

Ein ganzes GNU/Linux

Hans Buchmann FHNW/ISE

9. April 2020

Um was geht es ?

- ▶ ein GNU/Linux von Grund auf bauen
 - ▶ nicht mehr so schwer wie auch schon
- ▶ ein kleines angepasstes GNU/Linux
 - ▶ grosse GNU/Linux gibt es schon
- ▶ nicht völlig automatisiert
- ▶ Alternative zu **yocto** (www.yoctoproject.org) & Co.

Ziel

GNU/Linux auf dem **BeagleBoneGreen**

- ▶ command based
- ▶ Ethernet/Wi-Fi
- ▶ ssh
- ▶ sshfs
- ▶ moderne Toolchain inkl. *c++17* **C++**

Remark: parallel zu GNU/Linux bauen wir die Toolchain

Komponenten BeagleBoneGreen und *Host*

BeagleBoneGreen

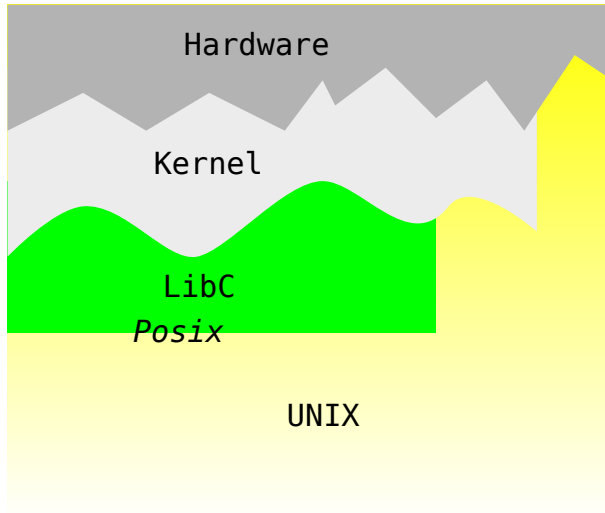
Kernel wenige Files (zwei)

root ein Filesystem viele Files

Host

Toolchain binutils, gcc, Bibliotheken für den Compiler

Übersicht



Die Komponenten für BeagleBoneGreen

Hardware **BeagleBoneGreen**

Kernel zugeschnitten auf **BeagleBoneGreen**

▶ github.com/beagleboard/linux

root das Filesystem

LibC glibc

▶ www.gnu.org/software/libc/index.html

UNIX busybox

▶ www.busybox.net/

... Weitere UNIX basierte Komponenten

▶ das configure, make, make install
Triple

Toolchain

binutils linker & Co.

gcc compiler

- ▶ libgcc die Bibliothek für den Compiler

Remark(s):

- ▶ die Toolchain muss zweimal gebaut werden
 - ▶ für den **kernel** und libc
 - ▶ für UNIX/**POSIX**
- ▶ das target
 - ▶ cpu-vendor-os

Die Verzeichnisstruktur

somewhere_on_the_host

```

├── tools
│   ├── config.sh ..... used in (all) scripts
│   └── component.sh ..... how to build
├── build ..... home of the build files
│   └── component ..... directory
├── target-root ..... top of target root
├── tc ..... the new toolchain
├── config ..... of some components
└── mount ..... for mounting the BBG (sshfs)

```


Toolchain

tc

- ▶ die grossen zwei:
 - ▶ Compiler
 - ▶ Linker
- ▶ kleinere Programme:
 - ▶ Assembler
 - ▶ ...

Toolchain

Beispiel

- ▶ Sourcefile `{c|cc}-source.{c|cc}`
- ▶ Compilat/object File `{c|cc}-source.o`
- ▶ Executable/Image `{c|cc}-source`

Cross toolchain

2 Verschiedene Rechner

Host Workstation leistungsfähiger Rechner

Target Eingebettetes System (**BeagleBoneGreen**)

Cross{Programm} Programm (Compiler etc.) das

- ▶ läuft auf dem *Host* und erzeugt Files für das *Target*

Cross toolchain

- ▶ erzeugt auf dem *Host* Programme für das *Target*

GNU/Toolchain

Zwei Komponenten

`binutils` Linker, assembler, ...

`gcc` Compiler

Build

die drei Schritte

- ▶ configure
- ▶ make
- ▶ make install

Remark: auf dem **Host**

Build der Kontext

prefix wo die Toolchain auf dem *Host* installiert wird

▶ option `--prefix=path-to-toolchain-install`

sysroot wo ist das *Target* root system (auf dem *Host*)

▶ option `--with-sysroot=path-to-target-sysroot`

target was für eine *Target* System

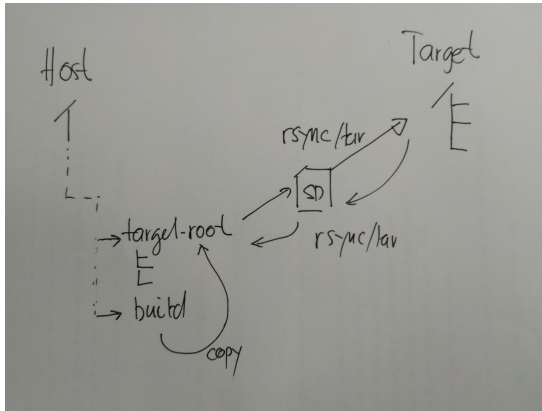
▶ option `--target=armv6l-unknown-linux-gnueabi`

Remark: Warum ???

Prinzip

- ▶ wir sind in `17-build`
- ▶ pro Komponente ein Skript in `tools`
- ▶ pro Komponente ein Unterverzeichnis in `build`
- ▶ der File `tools/common.sh`
 - ▶ Pfadnamen

Das BeagleBoneGreen Rootfilesystem an zwei Orten



Build 1

Toolchain 1

- ▶ `binutils.sh`
- ▶ `gcc-bare.sh`
 - ▶ nur für den *kernel*
 - ▶ das bare minimum
 - ▶ nur C

Build 2

Kernel

- ▶ `kernel.sh` mit ein paar *targets*
 - ▶ `bb.org_defconfig`
 - ▶ `zImage`
 - ▶ `headers_install`
 - ▶ Interface: *kernel-libc*

Build 3

libc

Wir brauchen glibc

▶ glibc

Build 4

Toolchain 2

- ▶ `gcc.sh`
 - ▶ mit `sysroot`
 - ▶ C und C++
- ▶ Test
 - ▶ im Verzeichnis `work`

Build 5

busybox

- ▶ `busybox.sh`
 - ▶ Installation auf SD-Card
 - ▶ `fakEROOT`

Skripts und Argumente

initiales System A

Skript	script.sh what	gebraucht für
binutils.sh		alle toolchains
gmp.sh		alle toolchains
mpfr.sh		alle toolchains
mpc.sh		alle toolchains
gcc-bare.sh		kernel, libc

Remark: Alle Skripte sind **bash** Skripte

Skripts und Argumente

initiales System B

Skript	script.sh what	gebraucht für
kernel.sh	defconfig zImage headers_install	
glibc.sh		POSIX
gcc.sh		C/C++, POSIX
busybox.sh	menuconfig busybox install	
target-root.sh		vervollständigt target-root

Remark: Alle Skripte sind **bash** Skripte

Target

erster Versuch

- ▶ transfer auf SD Karte
- ▶ Internet

Skripts und Argumente

ssh

zlib.sh

openssl.sh die kryptographischen Algorithmen

openssh.sh

Remark: openssh.sh hängt von zlib.sh und openssl.sh ab

ssh

- ▶ openssh die volle Implementation
 - ▶ zlib
 - ▶ openssl
 - ▶ openssh

WiFi

- ▶ kernel
 - ▶ Network/WiFi
 - ▶ Drivers/Network/WiFi/TI
 - ▶ → Firmware
- ▶ rootfs
 - ▶ libnl.sh
 - ▶ wpa_supplicant.sh
 - ▶ configuration

Workflow

Begriffe

`target-root` Verzeichnis auf dem *Host*

- ▶ enthält das **BeagleBoneGreen** Rootfilesystem
- ▶ soll aktuell sein

`SD-Card` Speicherkarte mit dem **BeagleBoneGreen** Rootfilesystem

- ▶ entspricht `target-root`

target-root - SD-Card

tar rsync

	target-root		SD-Card
initiales GNU/Linux	→	tar	→
SD-Card	←	rsync	←
target-root	→	rsync	→

sshfs

funktioniert noch nicht

- ▶ Die Bibliothek `glib`
- ▶ Ersatz
 - ▶ `sftp`

configure-make-make install

Installation neuer Komponenten

- ▶ aus den Quellen
- ▶ immer etwa gleich
 - ▶ `download`
 - ▶ `configure options`
 - ▶ `make`
 - ▶ `make install`
- ▶ Unterschiede in den Details