boot console de-registration.
[    0.000000] earlycon: uart8250 at MMIO32 0x00000000ff1a0000 (options '')
[    0.000000] printk: bootconsole [uart8250] enabled
[    0.000000] cma: Reserved 32 MiB at 0x00000000f6000000
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv1.0 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: MIGRATE_INFO_TYPE not supported.
[    0.000000] psci: SMC Calling Convention v1.0
[    0.000000] random: get_random_bytes called from start_kernel+0x9c/0x3d8 with
 crng_init=0
[    0.000000] percpu: Embedded 22 pages/cpu @(____ptrval____) s50776 r8192 d311
44 u90112
[    0.000000] Detected VIPT I-cache on CPU0
[    0.000000] CPU features: detected: ARM erratum 845719
[    0.000000] CPU features: detected: GIC system register CPU interface
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 999432
[    0.000000] Kernel command line: earlyprintk rw init=/sbin/init root=PARTUUID
=b921b045-1d rootwait rootfstype=ext4 keep_bootcon earlycon=uart8250,mmio32,0xff
1a0000 swiotlb=1
```

Linux bootargs / cmdline

- Liste in **admin-guide/kernel-parameters.txt**
- drei Quellen: Kernel config, Bootloader, devicetree

Beispiel:
earlyprintk rw init=/sbin/init root=PARTUUID=b921b045-1d rootwait rootfstype=ext4 keep_bootcon

- **earlyprintk** und **earlycon**: Ausgaben auf serielle Konsole möglichst früh - Hardware-Angaben nötig
- **keep_bootcon**: Konsole immer aktiv lassen
- **root**: Pfad zum root-Dateisystem, rw: schreibbar mounten
- **rootwait**: warten auf root-Dateisystem
- **init**=/bin/sh kann für root-shell im Userland genutzt werden

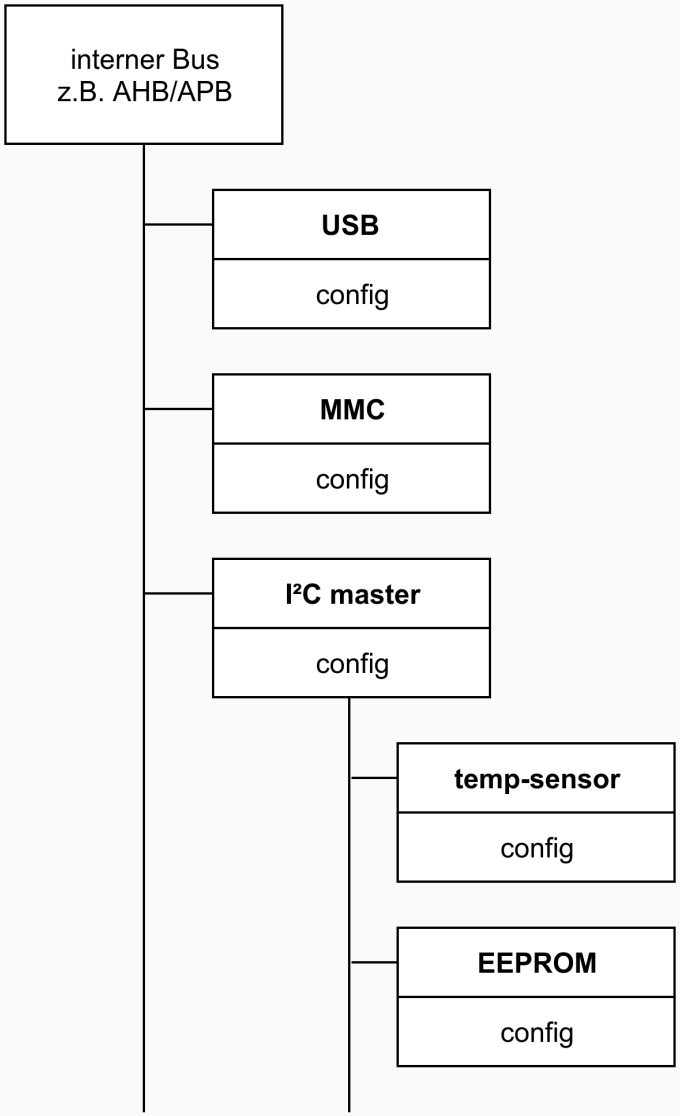
Devicetree

Devicetree

- wird dem Kernel vom Bootloader bereitgestellt
- Kernel-Konfiguration beinhaltet keine Informationen über Peripherie
- Kernel daher prinzipiell auf verschiedenen Boards lauffähig
- **devicetree** enthält Informationen über Peripherie
- Baumstruktur mit mehreren Ebenen
- Enthält Informationen über Busse, Peripherie, Speicheraddressen für Peripherie, IO-Konfiguration, etc.
- Kernel initialisiert nur solche Hardware die im device-tree korrekt referenziert ist

- **wichtig: Devicetree muss zur Kernelversion passen**
- Devicetree von älterem Kernel funktioniert oft nicht mit neuem Kernel

Devicetree Schema



Devicetree (2)

- Entwickler editiert ASCII .dts Datei
- dts-Datei kann weitere Dateien einbinden
  $\rightarrow$ Basisdatei für SoC - wird von Board-Dateien referenziert
- dts-Datei(en) werden vom device-tree-compiler übersetzt
- binärer devicetree (.dtb) wird am Zielsystem installiert
- Bootloader (U-Boot) lädt den tree und übergibt ihn an den Kernel
- bindings legen Zuordnung von devicetree-Elementen zu Treibern fest
- bindings ändern sich immer wieder
  $\rightarrow$ device-tree muss gepflegt werden
- immer den zur Kernelversion passenden devicetree benutzen
- falscher devicetree kann Hardware beschädigen (IO-Konfiguration)

Devicetree Beispiel

- Beispiel

Devicetree kompilieren

- Devicetrees werden mit **make dtbs** kompiliert
- devicetrees liegen in **arch/arm(64)/boot/dts/**
- einzelnen devicetree bauen mit **make name.dtb**
- Beispiel: **make rockchip/rk3399-rock-pi-4.dtb**

- auch decompilen möglich:
  dtc arch/arm64/boot/dts/rockchip/rk3399-rock-pi-4.dtb
- Symbolnamen werden im dtb aber durch Werte ersetzt

Devicetree bearbeiten

- bindings für device-tree legt Treiberautor fest
- Dokumentation in **Documentation/devicetree/bindings/**
- sortiert nach Hardwarekategorien
- teilweise sehr komplex, z.B. verschiedene Optionen für Peripherie

wichtig:

- die meiste Peripherie ist im SoC includefile (.dtsi) definiert
- aber deaktiviert, da nicht auf allen Boards genutzt
- aktivieren mit status = "okay";

Devicetree bearbeiten (2)

- klassisches Problem: wie binde ich IC xyz ein?
- Dokumentation aus bindings lesen
- Andere devicetrees als Vorlage benutzen
- Beispiel: I2C-Bus mit LM75 Sensor
- LM75-Sensor soll via Kernel ausgelesen werden
- im dts-Verzeichnis: **grep -i lm75**:
  lm75@48 {
  ...
- Treiber erkennt über **compatible** Liste geeignete Peripherie

Linux Userland

Debian für ARM

- nutzt Qemu um Distribution mit einer virtuellen ARM-Maschine zu erzeugen
- benötigte Pakete: qemu-user-static debootstrap
- Zielverzeichnis anlegen: **mkdir chroots**
- Pakete laden: **sudo qemu-debootstrap --arch=armhf stretch chroots/stretch-armhf**
- **armhf**: ARM SoC mit Floatingpoint Unit (z.B. SAMA5D4x)
- **stretch**: Version der Debian-Distribution
- Übersicht: `https://wiki.debian.org/ArmPorts`
- Anleitung: `https://wiki.debian.org/ArmHardFloatChroot`
- ARM64: `https://wiki.debian.org/Arm64Port#Debootstrap_arm64`
- Dauer: ca. 10 Minuten
- Platzbedarf: ca. 260 MB

Nutzung des frischen Systems

- **sudo chroot chroots/stretch-armhf/** wechselt in virtuelle ARM-Maschine mit neuem System
- manuelles mounten des proc-Dateisystems: **mount -t proc proc /proc**
- danach normale Nutzung möglich
- z.B. Installation von zusätzlichen Paketen

Fazit

Fazit

- einigermaßen komplex

- Bootloader selbst bauen meist nicht nötig
- Ausnahme z.B. custom bootlogo
- Userland meist auch nur einmal aufsetzen, dann updaten

- frischer Kernel bringt oft neue features, z.B. mehr Performance, bessere Stromspartechniken
- security-features

- serielle Konsole hilft beim debuggen

Fragen?

Fragen?