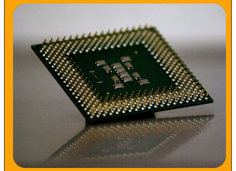


IPL+UBI: Flexible and Reliable with Linux as the Bootloader

John Ogness
Linutronix GmbH

Embedded Board Trends

- MMC / SD
- CompactFlash
- SATA
- USB (OTG)
- Firewire
- RFID
- Ethernet (wireless)

- TFT LCD
- Touchscreen
- Audio
- Sensors
  - temperature
  - brightness
  - proximity
  - accelerometer

Embedded Board Trends

- MMC / SD
- CompactFlash
- SATA
- USB (OTG)
- Firewire
- RFID
- Ethernet (wireless)

- TFT LCD
- Touchscreen
- Audio
- Sensors
  - temperature
  - brightness
  - proximity
  - accelerometer

Linux Drivers Necessary

Embedded Board Trends

- MMC / SD
- CompactFlash
- SATA
- USB (OTG)
- Firewire
- RFID
- Ethernet (wireless)

- TFT LCD
- Touchscreen
- Audio
- Sensors
  - temperature
  - brightness
  - proximity
  - accelerometer

... but what about the bootloader?

Bootloader Responsiblities

- init low-level hardware (RAM, MTD)
- load/run kernel from MTD

Bootloader Responsiblities

- init low-level hardware (RAM, MTD)
- load/run kernel from MTD
- ▸ failsafe boot environment

Bootloader Responsiblities

- init low-level hardware (RAM, MTD)
- load/run kernel from MTD

‣ failsafe boot environment

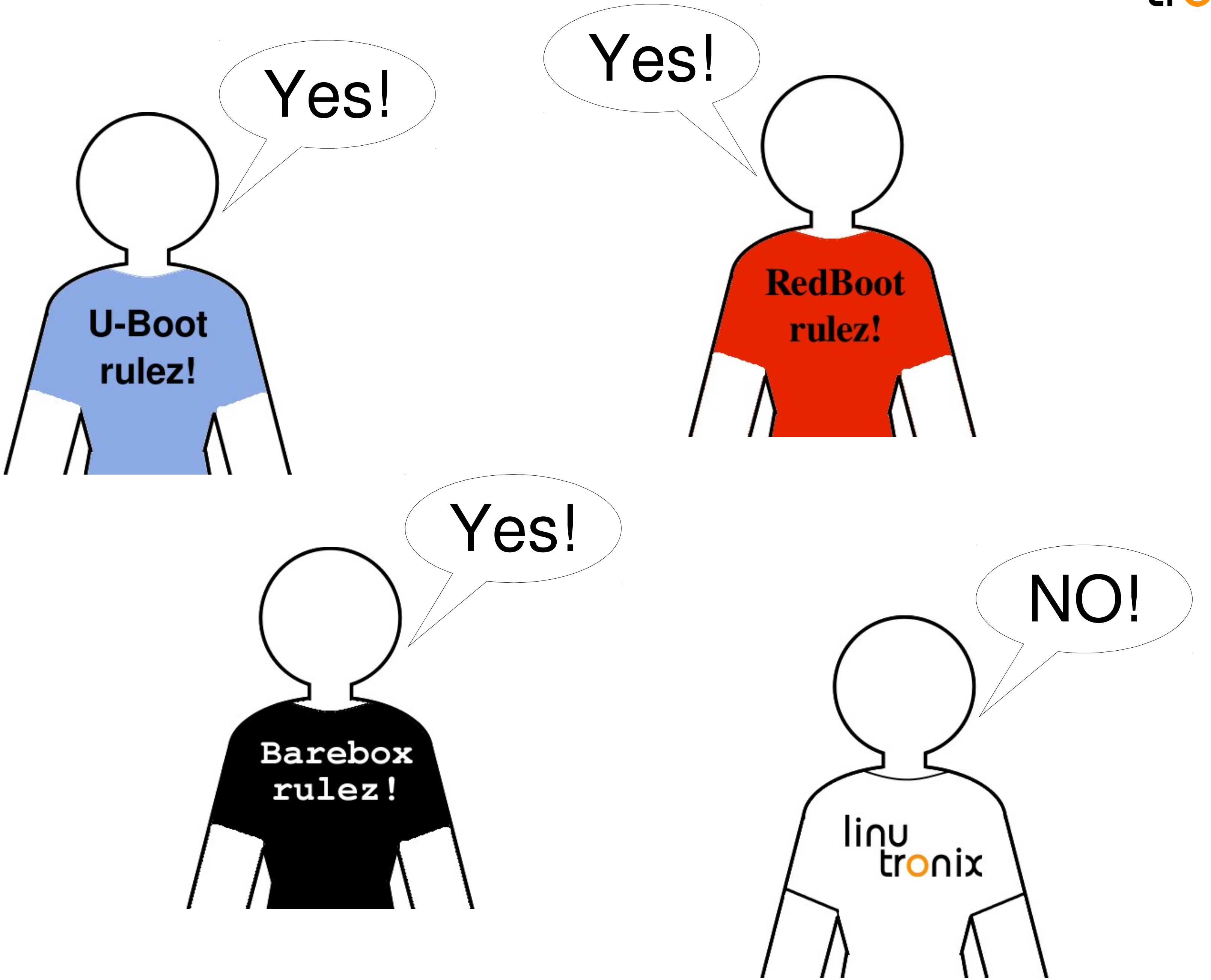
- show a splash screen
- play startup music
- BIOS-like application (user interaction)
- load/run kernel from external device (USB / MMC / SATA / ethernet)

Bo

Do we really want to implement all these drivers again for the bootloader?

- in
- load/run kerne
- failsafe boot environment

- show a splash screen
- play startup music
- BIOS-like application (user interaction)
- load/run kernel from external device (USB / MMC / SATA / ethernet)



Original icons (Restroom_symbols.svg) copyright Amy Young
under the Creative Commons Attribution-ShareAlike 3.0 License.

Drivers in Bootloaders

Advantage

- early access to hardware

Drivers in Bootloaders

Advantage

- early access to hardware

Disadvantages

- less availability
- less tested
- less maintained
- often incomplete
- less support for out-of-tree drivers

What if we use Linux as the bootloader?

Linux as Bootloader

- use a minimal kernel (to reduce startup times)
- load drivers on demand as needed
- BIOS-like application runs in a Linux userspace environment
- load final kernel from wherever
- use kexec to run final kernel

... but how does the Linux bootloader get loaded?

IPL (Initial Program Load)

- also known as “X-Loader” or “Stage One Bootloader”
- less than 16KB in size
- performs minimal tasks required to start the bootloader

Bootloader Responsiblities

- init low-level hardware (RAM, MTD)
- load/run kernel from MTD

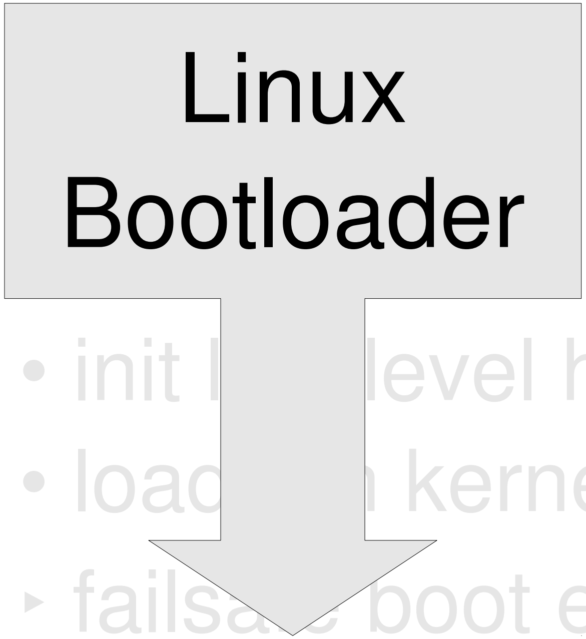
‣ failsafe boot environment

- show a splash screen
- play startup music
- BIOS-like application (user interaction)
- load/run kernel from external device (USB / MMC / SATA / ethernet)

IPL Responsiblities

- init low-level hardware (RAM, MTD)
- load/run kernel from MTD
- ‣ failsafe boot environment
- show a splash screen
- play startup music
- BIOS-like application (user interaction)
- load/run kernel from external device (USB / MMC / SATA / ethernet)

Responsiblities



Linux Bootloader

- init l level hardware (RAM, MTD)
- load kernel from MTD
  - failsa boot environment

- show a splash screen
- play startup music
- BIOS-like application (user interaction)
- load/run kernel from external device (USB / MMC / SATA / ethernet)

Boot Procedure

1. boot ROM loads and runs IPL
2. IPL loads its configuration from MTD
3. IPL loads and runs minimal kernel from MTD
4. BIOS-like application runs (initramfs)
5. load and run final kernel from wherever

Boot Procedure

1. boot ROM loads and runs IPL
2. IPL loads its configuration from MTD
3. IPL loads and runs minimal kernel from MTD
4. BIOS-like application runs (initramfs)
5. load and run final kernel from wherever

NAND (MTD)

NAND (MTD)

- data organized in addressable pages
  - 512, 2048, or 4096 bytes per page (+ spare bytes for ECC)
- pages organized in erase blocks
  - 32, 64, or 128 pages per erase block
- individual pages may be programmed
- to reprogram a page, the entire erase block must first be erased
- limited program-erase cycles

NAND (effective use)

- wear-leveling
  - wear down the entire MTD evenly
- bad block management
  - NAND erase blocks will go bad
- error correction code
  - bit-flips when reading NAND will occur
- data scrubbing
  - keep data on “healthy” erase blocks

UBI (Unsorted Block Images)

- volume management system for raw flash devices
- maps logical erase blocks to physical erase blocks
- transparently implements effective MTD usage
- additional data integrity features
  - erase block checksums
  - erase block sequence numbers

UBI (a few nice features)

- supports multiple volumes within a single MTD partition
- supports static volumes
  - useful for storing data blobs (e.g. kernel)
- supports dynamic volumes
  - used for filesystems (e.g. UBIFS)
- individual volumes may be updated
  - ... with effective MTD usage

UBI (how it works)

- UBI management “attaches” to a specified MTD partition
- UBI scans the MTD partition
  - LEB/PEB mapping created in RAM
  - erase block checksums and sequence numbers used to identify valid data
  - “global” sequence number set to the highest valid erase block sequence number found (in RAM)

UBI (in Linux)

- device nodes are created for each UBI volume
  - static volumes can be read
  - dynamic volumes can be mounted
- userspace UBI tools available
  - ubiformat
  - ubiattach / ubidetach
  - ubinfo
  - ubinize / ubimkvol / ubirmvol
  - ubiupdatevol

Boot Procedure

1. boot ROM loads and runs IPL
2. IPL loads its configuration from MTD
3. IPL loads and runs minimal kernel from MTD
4. BIOS-like application runs (initramfs)
5. load and run final kernel from wherever

Boot Procedure

1. boot ROM loads and runs IPL
2. IPL loads its configuration from an UBI static volume (MTD)
3. IPL loads and runs minimal kernel from an UBI static volume (MTD)
4. BIOS-like application runs (initramfs)
5. load and run final kernel from wherever

IPL+UBI (example)

- 1st MTD partition = 4 erase blocks
  - each has a copy of the IPL
- 2nd MTD partition = 10 MB, UBI managed with (at least?) 4 static UBI volumes
  - IPL config
  - minimal kernel
  - production kernel
  - failsafe kernel
- 3rd MTD partition = “the rest”
  - most likely also UBI managed ;-)

IPL+UBI (example cont.)

- OMAP3530 (ARM Cortex A8 @ 720 MHz)
- 64 MB NAND (0.5 KB pages, 32 PPB)
- IPL size = 12 KB
  - includes simple UBI library
- IPL config size = 0.1 KB
- “minimal” kernel size = 940 KB
  - compressed using LZMA
  - includes drivers for NAND, UBI, display
  - includes initramfs with busybox and kexec

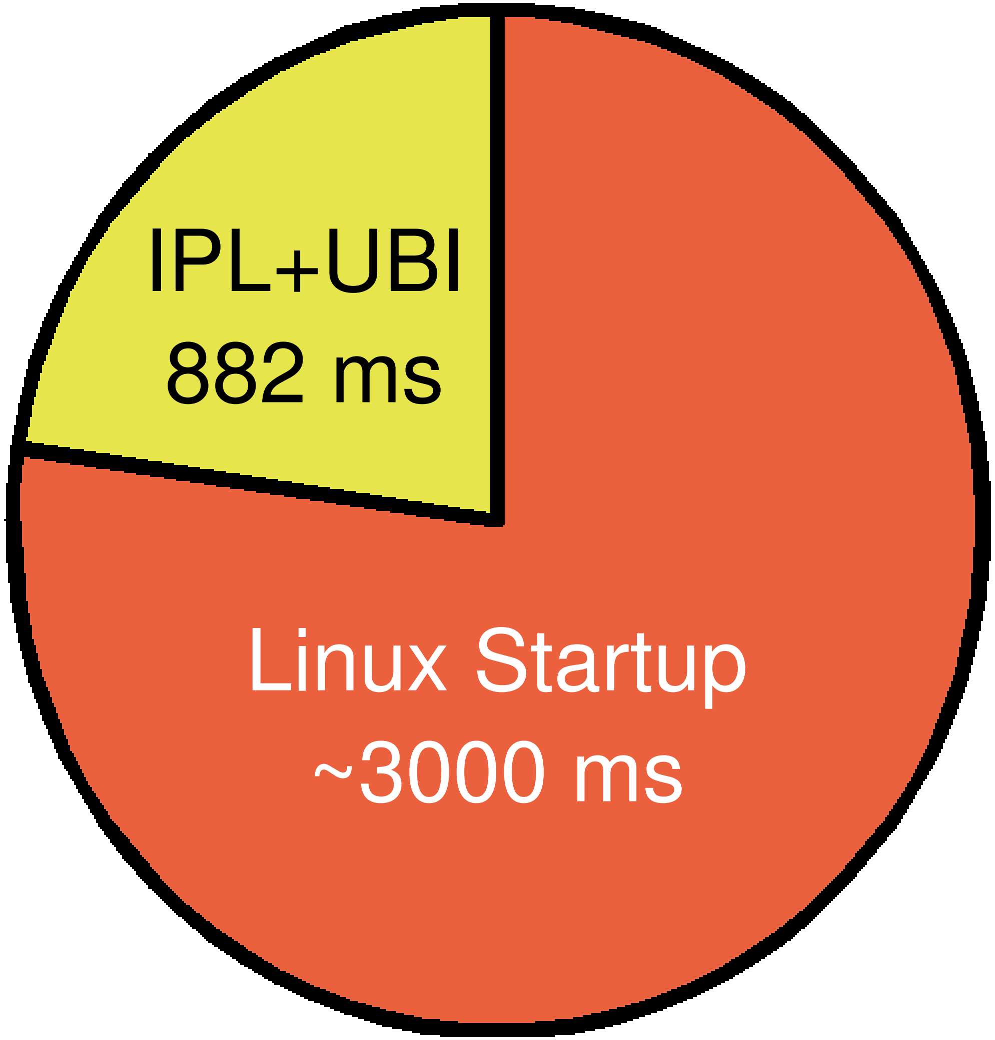
IPL+UBI (example cont.)

Startup time until Tux visible:

- UBI scan (2nd MTD partition) = 180 ms
- load IPL config = 2 ms
- load minimal kernel = 700 ms
- kernel startup (until Tux) ≈ 3000 ms
  - decompress kernel = 1000 ms
  - boot until Tux visible ≈ 2000 ms

Total time to Tux ≈ 4 seconds

Total Time to Tux



Questions / Comments

`RCPT TO:<john.ogness@linutronix.de>`

Other Presentations On This Topic

09:45 - The Right Approach to Minimal Boot Times

11:00 - Barebox: Booting Linux Fast and Fancy

15:20 - Flash Filesystem Benchmarks

16:10 - YAFFS Updates