OpenWrt/LEDE on NAND

First-aid instructions for survival inside a rotting environment

Daniel Golle, OpenWrt Summit 2016

Why?

- No standards
- No hard constraints
- No relevant sales arguments
- Little academic interest
- Trivial to get right, but people do get it wrong all the time

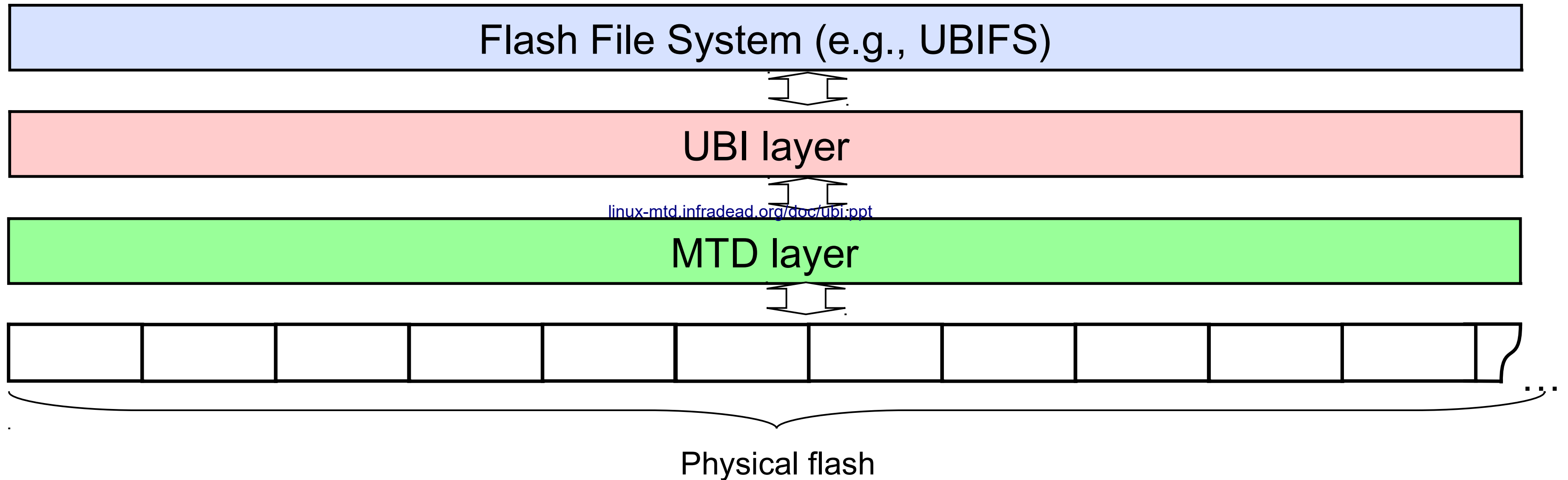
The good old days

- SPI or MMAP’ed NOR flash
- Slow and small, but reliable
- Bootloader, Kernel, rootfs and rootfs_overlay live in partitions, no need for special encodings or the like, a simple CRC is enough

The bright future

- NAND (and eMMC, but that’s on another page)
- Fast and big, but unreliable
- Bootloader needs to be read by chip-loader
  - usually simple FEC, eg. Hamming Code
- Kernel needs to be read by bootloader
  - ranging from plain binary to UBI
- rootfs and rootfs_overlay needs to be dealt with by the kernel
  - ranging from obscure things like JFFS2-NAND and YAFFS; ideally UBIFS

UBI layer



Source: http://linux-mtd.infradead.org/doc/ubi.ppt

Booting from NAND

- In-SoC SROM bootstrap loader needs to load real bootloader (ie. Das U-Boot) from NAND

  → SoC-specific encoding, eg. Hamming code
- U-Boot reads it’s environment and the kernel, launches Linux

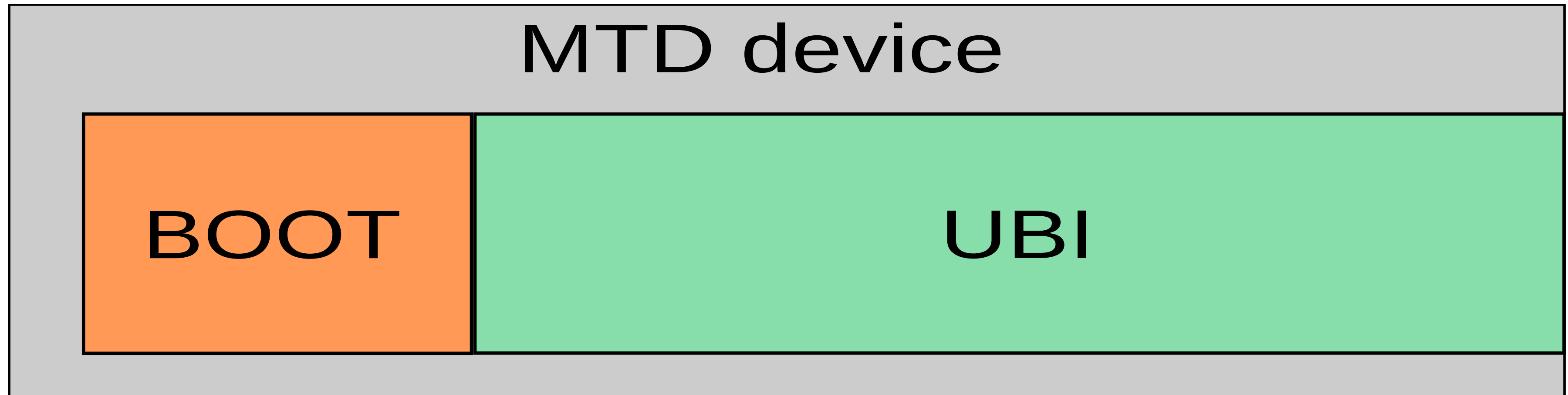
From UBI?

NAND, remember?

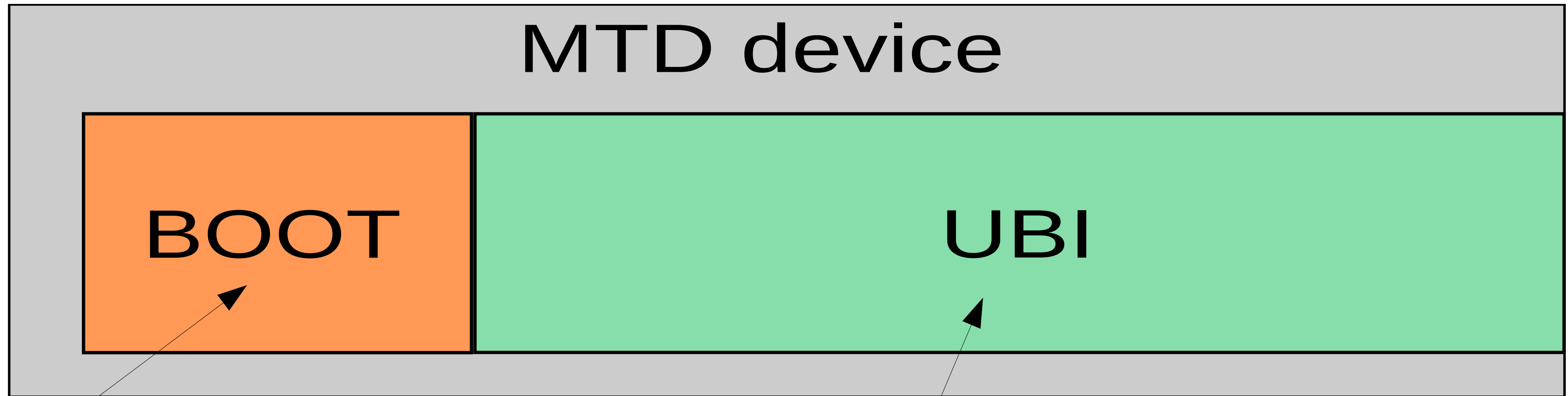
NAND, remember?

- NAND suffers, to different degrees, from data-rot, increasing bad-blocks and limited erase-cycles (MLC: thousands of cycles, SLC: ten-thousands)
- Bootloader is stored in extensive error-correction encoding to avoid bricking on bit-flips
- UBI can do the remaining job, but you got to use it consequently!

Boot layout on NAND Flash



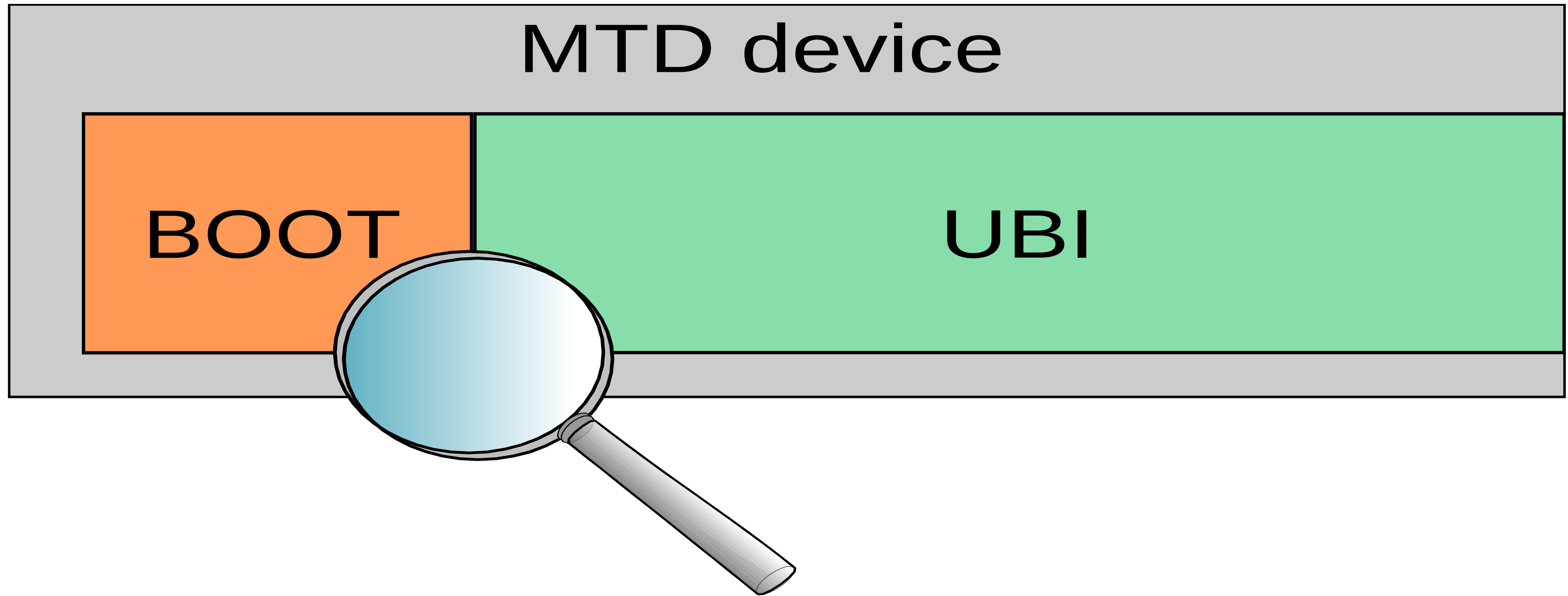
Boot layout on NAND Flash



Simple FEC, eg. Hamming Code

ECC, wear-leveling, ...

Boot layout on NAND Flash



MTD

BOOT

What’s hiding in the gap?

(and thus without error detection/correction, wear-leveling, ...):

U-Boot environment

Kernel

EEPROM (WiFi calibration, MAC addresses, …)

What’s hiding in the gap?

(and thus without error detection/correction, wear-leveling, ...):

U-Boot environment (write: never)

Kernel (write: on every sysupgrade!)

EEPROM (write: never)

U-Boot: UBI support

- Kept in sync with Linux Kernel sources
- Supports bootloader environment in UBI
- Boot image(s) (FIT/uImage) in UBI
- Use it!
- Use plain UBI volumes to avoid complexity of UBIFS

OpenWrt/LEDE: new image building template

- Replaces messy platform-specific code
- Eases support for more complex storage devices
- Pipeline syntax: `kernel-bin | lzma | uImage lzma`
- All per-device information in one place
- New: per-device root-filesystem generation *in parallel*

UBI image generation template parameters

UBOOTENV_IN_UBI :=

KERNEL_IN_UBI :=

BLOCKSIZE :=

PAGESIZE :=

SUBPAGESIZE :=

VID_HDR_OFFSET :=

UBINIZE_OPTS := -E 5

MKUBIFS_OPTS :=

Factory images

- ubinized image with ‘freespace-fixup’ set

  http://www.linux-mtd.infradead.org/faq/ubifs.html#L_free_space_fixup
- containing kernel and rootfs, maybe U-Boot’s environment as well if you need it
- ‘autoresize’ set for UBIFS r/w volume
- Integrated in OpenWrt/LEDE image generation

  `CONFIG_TARGET_UBIFS_FREE_SPACE_FIXUP`

Factory images

That alone is not enough, unfortunately:

- Remaining blocks need to be formated!
- Can be done similarly to JFFS2 fixup, on MTD block level
- Patch needs to be discussed more to get upstream

OpenWrt/LEDE Kernel hacks

- Attach and mount rootfs based on naming-convention, just like on plain (ie. NOR) flash layouts.
- MTD partition named ‘ubi’ is attached during boot
- Probe-mount UBIFS volume ‘rootfs’
- If there is a non-UBIFS volume named ‘rootfs’, a ubiblock device is created and set as ROOT_DEV, which is then mounted just like any other block-based rootfs device.

OpenWrt/LEDE early run-time

- mounts UBIFS overlay on-top of squashfs ROM
- support for extroot was added
- supports mounting additional custom/user UBIFS volumes

UBI support in OpenWrt/LEDE userspace tools

- libfstools and ‘block’ tool
- new tar-based sysupgrade format
- fw_printenv/fw_setenv uboot-envtools

Problems...

- Replacing non-UBI stock firmware
- Vendors use outdated versions of U-Boot lacking support for UBI and thus store kernel and environment on the bare MTD device (NAND, remember?)
- Great variety of custom dual-boot hacks, some wasting write-cycles of hard-coded physical blocks they depend on

Best practise

- use UBI for everything except the bootloader itself (until SoC chip-loaders will support UBI)
- avoid unnecessary writes!
- U-Boot is GPL licensed software, so publish your U-Boot patches (eg. using git{hub,lab,...})
- Transparently implement dual-boot and/or recovery features using U-Boot scripting
- Keep it simple!

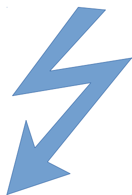
U-Boot ToDos

- Implement FIT TFTP update for UBI
- Create generic bootmgr handling dual-boot, rescue-/recovery/fallback-variants and TFTP update

  Each vendor-specific dual-boot solution needs it’s counter-part in our code! Many of them suffer from severe problems in their implementions (eg. using writing envorinment on *every* boot)

Kernel ToDos

- replace auto-attach hack by device-tree property of MTD partition (or even kernel cmdline, if that's the only acceptable solution)
- parse kernel cmdline's ubi volume syntax in 'rootfs=' parameter like UBIFS does instead of using the hard-coded name 'rootfs' for ubiblock-creation and ROOT_DEV magic



linux-mtd community doesn't like probe-hacks and wants us to use an initramfs to probe and mount things on UBI.

Userland ToDos

- Support for rootfs commit/rollback on UBI (libfstools)
- Generalize dual-boot sysupgrade
- Revive discussion on sysupgrade image and metadata which started during BattleMesh in Porto
- Some services may depend on storage to be ready

Acknowledgements

This talk would not have been possible without

- Eric Schulz of prpl Foundation

Thanks for debating and reviewing

- John Crispin and Hauke Mehrtens *(LEDE Project)*
- Richard Weinberger of *(Linux-MTD)*

Discussion