```
$ bitbake mein-os
```

Wie baut man ein Betriebssystem?



Embedded Linux mit Yocto – mit nur leichten Verbrennungen

Doralitze

Nook 2026 • 21. November 2026

KAPITEL 01

Einleitung

Oder: Wie ich lernte, die Mikrowelle zu lieben

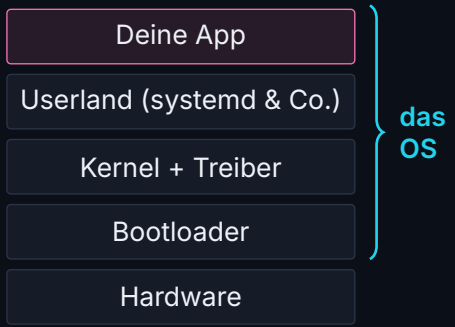
01

Was ist ein (eingebettetes) Betriebssystem?

Der **komplette Software-Stapel**, der ein Gerät zum Laufen bringt.

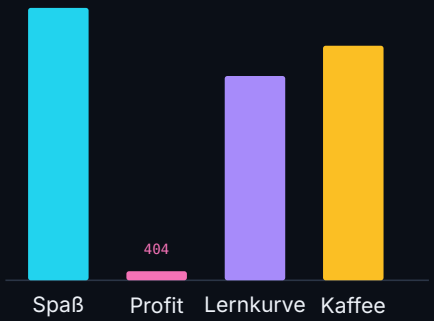
- Fernseher
- moderne Mikrowelle
- Router, 3D-Drucker, Waschmaschine …

Nutzer sehen nur die App. Wir sehen *alles* darunter.



(eine Platine, die du nicht selbst gelötet hast)

Warum sollte man so etwas selbst bauen?



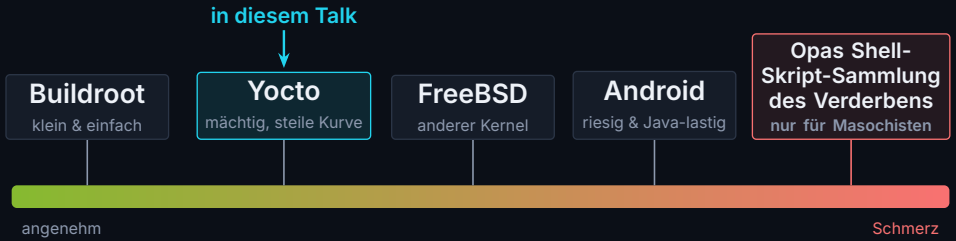
(nicht maßstabsgetreu, aber ehrlich)

For Fun.
Not Profit.

Netter Nebeneffekt: sehr nützlich, wenn man ein Gerät baut, das ein OS *braucht*.

Und man versteht endlich, was der eigene Laptop beim Booten eigentlich so treibt.

Die Werkzeugkiste



Alle Beispiele in diesem Talk: **Linux + Yocto** (Releases *Walnascar* → *Wrynose*).

KAPITEL 02

Voraussetzungen

Das Kapitel, in dem es teuer wird

Der Build-Rechner

CPU schnelle Kerne > viele Kerne – aber mehr Kerne sind viel besser

RAM ≥ 2 GB pro Kern

SSD ≥ 500 GB, bitte SSD

Du baust das komplette Image *sehr* oft neu.

FAUSTREGEL

16 Kerne ⇒ 32 GB RAM. Mehr RAM = weniger OOM-Killer-Überraschungen beim Linken von Chromium & Co.



dein Hardware-Budget

Bei aktuellen Hardwarepreisen heißt das:

Sei reich.

⚠ GOTCHA

Debian 13+: AppArmor blockiert unprivilegierte User-Namespaces – Bitbake braucht sie für seine Sandboxes.

Fix: `sysctl kernel.apparmor_restrict_unprivileged_userns=0`

Bitte keine Nvidia-GPU



- Out-of-tree-Treiber + Kernel-Updates = **Drama**
- GPU-Beschleunigung in QEMU (`virtio-vga-gl`) ist eine Glückssache
- Du willst Images debuggen, nicht Treiber

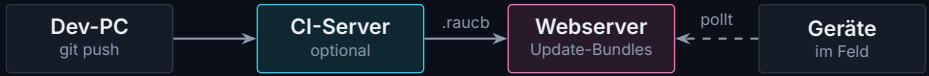
⚠ GOTCHA

`virtio-vga-gl` rendert nicht auf jedem Host. Notlösung: Software-Rendering in sway. Auf Intel/AMD-Hosts den Override wieder rausnehmen!

Fix: `Environment=WLR_RENDERER=pixman`

sonst: `-vga none -device virtio-vga -display gtk,show-cursor=on`

Infrastruktur: CI-Server & Webserver



⚠ GOTCHA: CI-RUNNER STIRBT

Kalter Build mit unbegrenztem `PARALLEL_MAKE` ⇒ 16 Linker gleichzeitig ⇒ OOM & `mmap`-Segfault.

Fix: `PARALLEL_MAKE = "-j 4"`
`BB_NUMBER_THREADS = "4"`

⚠ GOTCHA: 6 STUNDEN SIND NICHTS

Default-Timeout bei GitHub Actions: 6 h. Ein kalter Build braucht… mehr. Fetch-Hänger killen den ganzen Lauf.

Fix: `timeout-minutes: 4320`
`make fetch` separat, mit 3 Retries

CI: Wenn der Build länger dauert als dein Wochenende



make fetch mit Retry

```
for i in 1 2 3; do
    make fetch && break
    sleep 30
done
```

⚠ GOTCHA: FETCH-STEMPEL LÜGEN

Ein alter `do_fetch`-Stempel sagt „erledigt" – der Tarball ist aber längst gelöscht. `do_unpack` kracht.

Fix: `bitbake --force --runall=fetch mein-image`

⚠ GOTCHA: GIT WARTET AUF DICH

Bei einem GitHub-Schluckauf fragt git nach Zugangsdaten – und wartet ewig.

Fix: `export GIT_TERMINAL_PROMPT=0`

⚠ GOTCHA: DL_DIR WÄCHST EWIG

Tarballs mit geänderter Prüfsumme bleiben für immer liegen. Ein wöchentlicher Wartungsjob leert `DL_DIR` – `make clean` lässt es bewusst in Ruhe.

KAPITEL 03

Die Build-Umgebung

Ordnung ist das halbe Image

Yocto in 60 Sekunden



Rezept = Anleitung für *ein* Paket. **Layer** = Sammlung von Rezepten. **Image** = alles zusammen auf eine Disk.

Anatomie eines Rezepts

recipes-mein/tool/tool_1.2.bb

```
SUMMARY = "Mein kleines Tool"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://LICENSE;md5=0835ade6"

SRC_URI = "git://github.com/ich/tool.git;\
protocol=https;branch=main"
SRCREV  = "3f2a9c1e..."

inherit autotools pkgconfig

DEPENDS = "libusb1"            # zum Bauen
RDEPENDS:${PN} = "bash"        # zur Laufzeit
```

ABKÜRZUNG

`devtool add <url>` erzeugt das Grundgerüst.

- **Was?** Name und Lizenz
- **Woher?** `SRC_URI` + fester Commit
- **Wie bauen?** per `inherit` einer Klasse
- **Was braucht's?** `DEPENDS` (Build) vs. `RDEPENDS` (Laufzeit)

⚠ GOTCHA: KEINE LIZENZ

Rezept ohne `LICENSE` ⇒ kryptischer Parse-Fehler. Auch beim winzigsten Skript-Rezept!

Fix: `LICENSE + LIC_FILES_CHKSUM`

Layer als Git-Submodule

```
mein-os/
├── poky
├── meta-openembedded
├── meta-intel          ← Submodule
├── meta-rauc
├── meta-wayland
├── meta-custom         ← dein Kram
├── config
└── Makefile
```

bash

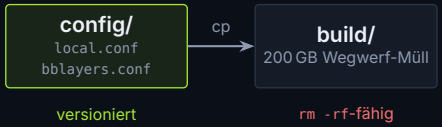
```
$ git submodule add -b walnascar \
    https://git.yoctoproject.org/poky layers/poky
$ git submodule add -b walnascar \
    https://github.com/rauc/meta-rauc \
    layers/meta-rauc
# ... meta-openembedded, meta-intel, meta-wayland

# Frisch geklont? Alles auf den gepinnten Stand:
$ git submodule update --init --recursive
```

WARUM?

Jeder Commit pinnt *exakt* die Layer-Versionen. Builds sind reproduzierbar – auch in zwei Jahren.

Konfiguration getrennt halten & das Makefile



Anders als die offizielle Doku: config **nicht** in `build/conf` pflegen.

Makefile = externes Gedächtnis. Du weißt in drei Monaten nicht mehr, welche zwölf Flags QEMU brauchte. Versprochen.

Makefile

```
BUILD ?= build
ENV    = . layers/poky/oe-init-build-env $(BUILD)

conf:
    mkdir -p $(BUILD)/conf
    cp config/*.conf $(BUILD)/conf/
build: conf
    $(ENV) && bitbake mein-image
fetch: conf
    $(ENV) && bitbake --runall=fetch mein-image
run:          # QEMU, User-Netzwerk
    qemu-system-x86_64 $(QEMU_FLAGS) ...
deploy:
    ./scripts/publish-updates.py
```

Bitbake lügt (manchmal)

`bitbake: "Alles aktuell, nehme den Cache!"` — Erzähler: Es war nicht aktuell.

⚠ GOTCHA: TASKS PARALLEL

`do_image_ext4` und `do_image_wic` laufen parallel. WIC kopiert Slot A nach B, bevor es Slot A gibt ⇒ Slot B **immer noch leer**.

Fix:

```
do_image_wic[depends] +=
"${PN}:do_image_ext4"
```

⚠ GOTCHA: DATETIME

`${DATETIME}` in einer Task ⇒ neue Signatur *jede Sekunde* ⇒ der gesamte rootfs-Cache ist für die Tonne.

Fix:

```
do_x[vardepsexclude]
= "DATETIME"
```

Zeitstempel per Sidecar-Datei ins Image.

⚠ GOTCHA: UNGEHASHT

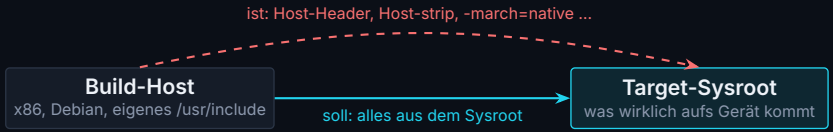
`do_image_wic` hasht die eingebettete `grub.cfg` nicht. Config geändert ⇒ altes WIC aus dem Cache.

Fix:

```
do_image_wic[file-
checksums] +=
"${GRUB_CFG}:True"
```

Merke: `IMAGE_TYPEDEP` ändert nur DEPENDS – nicht die Reihenfolge der Tasks.

Upstream-Makefiles vs. Cross-Compiling



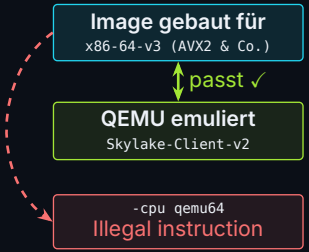
Upstream macht …	Fix im Rezept
`CFLAGS = …` überschreibt Yocto-Flags	`sed`: `CFLAGS =` → `CFLAGS +=`
`LDFLAGS` kommen nie beim Linker an	`$(LDFLAGS)` in `LFLAGS` einschleusen
`-march=native -mavx2`	`EXTRA_OEMAKE = "MARCH_ARGS=''"`
`-I/usr/local/include` (poison-system-dirs)	per `sed` entfernen
configure sucht im Host-`/usr/include`	`--with-slanginc=${STAGING_INCDIR}`
`install -s` nutzt den Host-`strip`	`-s` raus, Yocto strippt selbst
configure *startet* ein Target-Binary	diesen Check entfernen
alte pkg-config-Namen (`libusb`)	`libusb-1.0`, `libftdi1`

KAPITEL 04

Testen mit QEMU

Hardware ist teuer. Emulation ist geduldig.

Gleiche CPU wie im Image



local.conf + bash

```
DEFAULTTUNE = "x86-64-v3"       # im Image

$ qemu-system-x86_64 -machine q35 -accel kvm \
    -cpu Skylake-Client-v2 -smp 4 -m 4G \
    -drive if=pflash,readonly=on,file=$CODE \
    -drive if=pflash,file=$VARS \
    -drive file=mein-os.qcow2,if=virtio
```

⚠ GOTCHA: OVMF

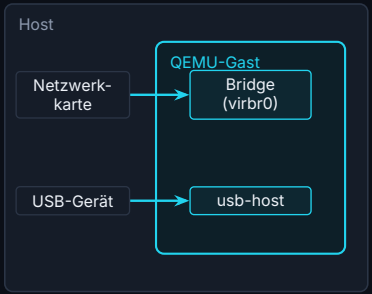
UEFI-Firmware heißt auf jeder Distro anders, und das NVRAM überlebt kaputte Läufe.

Fix: `make run OVMF_CODE=…`
`make reset-nvram`

WARUM?

Image-Flags und emulierte CPU müssen zusammenpassen. Zu alte CPU ⇒ Absturz beim ersten AVX2-Befehl. Zu neue ⇒ du testest etwas anderes als die echte Hardware.

Durchreichen statt emulieren



Echte Hardware im Gast ⇒ echte Treiber getestet. **virtio** statt emulierter Antiquitäten.

⚠ GOTCHA: BRIDGE

`-netdev bridge` braucht `virbr0` (libvirt) auf dem Host. Ohne libvirt: Usermode-Netz.

Fix: `make run (nutzt -netdev user)`

⚠ GOTCHA: WACKELIGE MAUS

`-device usb-tablet` ist unter KVM unzuverlässig.

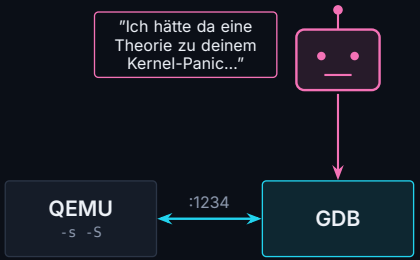
Fix:
```
-device virtio-tablet-pci
-device virtio-keyboard-pci
```

⚠ GOTCHA: DISK VOLL

Das WIC ist exakt 32 GiB – kein Platz für die Datenpartition beim ersten Boot.

Fix:
```
qemu-img create -f qcow2 -F raw
-b mein-os.wic overlay.qcow2 64G
```

Debuggen mit GDB (und deinem LLM)



bash

```
# QEMU stoppt und wartet auf GDB
$ make debug QEMU_EXTRA="-s -S"

$ gdb vmlinux \
    -ex "target remote :1234" \
    -ex "hbreak start_kernel" \
    -ex "continue"
```

RUBBER DUCK 2.0

Sag deinem LLM, dass es per GDB an QEMU hängen darf. Es liest Backtraces schneller als du – und wird nie müde.

KAPITEL 05

Release-Images

Ein Image für jeden Anlass

Drei Images pro Target

Test
`.wic`
Debug-Tools, SSH,
Root ohne Passwort
nur Labor & QEMU

Factory
`.wic`
komplette Disk:
GPT, ESP, Slot A+B
wird in der Fabrik geflasht

Upgrade
`.raucb`
nur ein Slot
signiert & verity-geschützt
kommt übers Netz

bash

```
$ bitbake mein-image-test mein-image-factory mein-bundle
```

Ein Test-Image in der Fabrik ist ein offenes Scheunentor mit Willkommensschild.

Partitionslayout: A/B + Daten



- Slots **read-only**: kein Zustand, kein Drift
- Kaputtes Update? Einfach den *anderen* Slot booten
- Alles Persönliche liegt auf **userdata**

⚠ GOTCHA: UUID VERBOTEN

Slot B ist eine Bit-Kopie von Slot A ⇒ *gleiche* Dateisystem-UUID. `root=UUID=…` ist mehrdeutig.

Fix: `root=PARTLABEL=rootfsA`

überall: cmdline, fstab, RAUC, GRUB

Das Disk-Layout als Text: `.wks`

files/wic/mein-os.wks.in

```
bootloader --ptable gpt --configfile="mein-grub.cfg"
part /boot/efi --source bootimg-efi --sourceparams="loader=grub-efi" \
               --fstype=vfat --label esp --size 256M
part /         --source rootfs  --fstype=ext4 --part-name rootfsA --size 8G
part           --source rawcopy --sourceparams="file=mein-image.ext4" \
               --part-name rootfsB --size 8G
# userdata fehlt absichtlich: entsteht beim ersten Boot
```

WARUM KEIN USERDATA?

Beim ersten Boot angelegt, nutzt sie den *gesamten* Rest der Disk – egal wie groß die ist.

⚠ GOTCHA: SUCHPFAD

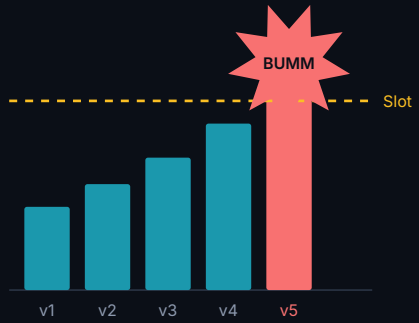
Ab Wrynose sucht wic in `files/wic/`, nicht mehr in `wic/`. Kein Fund, kein Image.

⚠ GOTCHA: ÜBERSCHRIEBEN

`WKS_FILE_DEPENDS_BOOTLOADERS` aus `local.conf` wird je Architektur überschrieben.

Fix: `do_image_wic[depends]` in eigener `bbclass`

Read-only heißt read-only (und Platz reservieren!)



A/B-Partitionen lassen sich per Update **nicht** vergrößern. **Großzügig** planen – das Image wächst. Immer.

⚠ GOTCHA: EROFS

Nach `switch_root` scheitert jedes `mkdir`. Mountpoints müssen schon im Image existieren.

Fix: `beim Image-Bau anlegen (Postprocess)`

⚠ GOTCHA: TMPFS FRISST /var

`volatile-binds` legt tmpfs über `/var/lib` & Co. und verdeckt die echten Daten.

Fix: `VOLATILE_BINDS = ""`

⚠ GOTCHA: CONFIG IN /etc

`/etc/motd` liegt im RO-Slot – Änderungen sind beim Update weg.

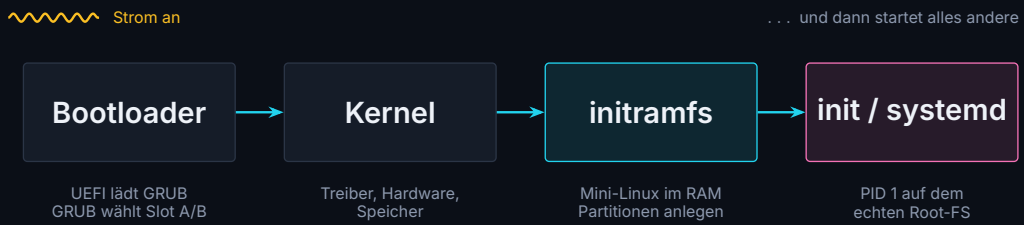
Fix: `ln -s /var/motd /etc/motd`

KAPITEL 06

Der Bootprozess

Vom Strom zum Prompt in vier Akten

Die Boot-Kette



Jede Stufe hat genau einen Job: **die nächste Stufe finden und starten**.

Wenn's hängt: Stufe bestimmen, dann debuggen. UART und `console=ttyS0` sind deine Freunde.

Den Kernel konfigurieren: Fragmente

Kernel-bbappend + Fragment

```
# linux-yocto_%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI += "file://rauc-verity.cfg"

# files/rauc-verity.cfg
CONFIG_MISC_FILESYSTEMS=y
CONFIG_SQUASHFS=y
CONFIG_DM_VERITY=y
```

Fragment erzeugen

```
$ bitbake -c menuconfig virtual/kernel
$ bitbake -c diffconfig virtual/kernel
# -> fragment.cfg mit genau deinen Änderungen
```

WARUM FRAGMENTE?

Nur das Delta zur Standard-config wird versioniert. Überlebt Kernel-Upgrades viel besser als eine komplette `.config`.

⚠ GOTCHA: SHEBANG GEHT NICHT

`#!/bin/sh`-Skripte starten nicht: *Exec format error*.

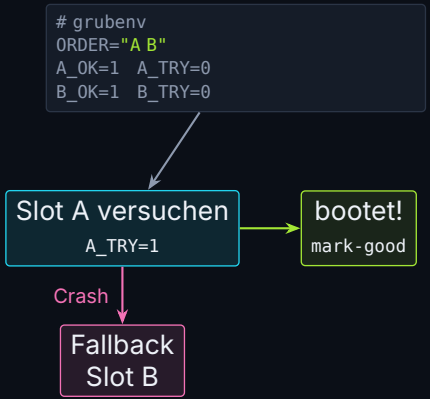
Fix: `CONFIG_BINFMT_SCRIPT=y`

⚠ GOTCHA: BUILDPATHS-QA

Der Intel-Xe-Treiber generiert einen Header mit Pfad vom Build-Host.

Fix: `INSANE_SKIP:${PN}-src += "buildpaths"`

GRUB & A/B: Wer bootet heute?



⚠ GOTCHA: KEIN --part-label

GRUB kann nicht nach Partitionsnamen suchen, `search --label` findet beide Slots.

Fix:
```
set root=(hd0,gpt2) # Index
linux … root=PARTLABEL=rootfsA
```

⚠ GOTCHA: FACTORY-GRUBENV

Fehlt eine Variable, rät GRUB beim ersten Boot – meistens falsch.

Fix:
```
grub-editenv grubenv create
grub-editenv grubenv set ORDER="A B" \
  A_OK=1 A_TRY=0 B_OK=1 B_TRY=0
```

Erster Boot: Was die initramfs erledigt



⚠ GOTCHA: OHNE INITRAMFS KEIN /home

Fehlte die initramfs, blieb `/home` read-only. Gemeldet nur als `bbwarn` im Build-Log.

Fix: `bbfatal + do_rootfs[depends] += "mein-initramfs:do_image_complete"`

⚠ GOTCHA: SEEDEN BEI JEDEM BOOT

`/var` und `/home` wurden bei jedem Start neu befüllt – und überschrieben fröhlich die Nutzerdaten.

Fix: `.seeded-Marker, cp -a nur einmal`

initramfs: Der allererste Boot

firstboot (gekürzt)

```
sgdisk -e "$disk"     # GPT-Backup ans Ende
sgdisk -n 0:0:0 -c 0:userdata "$disk"
partprobe "$disk"; udevadm settle
mkfs.ext4 -L userdata "$part"
# /var und /home nur EINMAL seeden
[ -e "$ud/.seeded" ] || seed_var_home
mount --move /mnt/userdata \
      "$ROOTFS_DIR/mnt/userdata"
```

WARUM sgdisk -e?

Image auf eine größere Disk geflasht? Dann liegt der GPT-Backup-Header mitten im Nirgendwo und `sgdisk` sieht den freien Platz nicht.

⚠ GOTCHA: DATEINAME = API

`initramfs-framework` leitet Callback-Namen aus dem Dateinamen ab: `95-app-firstboot` ruft `app_run`, nicht `firstboot_run`.

Fix: `Datei 95-firstboot nennen`

⚠ GOTCHA: UNSICHTBARE PARTITION

`sgdisk` legt an, der Kernel merkt nichts ⇒ kein `by-partlabel`-Link.

Fix: `partprobe; udevadm settle`

⚠ GOTCHA: SWITCH_ROOT VERGISST

Mounts aus der initramfs folgen `switch_root` nicht.

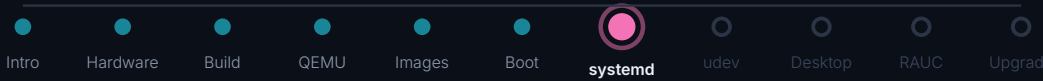
Fix: `mount --move vor switch_root`

KAPITEL 07

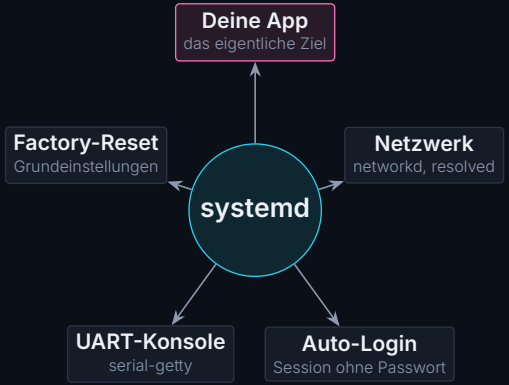
systemd & Userland

PID 1 und seine Freunde

07



Was alles laufen muss



⚠ GOTCHA: KEIN SYSTEMD

`INIT_MANAGER` allein reichte nicht – systemd landete gar nicht im rootfs.

local.conf

```
INIT_MANAGER = "systemd"
DISTRO_FEATURES:append = " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
VIRTUAL-RUNTIME_initscripts = ""
```

Netzwerk & der Notausgang

⚠ GOTCHA: KEIN NETZ

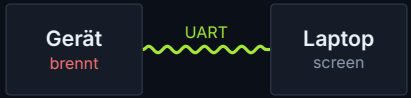
`systemd-networkd` & `-resolved` sind per Default **aus**. Erster Boot: offline.

Fix: `Preset-Datei ausliefern`

20-wired.network + Preset

```
[Match]
Name=en*
[Network]
DHCP=yes
# 80-mein-os.preset
enable systemd-networkd.service
enable systemd-resolved.service
```

Plus in `local.conf`: `RESOLV_CONF = "stub-resolv"`



Wenn *alles* brennt, ist die serielle Konsole dein Feuerlöscher.

machine.conf + Host

```
SERIAL_CONSOLES = "115200;ttyS0"
# Kernel-Cmdline: console=ttyS0,115200
$ screen /dev/ttyUSB0 115200
```

Factory-Reset & Grundeinstellungen



Read-only-Slots machen den Reset trivial: **userdata löschen, neu starten**. Der Rest ist ja unveränderlich.

`/etc/profile.d/mein-tz.sh`

```
TZ_FILE=/var/lib/mein-os/timezone
export TZ="$(cat $TZ_FILE || echo UTC)"
```

⚠ GOTCHA: ZEITZONE AUF RO

`timedatectl set-timezone` will `/etc/localtime` schreiben – geht nicht.

Fix: `Zone auf userdata speichern, TZ per profile.d exportieren`

⚠ GOTCHA: ZU FRÜH, TEIL 3

`user-init.service` schrieb die MOTD, bevor `/var` gemountet war.

Fix: `After=multi-user.target`

Reihenfolge ist alles

mein-app.service

```
[Unit]
Description=Die eigentliche Anwendung
After=network-online.target \
      time-sync.target sway.service
Wants=network-online.target \
      time-sync.target

[Service]
User=user
ExecStart=/opt/mein-app/bin/app
Restart=on-failure
Nice=10

[Install]
WantedBy=graphical.target
```

⚠ GOTCHA: ZU FRÜH, TEIL 1

Update-Check vor NTP-Sync ⇒ Server-Zertifikat ist *"noch nicht gültig"* (Systemuhr: 1970).

Fix: `Wants=time-sync.target`

⚠ GOTCHA: ZU FRÜH, TEIL 2

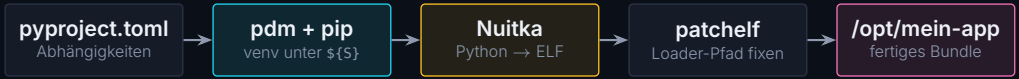
Boot-Check lief, bevor `/boot/efi` gemountet war. `grub-editenv` scheiterte – **heimlich**.

Fix: `ConditionPathExists=/boot/efi/EFI/BOOT/grubenv`

MERKE

`After=` ordnet nur. `Wants=` fordert sie auch an. Du brauchst fast immer *beides*.

Deine App ins Image: Python, PySide6 & Nuitka



⚠ GOTCHA: PDM BAUT HEIMLICH VENVS

`pdm` ≥ 2.29 baut bei `pdm export` eine eigene venv – ohne Yoctos Bibliothekspfade. Ergebnis: opaker `CalledProcessError`.

Fix: `pdm<2.28` `PDM_PYTHON_USE_VENV=false`

⚠ GOTCHA: 1 GB WHEELS

PySide6-Wheels werden erst im `do_compile` geladen. Yocto erlaubt Netzwerk aber nur in `do_fetch`.

Fix: `do_compile[network]="1"`

⚠ GOTCHA: FALSCHER LOADER

Nuitka schreibt `/lib64/ld-linux-x86-64.so.2` ins ELF. Mit `usrmerge` gibt's nur `/usr/lib/…`

Fix: `patchelf` setzt den Interpreter jeder ELF-Datei neu

Python-Apps: Überraschungen zur Laufzeit

⚠ GOTCHA: CTYPES

`pysdl2` lädt `libsdl2` per `ctypes` – kein DT_NEEDED, Yocto sieht nichts.

Fix: `RDEPENDS:${PN} += "libsdl2"`

⚠ GOTCHA: TYPELIB FEHLT

Namespace Poppler not available: `.typelib` liegt in `poppler`, nicht `libpoppler-glib`.

Fix: `RDEPENDS auf das Hauptpaket`

⚠ GOTCHA: LIBGL

`opencv-python` will `libGL` schon beim Nuitka-Build.

Fix: `opencv-python-headless`

⚠ GOTCHA: KRYPTISCHE SONAMES

manylinux-Wheels bringen `libcrypto-5409cd36.so.1.1.1k` mit – RPM erfindet daraus unerfüllbare Requires.

Fix: `SKIP_FILEDEPS:${PN} = "1"`

⚠ GOTCHA: QA SCHLÄGT FALSCH AN

Ein eigenständiges `/opt`-Bundle verletzt strip-, rpath- und textrel-Checks.

Fix: `INSANE_SKIP + INHIBIT_PACKAGE_STRIP`

⚠ GOTCHA: SYMLINK INS LEERE

Der Icon-Pfad zeigt in ein Submodul. `git://` holt keine Submodule.

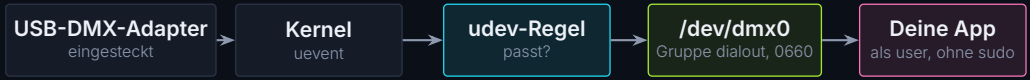
Fix: `SRC_URI = "gitsm://…"`

KAPITEL 08

udev & polkit

Wer darf was? (Spoiler: nicht alles als root)

udev: Rechte automatisch verteilen



/etc/udev/rules.d/60-dmx.rules

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", \
  ATTRS{idProduct}=="6001", \
  GROUP="dialout", MODE="0660", SYMLINK+="
  ↪ dmx0"
```

⚠ GOTCHA: AUTOMOUNTER

Der udev-Automounter mountete den *inaktiven* Slot – und blockierte damit `rauc install`.

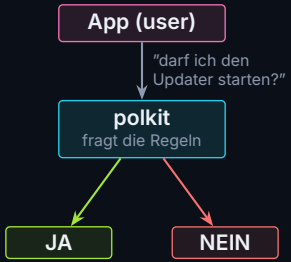
Fix: `ENV{ID_PART_ENTRY_NAME}== "rootfs[AB]"` überspringen

⚠ GOTCHA: USB-STICK NUR FÜR ROOT

vfat/exfat kennen keine Rechte.

Fix: `mount -o uid=1000,gid=1000 …`

polkit: kontrollierte Rechteausweitung



Kein `sudo`, keine Passwörter auf dem Gerät – **genau eine** erlaubte Aktion.

`/usr/share/polkit-1/rules.d/50-updater.rules`

```
polkit.addRule(function(action, subject) {
  if (action.id ==
      "org.freedesktop.systemd1.manage-units" &&
    action.lookup("unit") == "updater.service" &&
    subject.isInGroup("updater"))
    return polkit.Result.YES;
});
```

⚠ GOTCHA: QA-CHECK

Regeln unter `/etc/polkit-1/rules.d/` scheitern am Ownership-QA-Check (Verzeichnis gehört polkitd, `0700`).

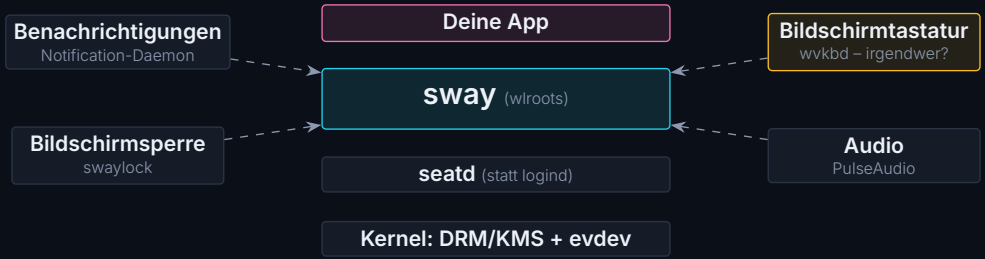
Fix: `/usr/share/polkit-1/rules.d/ nutzen`

KAPITEL 09

Desktop-Umgebung

Jetzt wird’s bunt

Der Desktop-Baukasten



Ein Touch-Gerät ohne Tastatur und ohne Bildschirmtastatur ist ein sehr teurer Bilderrahmen.

sway ohne logind

sway.service (Auszug)

```
[Service]
User=user
Environment=XDG_RUNTIME_DIR=/run/sway
WorkingDirectory=/home/user
TTYPath=/dev/tty1
ExecStart=/bin/sh -c \
    'trap "" INT QUIT; exec sway'
```

WARUM SEATD?

Ohne logind darf sway weder DRM noch Input öffnen. `seatd` läuft als root und reicht die File-Deskriptoren durch.

⚠ GOTCHA: CTRL-C = DESKTOP WEG

Strg+C auf tty1 killte die ganze Prozessgruppe – inklusive sway.

Fix: `trap "" INT QUIT vor exec`

⚠ GOTCHA: SCHWARZER BILDSCHIRM

QEMU: virtio-gpu fehlt im Kernel, wlroots ohne gbm.

Fix: `CONFIG_DRM_VIRTIO_GPU=y`
`PACKAGECONFIG:append = " gbm"`

⚠ GOTCHA: SPLASH-KACHEL

Der Qt-Splash landet als Kachel im Layout.

Fix: `for_window [app_id="editor"] floating enable`

Ton & Tastatur



⚠ GOTCHA: KEIN TON

Per-User-PulseAudio braucht eine logind-Session. Haben wir nicht.

Fix: `PulseAudio im System-Modus, Before=sway.service`

client.conf.d/50-system-server.conf

```
default-server=unix:/var/run/pulse/native
autospawn=no
```



⚠ GOTCHA: TASTATUR KOMMT NICHT

`wvkbd` erscheint nicht automatisch bei Textfokus.

Fix: `wvkbd --auto`

\+ kleiner Manager, der `--auto` abschaltet, sobald (per udev) eine echte Tastatur steckt.

osk-manager.py (Idee)

```
# pyudev: echte Tastatur? -> kein --auto
for dev in iter(mon.poll, None):
  if dev.get("ID_INPUT_KEYBOARD"):
    osk(auto=dev.action != "add")
```

Benachrichtigungen, Sperre & Feinschliff

/etc/sway/config.d/50-session.conf

```
exec mako                    # Benachrichtigungen
exec swayidle -w \
    timeout 300 'swaylock -f -c 000000' \
    timeout 600 'swaymsg "output * power off"' \
    resume 'swaymsg "output * power on"'
exec generate_wallpaper.py # passt zur Auflösung
```

Test

```
$ notify-send "Update bereit" "Jetzt neu starten?"
```

TOUCH STATT TASTATUR

Eigene Keybindings als Drop-in: Tabs statt Kacheln, Fenster schließen mit einem Griff.

⚠ GOTCHA: TERMINAL IN /

`foot` öffnete sich im Wurzelverzeichnis statt in `$HOME`.

Fix: `WorkingDirectory=/home/user`

⚠ GOTCHA: NAMENLOSE SCHRIFT

Ohne config fällt foot auf irgendein Monospace zurück.

Fix: `font=Hack:size=11` in `/etc/xdg/foot/foot.ini`

⚠ GOTCHA: VERZERRTES HINTERGRUNDBILD

Nach Drehen oder Andocken passte das Bild nicht mehr.

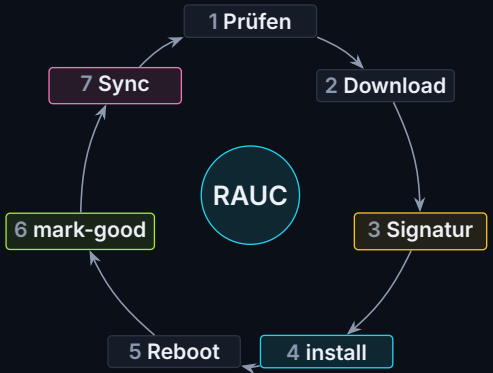
Fix: `beim Start generieren`

KAPITEL 10

Updates mit RAUC

Updates, ohne das Gerät zu bricken (meistens)

Der Update-Zyklus



1. Daemon prüft regelmäßig
2. Download – **robust**!
3. Signatur gegen den keyring
4. `rauc install` → *anderer* Slot
5. Reboot in den neuen Slot
6. Boot als gut markieren
7. alten Slot nachziehen

⚠ GOTCHA: FUNKLOCH

Ein HTTPS-Schluckauf darf nicht 800 MB neu laden: Resume, Retry, Backoff.

Ein Downloader, der Funklöcher überlebt

mein-updater.py (Skizze)

```
def download(url, ziel):
  for versuch in range(10):
    da = groesse_oder_0(ziel)       # Resume!
    try:
      r = requests.get(url, stream=True,
            timeout=(10, 60),
            headers={"Range": f"bytes={da}-"})
      r.raise_for_status()
      with ziel.open("ab") as f:
        for block in r.iter_content(1 << 20):
          f.write(block)
      return pruefe_sha256(ziel)
    except requests.RequestException:
      time.sleep(min(2 ** versuch, 300))
  raise DownloadFehler(url)
```

⚠ GOTCHA: TEXT = ZEIT

Versionen sind `YYYYMMDDhhmmss`-Strings: lexikalisch = chronologisch. Nie auf Epoch-Sekunden umstellen!

⚠ GOTCHA: DOPPELT VERÖFFENTLICHT

Bundle + Symlink darauf ⇒ doppelt hochgeladen.

Fix: `Path.resolve() + Duplikate raus`

⚠ GOTCHA: HARTKODIERTES HOME

Cache fest unter `/home/user/…`

Fix: `Path.home()`

Vertrauen ist gut, Signaturen sind besser

/etc/rauc/system.conf

```
[system]
compatible=mein-os-x86
bootloader=grub
bundle-formats=-plain

[keyring]
path=/etc/rauc/keyring.pem
check-purpose=codesign-rauc

[slot.rootfs.0]
device=/dev/disk/by-partlabel/rootfsA
bootname=A
```

⚠ GOTCHA: FALSCHER ZWECK

RAUC erwartet per Default S/MIME, unser Zertifikat ist `codeSigning`: *unsuitable certificate purpose*.

Fix: `check-purpose=codesign-rauc`

⚠ GOTCHA: VERITY OHNE KERNEL

`linux-yocto` schaltet `MISC_FILESYSTEMS` ab – und damit heimlich auch SquashFS.

Fix: `CONFIG_SQUASHFS=y`
`CONFIG_DM_VERITY=y`

⚠ GOTCHA: POST-QUANTEN-MIKROWELLE

ML-DSA braucht OpenSSL $\geq$ 3.5 – auch auf dem Gerät. Und die Sidecar-Signatur *vor* `rauc install` prüfen, sonst ist sie Deko.

Bundles bauen & doppelt signieren



Gerät: erst Sidecar prüfen, dann installieren

```
$ openssl cms -verify -binary \
    -inform DER -in update.raucb.sig \
    -content update.raucb \
    -CAfile /etc/rauc/keyring.pem \
    -out /dev/null \
  && rauc install update.raucb
```

Hybrid: RSA für heute, ML-DSA für den Tag, an dem jemand einen Quantencomputer auf deine Mikrowelle richtet.

⚠ GOTCHA: KEYRING-DRIFT

Neue Schlüssel – das Rezept lieferte noch das alte Trust-Bundle.

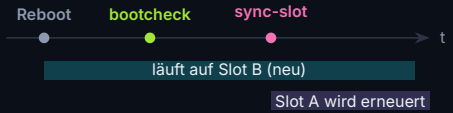
Fix: `--stage-keyring` (an beide Orte, `:=` statt `=`)

⚠ GOTCHA: SCHLÜSSEL FEHLEN

Kryptischer Fehler erst tief im `do_compile`.

Fix: `bb.fatal()` schon beim Parsen, mit Anleitung

Nach dem Reboot: gut markieren, dann nachziehen



bash

```
# Boot ok? Markieren (grubenv + RAUC)
$ rauc status mark-good booted
# erst danach: anderen Slot nachziehen
$ rauc install /var/cache/update.raucb
```

⚠ GOTCHA: RICHTIGE VERSION?

Nur nachziehen, wenn der Build-Zeitstempel des gebooteten Slots zum Bundle passt – sonst lief gar nicht die neue Version.

⚠ GOTCHA: ZWEI WAHRHEITEN

Nur `grub-editenv` zurücksetzen ⇒ RAUC hält den Slot weiter für *bad* und verweigert die nächste Installation.

Fix: `rauc status mark-good booted`

⚠ GOTCHA: ORDER WIRD UMGESCHRIEBEN

`rauc install` macht den frisch geschriebenen Slot zum primären – auch beim Nachziehen!

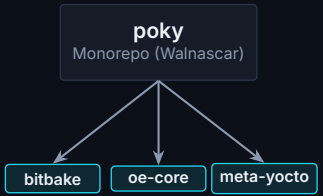
Fix: `danach ORDER wiederherstellen, rauc.slot= aus /proc/cmdline`

KAPITEL 11

Yocto-Upgrade

Alle sechs Monate wieder: Was ist diesmal kaputt?

Was beim Release-Wechsel kaputt geht

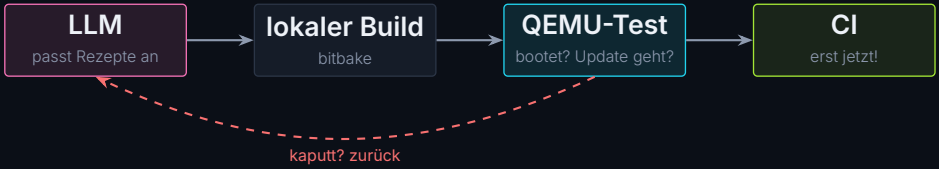


Wrynose: drei Submodule

Walnascar		Wrynose
`poky/meta`	→	`openembedded-core/meta`
`LAYERSERIES_COMPAT`		`"walnascar"` → `"wrynose"`
`wlroots_git.bbappend`	→	`wlroots_%.bbappend`
`S = "${WORKDIR}/git"`	→	löschen (passiert automatisch)
`S` bei Tarballs	→	`${UNPACKDIR}/…` statt `WORKDIR`
`wic/`	→	`files/wic/`

Viele kleine Änderungen in config & Rezepten – aber **die Rezepte existieren schon**.

LLMs sind deine Freunde (testen musst du trotzdem)



PROMPT-IDEE

„Hier sind Rezept und bitbake-Log. Finde die Ursache und fixe sie minimal per `.bbappend`."

Die Cross-Compile-Klassiker aus Kapitel 3 erkennt es zuverlässig.

DAS LLM

…hat die Rezepte nicht geschrieben. Aber es hat sie *alle* gelesen. Du musst nur noch prüfen, ob es auch bootet.

⚠ GOTCHA

Ungetestet in die CI pushen = 48 h Wartezeit für einen Tippfehler.

Fragen?

```
$ bitbake antworten
ERROR: Nothing PROVIDES 'antworten'
# ...dann eben live.
```

Danke! • Nook 2026