

Embedded-Linux

Wie komme ich zu einem Embedded-Linux-System?

Andreas Klinger
ak@it-klinger.de

IT - Klinger
<http://www.it-klinger.de>

Linux-Tag Berlin
22.05.2013

Teil I

Embedded-Linux-System

1 Aufbau — Embedded-Linux

2 Toolchain

3 JTAG und OpenOCD

4 Linux-Kernel

5 Root-Filesystem

6 Dateisysteme

7 Ausblick

1 Aufbau — Embedded-Linux

Ist Embedded-Linux ein anderes Linux? I

Embedded- und Desktop-/Server-Linux haben gemeinsam:

- Bootloader für Initialisierungen und Ladevorgänge
- Linux-Kernel, das eigentliche Betriebssystem
- Root-Filesystem, eine Sammlung von Programmen und Tools
- Dämonen als Hintergrund-Prozesse

Unterschiede hinsichtlich:

- Hardware-Architektur
- Ressourcen-Limitierungen (Rechenleistung, Speicher, Peripherie, ...)
- Verfügbarkeit, auch im autonomen Betrieb
- Echtzeitanforderungen
- System-Update im Feld
- Reproduzierbarkeit des Komplettsystems
- Massenspeicher (managed / unmanaged Flash)

Deshalb:

- Embedded-Linux ist ein kleines, auf definierte Aufgaben reduziertes Linux
- Produkt wird verkauft
Benutzer kennt eingesetztes Betriebssystem nicht
⇒ kein User / Admin vorhanden
- Die **eine** Distribution für **alle** Embedded-Linux-Systeme nicht vorhanden

Linux-Embedded-System

Bootloader

Linux-Kernel

Root-Filesystem

Init-
Prozeß

/sbin/init

Libraries
libc

/lib

Device
Nodes

/dev

Shell-
Programme
Dämonen

/bin
/sbin
/usr/bin
...

Konfiguration

/etc

Entwicklungsschritte für ein Board ohne Bootloader und Linux

- 1 Cross-Development-Toolchain (gcc, binutils, gdb, libc, ...)
- 2 Bootloader (aufspielen und debuggen mittels JTAG und OpenOCD)
- 3 laden von Linux-Kernel mit RAM-Disk durch Bootloader
- 4 schreibbares Root-Filesystem (UBI-FS, ext3, ...) anlegen und Kernel flashen

Beispiele im Vortrag beziehen sich auf nachfolgende Hardware

- Zielsystem ähnlich aber nicht identisch mit Sheevaplug
- Board von Wiesemann & Theiss: ARM 926ejs, Marvell Kirkwood, Feroceon-CPU - Typ 88F6180
- NAND-Flash 1 GB mit 128 k Erasesize und 2 k Pagesize, Samsung K9K8G08U0A
- 128 MB DDR2-RAM mit 64 Mib x 16 (MT47H64M16HR-3:E; D9HNZ)
- Console auf UART-1 (ttyS1), MPP-Pins 13 u. 14, Typ NC16550
- On-Chip-Debugging mit OpenOCD
- JTAG-Adapter: ARM-USB-TINY-H, FTDI-2232-Chip

Start	Größe	Verwendung
0x00000000	0x000E0000	u-boot (896k) Bootloader
0x000E0000	0x00020000	u-boot-env (128k) Umgebungsvariablen vom Bootloader
0x00100000	0x02000000	kernel (32M) Linux-Kernel
0x02100000	0x08000000	rootfs (128M) Root-Filesystem
0x0A100000	0x35F00000	data (951M) Anwendungsdaten

2 Toolchain

- Entwicklungswerkzeuge auf dem Host-System erstellen
Bootloader, Kernel und Root-Filesystem des Target
- produzieren und verarbeiten Instruktionen des Ziel-Systems
⇒ Cross-Development-Toolchain
- Open-Source-Tools für die Generierung der Toolchain
- die Erstellung einer Toolchain „From-The-Scratch“ kann erheblichen Aufwand bedeuten
- auf passende Version hinsichtlich Kernel und Target achten

Entwicklungsrechner — buildroot

<http://www.buildroot.org>

Version 2013.02

```
cd /home/linux/inst
```

```
tar -xzf buildroot-2013.02.tar.gz
```

```
cd buildroot-2013.02
```

```
make sheevaplug_defconfig
```

```
make menuconfig
```

```
make
```

⇒ **Cross-Development-Toolchain** (/usr/arm-linux)

⇒ **Root-Filesystem** (output/images/rootfs.ext2)

Installationspfad der Toolchain einstellen

Build options

→ Host dir

→ /usr/arm-linux

→ Toolchain dort verwenden, wo sie erstellt wurde (absoluter Pfad)

→ auf anderem Rechner im identischen Pfad verwendbar

→ ansonsten: Sysroot setzen

Cross-Development-Toolchain verwenden II

Source-Datei `armenv.sh` erstellen

```
export PATH=/usr/arm-linux/usr/bin:$PATH
export ARCH=arm
export CROSS_COMPILE=arm-linux-
```

Umgebungsvariablen sourcen

```
source armenv.sh
. armenv.sh
```

oder

⇒ Bootloader, Kernel, Anwendungen, ... damit kompilieren

Beispiel für eigenes Makefile

```
all:
    ${CROSS_COMPILE}gcc appl.c -o appl
```

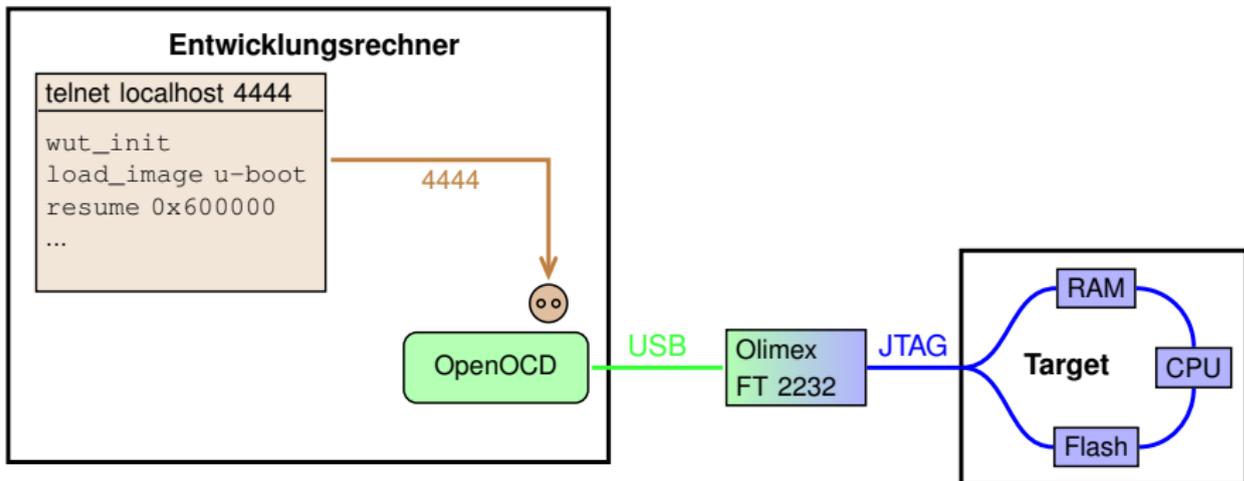
TAG

3 JTAG und OpenOCD

- JTAG (Joint Test Action Group)
- ursprünglich zum Testen und Debuggen integrierter Schaltungen entwickelt (In-Circuit)
- bei vielen Boards als Hardware-Debugging und Flash-Programmier-Schnittstelle nutzbar
- Bsp: Aufspielen und Debuggen des Bootloaders
- Voraussetzung: JTAG-Adapter zwischen Embedded-Board und Entwicklungsrechner sowie unterstützende Software zum Programmieren, Debuggen, ...
- neben kommerziellen Produkten auch Open-Source-Entwicklungen verfügbar

- OpenOCD (Open On-Chip Debugger) wurde im Rahmen einer Diplomarbeit an der FH Augsburg entwickelt
- OpenOCD-Dämon kommuniziert mit JTAG-Adapter und liefert
 - Schnittstelle für Terminal-Verbindung (Port 4444)
 - Schnittstelle für gdb-Debugger (Port 3333)
- Konfigurationsskripte:
 - JTAG-Adapter
 - CPU-Kern
 - Board
- Voraussetzung: JTAG-Verbindung zum Embedded-Board

JTAG-Verbindung — OpenOCD



libftdi installieren

Download von libftdi:

<http://www.intra2net.com/en/developer/libftdi/download.php>

```
cd /home/linux/inst
```

```
tar -xzf libftdi-0.20.tar.gz
```

```
cd libftdi-0.20
```

```
./configure --prefix=/usr
```

```
make
```

```
make install
```

OpenOCD installieren

```
cd /home/linux/inst
```

```
git clone git://openocd.git.sourceforge.net/  
gitroot/openocd/openocd
```

```
cd openocd
```

```
./bootstrap
```

```
./configure --enable-maintainer-mode
```

```
--enable-ft2232_libftdi --prefix=/usr
```

```
make
```

```
make install
```

```
(installiert unter /usr/share/openocd/)
```

OpenOCD verwenden

```
cd /home/linux/inst
```

```
cp /usr/share/openocd/scripts/  
board/sheevaplug.cfg ./openocd.cfg
```

```
openocd
```

Default-Skript `openocd.cfg` anpassen:

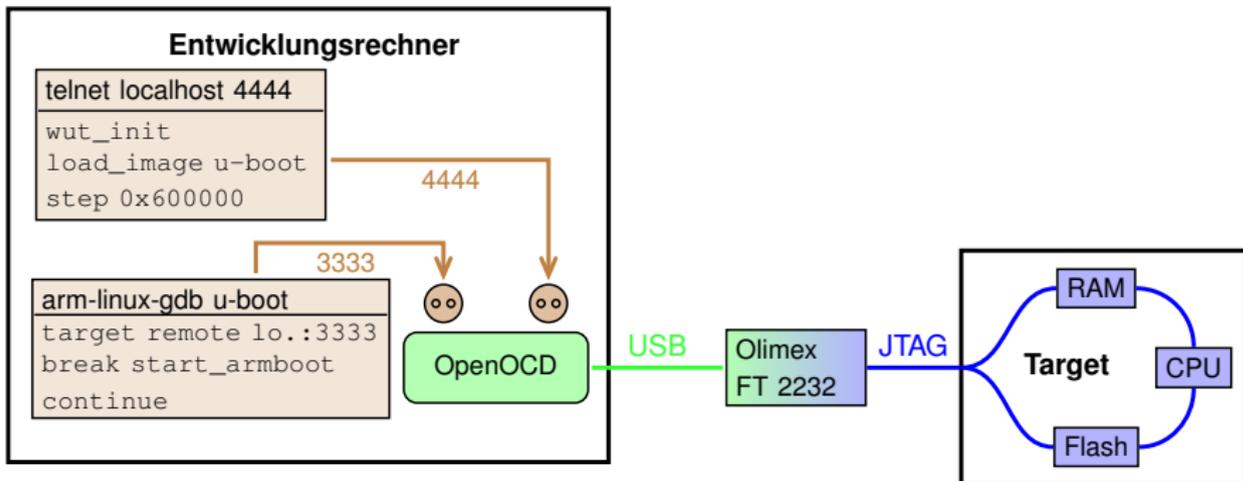
Interface richtig einstellen (`find interface/...`)

JTAG-Geschwindigkeit (`jtag_khz 2000`)

Prozedur `wut_init()` erstellen mit CPU-Register-Einstellungen

(abgeleitet von `sheevaplug_init()`)

Debuggen mit JTAG-Verbindung — OpenOCD

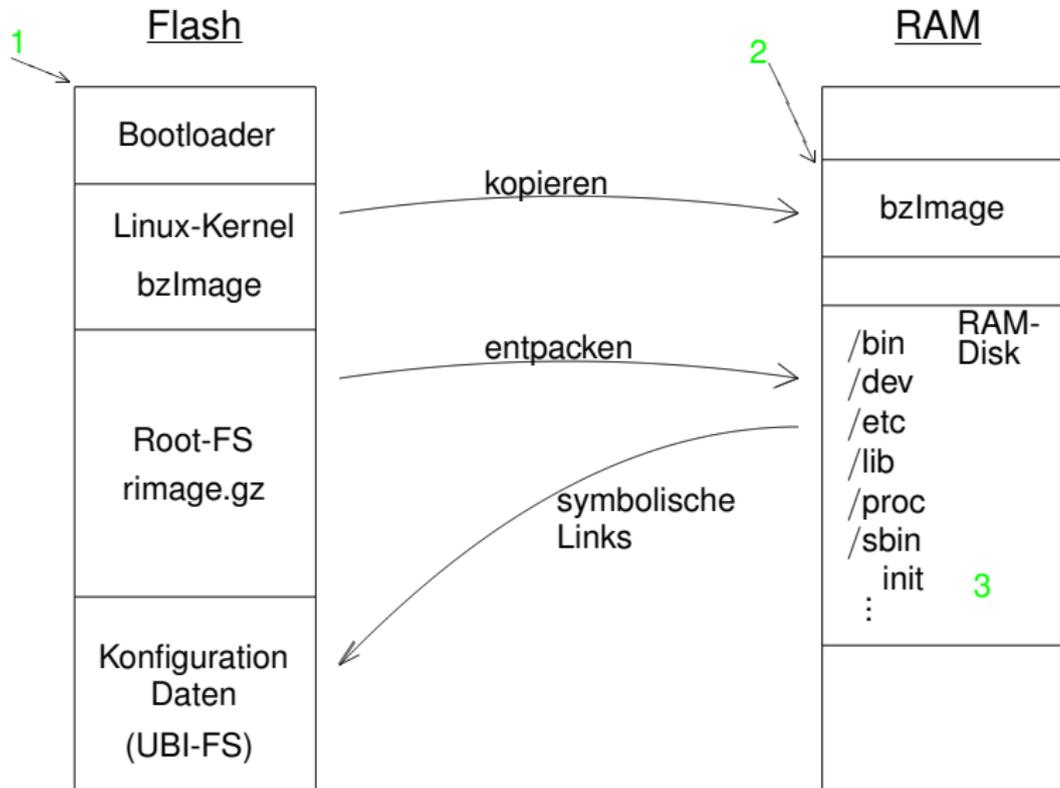


4 Linux-Kernel

Booten des Linux-Kernels

- öffnet Konsole
- entpackt sich selber (optional)
- initialisiert zentrale Einheiten, den „Kern des Kernels“:
Memory-Management, Scheduling, Interprozess-Kommunikation
- initialisiert Geräte- und Dateisystemtreiber
- mountet das Root-Filesystem
- startet den init-Dämon `/sbin/init`

Bootvorgang eines Embedded-Linux mit RAM-Disk



Entwicklungsrechner — Kernel erstellen

`http://ftp.kernel.org`

linux-3.6.1-rt1

Kernel entpacken, RT-Patch einspielen

```
source armenv.sh
```

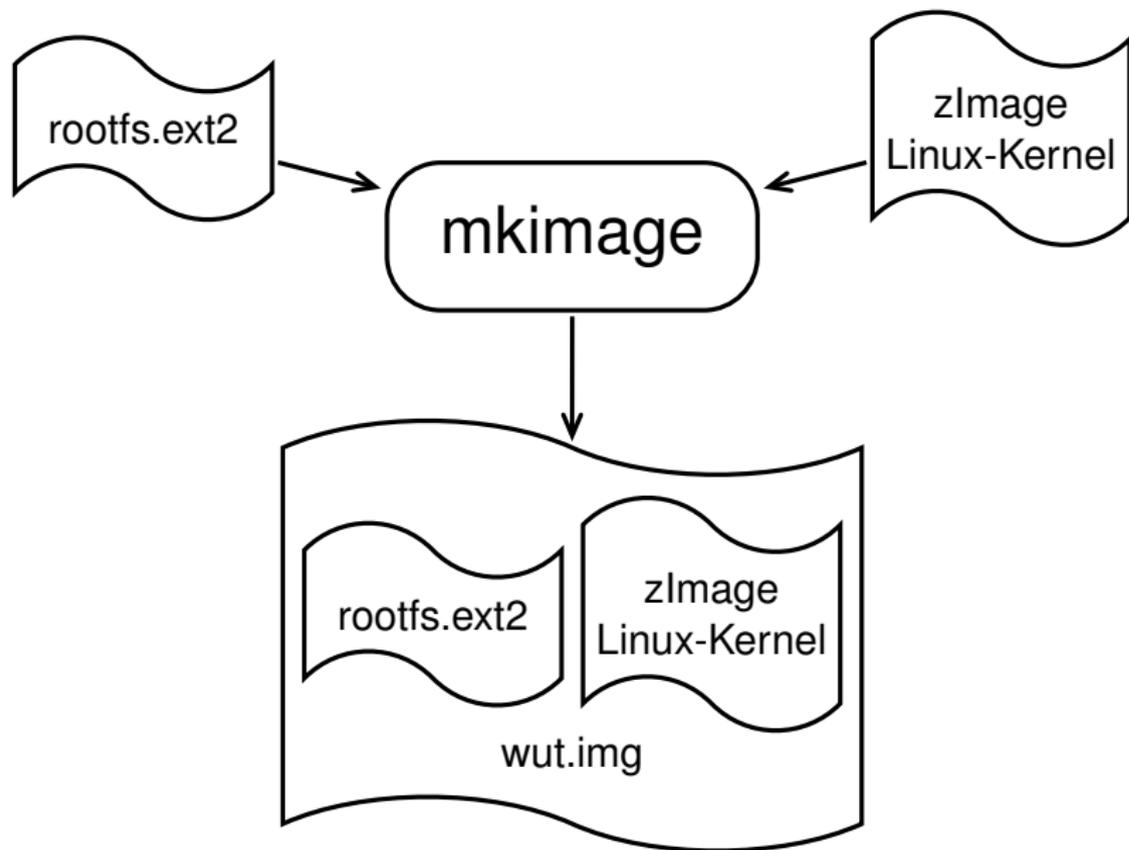
```
make kirkwood_defconfig
```

```
make menuconfig
```

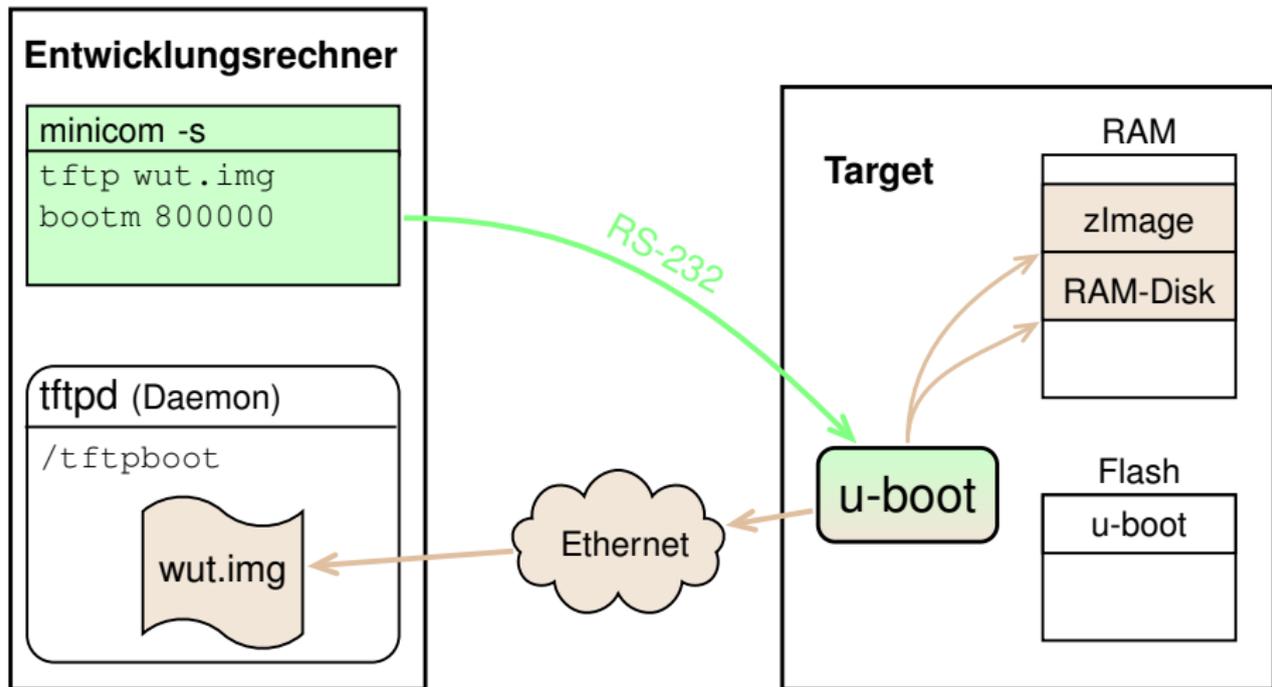
```
make
```

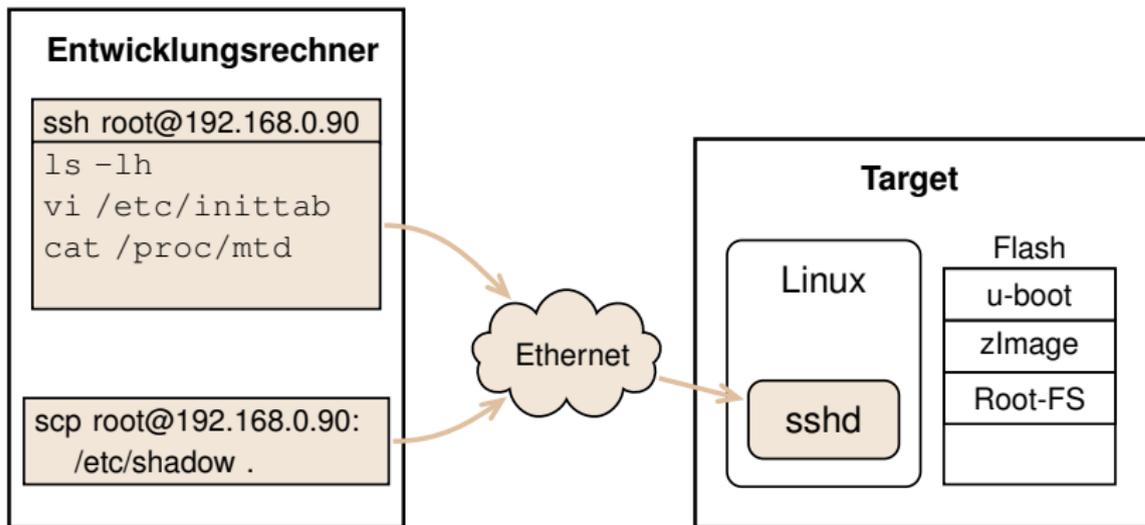
Kernel-Image

U-Boot-Image generieren



Bootloader-Verbindung





5 Root-Filesystem

- Aufbau vom Root-FS
- buildroot
- Root-FS flashen

- Konfigurationsdatei: `/etc/inittab`
- nimmt Systemeinstellungen vor (IP-Adresse, ...)
- mountet weitere Dateisysteme (`/proc`, `/sys`, ...)
- startet Dämonen entsprechend dem Runlevel (`syslogd`, `sshd`, ...)
- respawned Terminals (Login-Shell)
- **Aufwand:**

Konfiguration vornehmen

ext2-Filesystem-Image erstellen

- leere Datei mit gewünschter Zielgröße anlegen

```
dd if=/dev/zero of=rimage bs=1k count=8192
```

- Dateisystem in Datei anlegen (kein reservierter Bereich, 512 I-Nodes, 128 Bytes / I-Node)

```
mkfs.ext2 -m0 -v -N 512 -I 128 -F rimage
```

- Datei als Dateisystem mittels Loopback-Device mounten

```
mkdir mnt
```

```
mount -o loop [-t ext2] rimage mnt
```

- Root-FS in gemountete Datei kopieren, unmounten und komprimieren

```
cp -av /my_rootfs/* mnt/
```

```
umount mnt
```

```
gzip -9 < rimage > rimage.gz
```

- Mounten des Root-FS im Target:

```
mkdir my_rootfs
```

```
mount -o loop rootfs.ext2 my_rootfs
```

- Shell mit Wurzelverzeichnis / auf gemountetes Root-FS ausführen:

```
chroot my_rootfs /bin/sh
```

- Beendigung mit `exit`

- Implementierung von Shell-Programmen und Dämonen speziell für Embedded Systeme
- Programme wurden in ihrer Funktionalität auf das Wesentliche beschränkt (80:20-Regel)
- nur ein einziges Executable, welches über Soft-Links aufgerufen wird

⇒ spart Entry- und Exit-Codesequenzen
- Funktionsumfang konfigurierbar

→ `make menuconfig` - Skripte

- libc-Implementierung; jedoch erheblich kleiner als glibc
- Konfigurierbar (locale-Support, IPV6, ...)
- Optimierung auf minimalen Footprint
- kein Support anderer Betriebssysteme wie MS-DOS
- Verzicht auf Debugging- und Tracing-Features (z. B. `backtrace()`)
- alle wichtigen Funktionalitäten sind enthalten

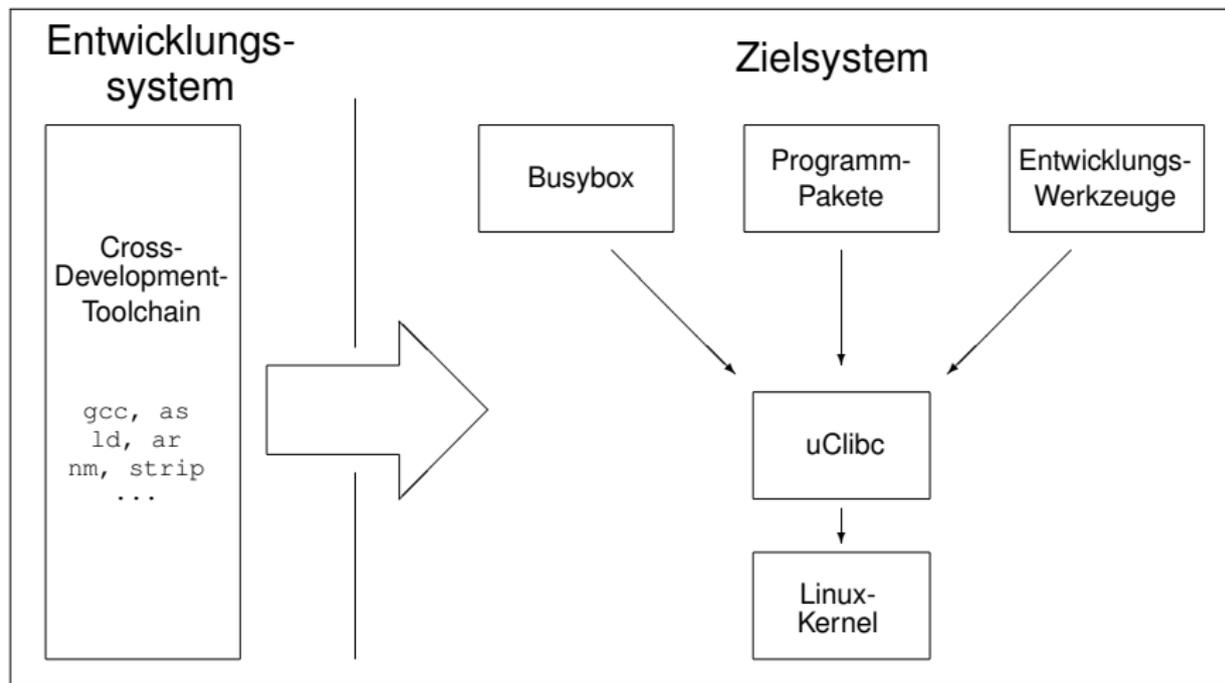
- Implementierung neuer Features mit zeitlichem Versatz (z. B. NPTL-Support)
- keine API-Kompatibilität zwischen Versionen oder Konfigurationen
⇒ Anwendungen immer gegen die im Target verwendete μ CLibc erstellen

⇒ einer erstellt Toolchain inkl. μ CLibc und verteilt diese im Entwicklungsteam, z. B.:

```
tar -czf unsere-toolchain.tgz /usr/arm-linux
```

- für Embedded Linux reduzierte glibc
- Abwärtskompatibilität
- Ziel: Binärkompatibel mit glibc
- neue Features zusammen mit glibc

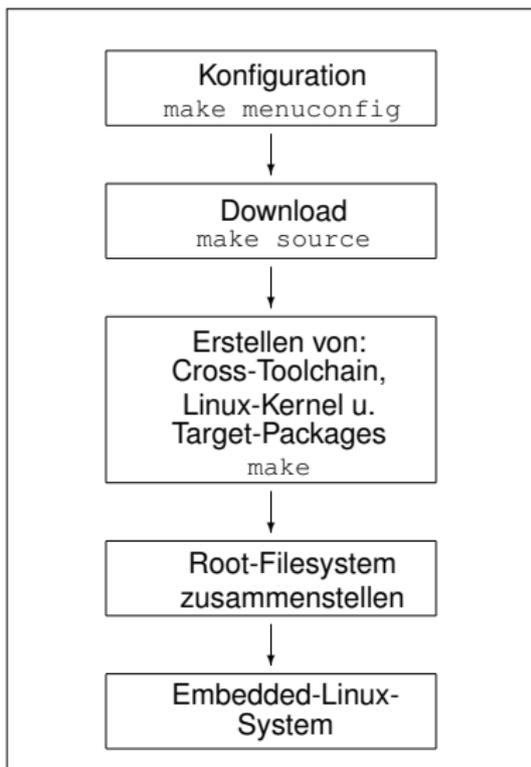
Erstellungsprozess eines Embedded Linux Systems



buildroot — automatisierter Target-Buildprozess

- Erstellung des Target-Images weitestgehend automatisiert mittels Makefiles
- konfigurierbar, adaptierbar und erweiterbar
- verfügbar für viele Architekturen
- downloaded, patched und erstellt automatisch aus den Sourcen
- komplett Open-Source-Software
- Root-Filesystem inklusive Konfigurationsdateien und Device-Nodes

Ablaufsequenz bei Verwendung von buildroot



Was beinhaltet buildroot?

- Cross-Development-Toolchain für Hostsystem
- native Entwicklungs- und Debugging-Werkzeuge für das Target
- busybox und μ Clibc
- viele zusätzliche Open-Source-Softwarepakete
- Bootloader für unterschiedliche Architekturen und Anwendungsfälle
- Linux-Kernel

Konfiguration

- **busybox anpassen:**
`make busybox-menuconfig`
- **μ Clibc anpassen:**
`make uclibc-menuconfig`
- **zusätzliche Pakete im buildroot auswählen**

Root-FS anpassen

- Login-Konsole setzen:

```
/etc/inittab → ttyS1
```

- Passwort vergeben für ssh-Login:

```
passwd
```

```
→ /etc/shadow
```

- IP-Adresse einstellen:

```
/etc/network/interface
```

```
/etc/init.d/rcS/S40network restart
```

- dropbear-Keys weiter verwenden:

```
/etc/dropbear
```

Anpassungen im Root-FS integrieren

- vom Target Dateien und Verzeichnisse in Root-FS übernehmen:
passwd, shadow, dropbear, ...
siehe Unterverzeichnis: `fs/skeleton`
- Datei-Rechtevergabe:
`target/generic/device_table.txt`
- generierte Device-Nodes:
`target/generic/device_table_dev.txt`

Anwendungen in Buildprozeß integrieren

Anwendung kopieren nach: `package/customize/source`
und Option `customize` in Konfiguration aktivieren

- QEMU emuliert Zielsystem auf Entwicklungsrechner
- Nutzung von vorkompilierten Debian-Paketen für das Zielsystem
- Definition des Zielsystems in XML-Datei
- Architekturen: x86, amd64, arm, powerpc
- Open-Source-Tool „elbe“ automatisiert diese Schritte
→ Ansatz für Debian-basierte Embedded-Linux-Distribution

Entwicklungsrechner

`rootfs.ext2` wird von `buildroot` erstellt

```
scp rootfs.ext2 root@192.168.0.90:/tmp/
```

Erasen und Kopieren

Zielsystem

Zielsystem mit Kernel und RAM-Disk booten

```
flash_erase /dev/mtd3 0 0
mount -t jffs2 /dev/mtdblock3 /mnt

mkdir /mnt2
mount -o loop /tmp/rootfs.ext2 /mnt2

cp -av /mnt2/* /mnt/
umount /mnt2
umount /mnt
```

Vorteil: Kernel-Treiber erstellt Filesystem das er später nutzt

Entwicklungsrechner — U-Boot-Kernel-Image generieren

Erstellung eines U-Boot-Images, welches zum Booten eines JFFS2-Dateisystems geeignet ist und nur den Linux-Kernel enthält:

```
mkimage -A arm -O linux -T kernel -C none  
-a 0x00008000 -e 0x00008000  
-n Linux-Kernel-Image -d zImage wut.kernel  
  
cp wut.kernel /tftpboot
```

Beispiel: Kernel für JFFS2-Root-FS flashen II

U-Boot

Zielsystem

```
U-Boot> tftp wut.kernel
```

```
U-Boot> nand erase 100000 2000000
```

```
U-Boot> nand write 800000 100000 2000000
```

```
U-Boot> setenv bootargs 'console=ttyS1,115200  
root=/dev/mtdblock3 rw rootfstype=jffs2'
```

```
U-Boot> setenv bootcmd  
'nand read 800000 100000 2000000;  
bootm 800000'
```

```
U-Boot> saveenv
```

```
U-Boot> boot
```

6

Dateisysteme

- Dateisysteme-Übersicht
- unmanaged Flash (Raw-Flash)
- managed Flash (FTL-Flash)

Dateisysteme - Übersicht

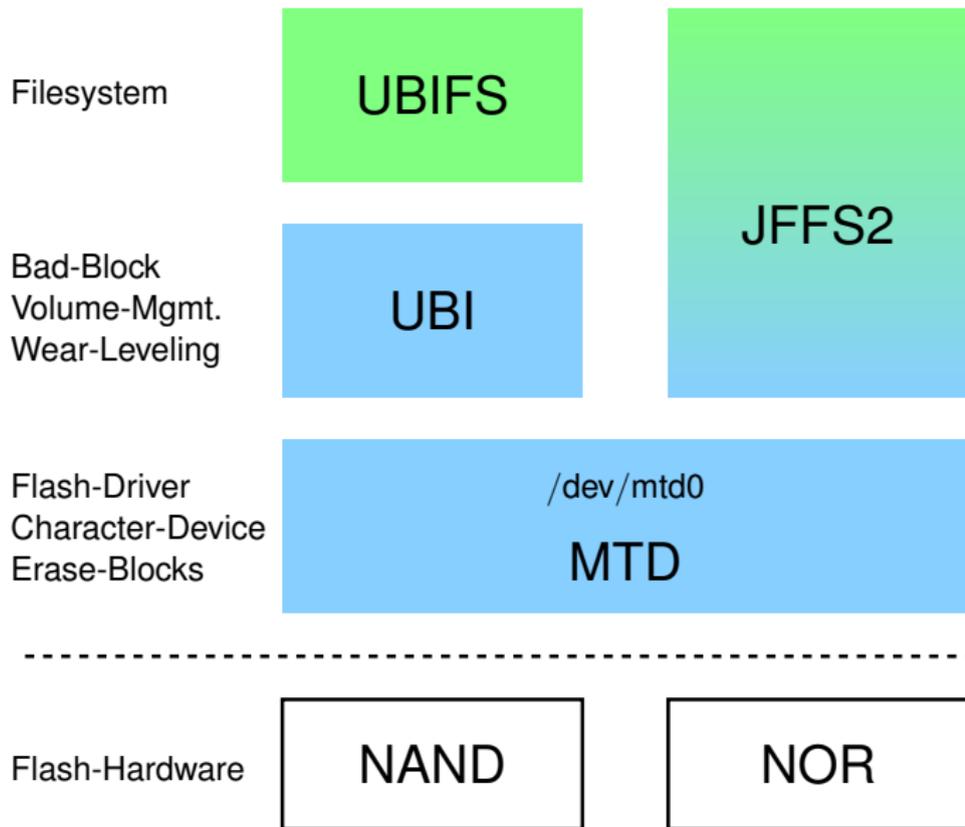
Anwendungsfall	Dateisystem	Beispiele
NAND- / NOR-Flash	jffs2 ubifs	Root-FS auf Flash; schreibbare Konfiguration
Flash mit Controller	ext2, ext3 fat	Root-FS auf CF-/SD-Card, ... Massenspeicher
„flüchtige“ Daten	tmpfs ramfs	/tmp, /var, ... InitRD beim Booten
rein lesbare Daten, komprimiert	squashfs	Archiv-Daten
Netzwerk	nfs, cifs	Unix-Netzwerk heterogenes Netz (Samba)
Pseudo-Dateisysteme	procfs, sysfs debugfs	Systeminformationen Debugging u. Tracing

- kein FTL-Controller für Schreib-Lese-Zyklen-Optimierung
- Erase-Block als kleinster löschbarer Bereich; z. B. 128 kB
- Page- bzw. Sub-Page als kleinste schreibbare Datenmenge; z. B. 2 kB bei NAND; bei NOR bis zu 1 Byte
- **Schreib-Lese-Zyklus** besteht aus:
einem Löschvorgang (Erase) plus
einem Schreibvorgang (Write) plus
beliebig vielen Lesevorgängen (Read)
- Anzahl an Schreib-Lese-Zyklen bezieht sich auf Erase-Block und ist technologisch begrenzt; z. B. 100.000

Anforderungen

- Kommunikation mit Flash
 - Treiber für unterschiedliche Flash-Technologien (MTD-Schicht: Memory Technology Devices)
- Dateisystem muß Erase-Write-Read-Zyklen unterstützen
 - spezielle Flash-Filesysteme (jffs2, ubifs, yaffs, logfs)
- Anwendungsdesign
 - Schreiboperationen minimieren bzw. auslagern in RAM-basiertes Filesystem
 - Beispiel: `/var/log` und `/tmp` im Hauptspeicher als tmpfs

Raw-Flash-Support



UBI-FS (I)

```
flash_erase /dev/mtd4 0 0
ubiformat /dev/mtd4 -s 512
ubinformat
...
UBI control device major/minor: 10:62

mknod /dev/ubictl c 10 62
ubiattach /dev/ubictl -m 4
```

Zielsystem

nur in Entwicklung

UBI-FS (II)

Zielsystem

```
cat /sys/class/ubi/ubi0/dev
253:0
mknod /dev/ubi0 c 253 0

ubimkvol /dev/ubi0 -N mydaten -m

mount -t ubifs ubi0:mydaten /mnt
```

UBI-FS als Root-FS

Zielsystem

Kernel-Kommandozeile:

```
ubi.mtd=data root=ubi0:mydaten rootfstype=ubifs
```

`ubi.mtd=data` — Name der MTD-Partition (`/proc/mtd`)

`root=ubi0:mydaten` — 1. UBI-Device und Volume-Name

UBI-FS löschen

Zielsystem

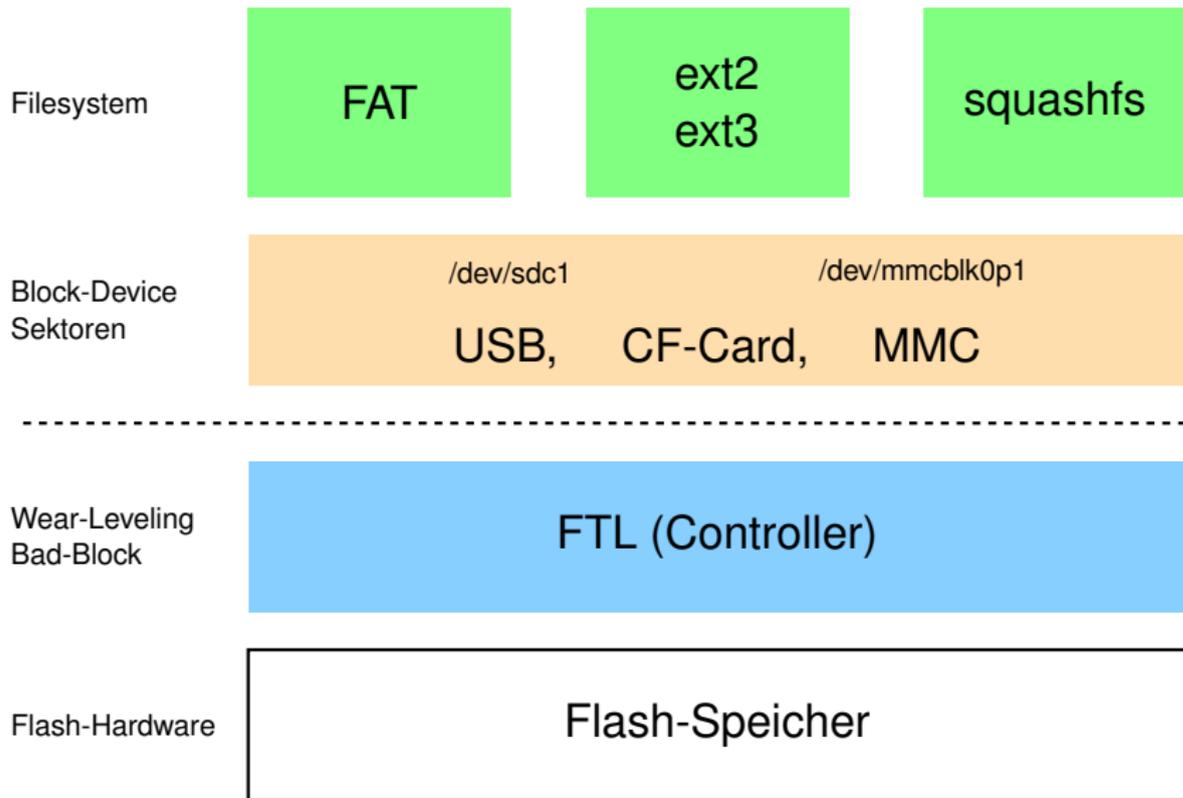
```
umount /mnt
```

```
ubirmvol /dev/ubi0 -N data
```

```
ubidetach /dev/ubictl4 -m 4
```

- Beispiele: USB-Stick, CF-Card, ...
- kein spezielles Flash-Filesystem notwendig
- handhabbar wie gewöhnliche Festplatte; frei partitionierbar
- „normale“ Dateisysteme verwendbar (ext2, ext3, fat)
 - diese optimal konfigurieren
 - ⇒ häufiges und zyklisches Schreiben vermeiden
- Controller optimiert Schreib-/Lese-Zyklen-Beschränkung
 - Spezifikation und Grenzen des Flash beachten
 - große Qualitätsunterschiede
- keinen Swap-Space anlegen

FTL-Flash-Support



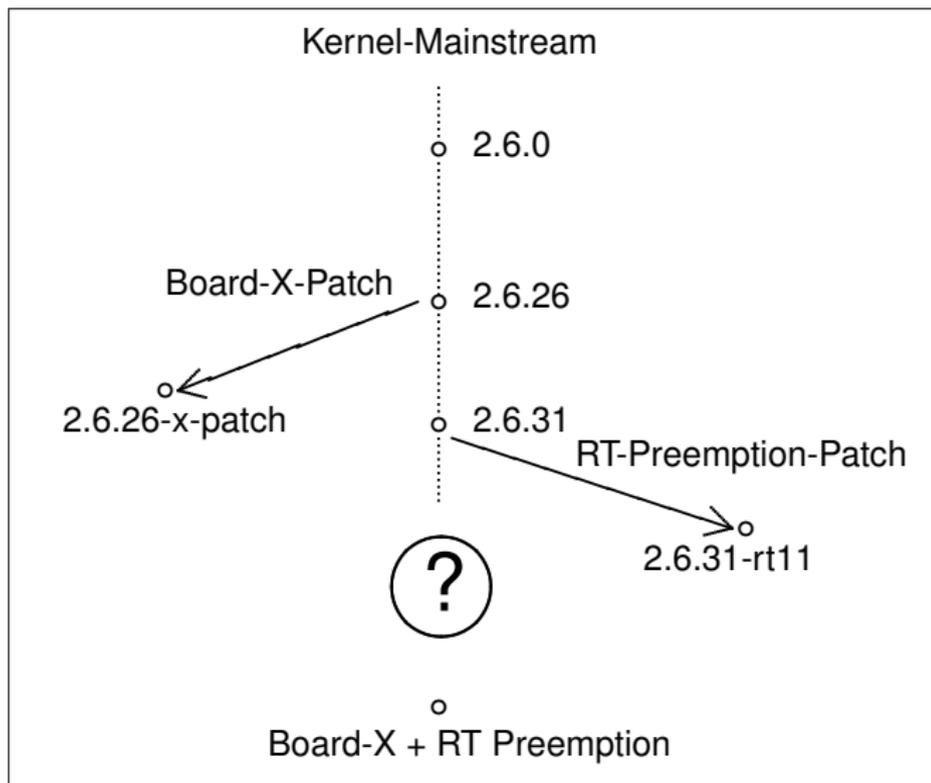
7 Ausblick

Erstellvorgang reproduzierbar machen

- bei der Installation zusätzlicher Pakete auf einem Linux-System ändert sich ggf. der Erstellvorgang
- Erstellung der Toolchain und aller Hilfsprogramme in einem eigenen Unterverzeichnis
- Kopieren oder Verlinken der Sourcecodes in das Toolchain-Verzeichnis
- Wechsel in dieses Unterverzeichnis mittels z. B.
`chroot /usr/arm-linux /bin/bash`
- Erstellvorgang starten
- Change-Root-Umgebung verlassen:
`exit`

Welchen Kernel soll ich verwenden?

- mit Mainstream-Kernel kommt man in den Genuß zukünftiger Verbesserungen und Entwicklungen
- gegebenenfalls sind Patches notwendig:
 - Hardware-Architektur
 - Echtzeitfähigkeit (RT-Preemption-Patch)



- Eigenschaften der notwendigen Komponenten können im Vorfeld abgeklärt werden

- Buildprozess vor der Auswahl des Targets testen
 - ⇒ Unverträglichkeiten und Probleme treten zutage
 - ⇒ Abschätzung, ob Feature X den Aufwand Y wert ist
 - ⇒ Eckdaten des Zielsystems werden messbar

- Zusammenspiel von Komponenten testen

- es existieren viele gut funktionierende Buildumgebung
 - ⇒ an Problemen dranbleiben und sich Hilfe holen
- Start mit minimaler Default-Konfiguration beginnen und darauf sukzessive aufbauen
 - ⇒ die zu lösenden Probleme sind kleiner und handlicher
- Verfügbarkeit der Quellcodes für alle verwendeten Komponenten sicherstellen
 - ⇒ minimiert ungewollte Abhängigkeiten von Dritten

- funktionierende Kombinationen wählen hinsichtlich:
 - Hostsystem
 - Targetsystem
 - Kernel u. notwendigen Patches (Architektur + Echtzeit)
 - Toolchain (binutils, gcc, Debugging-Werkzeuge)
 - Bibliotheken (μ Clibc, glibc, dietlibc)
- in der riesigen Auswahl an Optionen und verfügbarer Software das Ziel immer im Auge behalten