

Ein ganzes GNU/Linux

Hans Buchmann FHNW/ISE

4. Dezember 2018

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 **BeagleBoneBlack**

- ▶ command based
- ▶ Ethernet/Wi-Fi
- ▶ ssh
- ▶ sshfs
- ▶ moderne Toolchain inkl. *c++14* **C++**

Remark: parallel zu GNU/Linux bauen wir die Toolchain

Komponenten BeagleBoneBlack und *Host*

BeagleBoneBlack

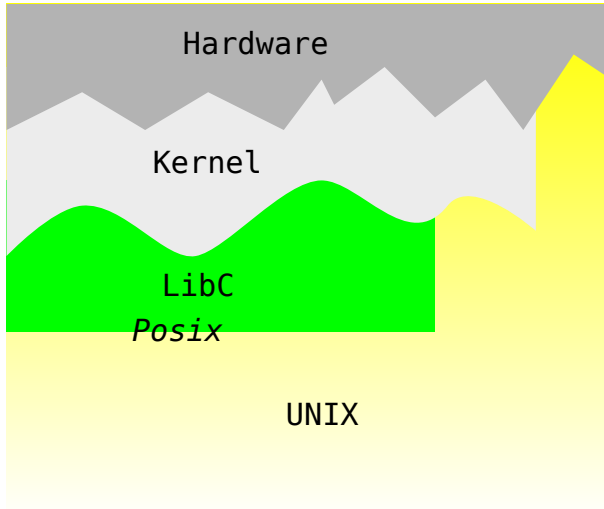
Kernel ein File

root ein Filesystem

Host

Toolchain binutils, gcc

Übersicht



Die Komponenten für BeagleBoneBlack

Hardware **BeagleBoneBlack**

Kernel zugeschnitten auf **BeagleBoneBlack**

▶ 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

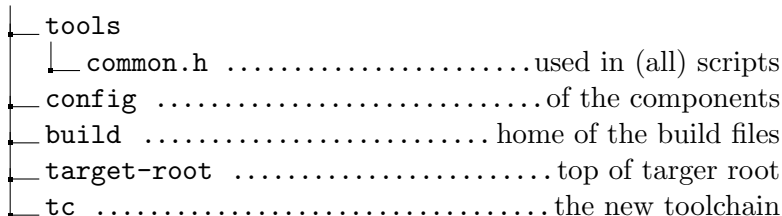
- ▶ libgcc die Bibliothek für den Compiler

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



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 (**BeagleBoneBlack**)

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

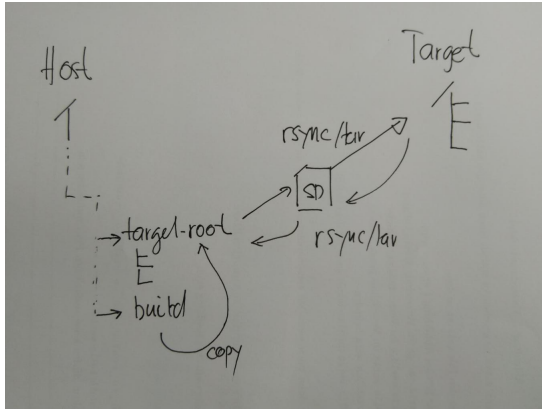
- ▶ option `--target=armv6l-unknown-linux-gnueabi`

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 BeagleBoneBlack 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

| Skript | target | gebraucht für |
|----------------|--|----------------------|
| binutils.sh | | alles |
| gcc-bare.sh | | kernel, libc |
| kernel.sh | defconfig zImage headers_install | |
| glibc.sh | | POSIX |
| gcc.sh | | C/C++, POSIX |
| busybox.sh | busybox install | |
| target-root.sh | | Transfer auf SD-Card |

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

Workflow

Begriffe

`target-root` Verzeichnis auf dem *Host*

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

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

- ▶ entspricht `target-root`

target-root - SD-Card

`tar rsync`

| | target-root | | SD-Card |
|---------------------|-------------|--------------------|---------|
| initiales GNU/Linux | → | <code>tar</code> | → |
| SD-Card | ← | <code>rsync</code> | ← |
| target-root | → | <code>rsync</code> | → |

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

`rsync.samba.org`
als Beispiel

- ▶ auf dem *Host*
- ▶ auf dem **BBB**

Verzeichnisstruktur

- ▶ Source:rsync-3.1.2
- ▶ Build: für die (vielen) Zwischenfiles
- ▶ Install: prefix
- ▶ rsync.sh: das Skript

Skript: `rsync.sh` schrittweise für den *Host*

- ▶ `configure --help`
- ▶ `configure --prefix`
 - ▶ prefix: wohin kommt das Resultat
 - ▶ Files in `rsync-build`
- ▶ `make`
 - ▶ Files in `rsync-build`
- ▶ `make install`
 - ▶ Files in `prefix`

Aufgabe Skript: `rsync.sh` für **BBB**

- ▶ Crosscompile `--sysroot`
- ▶ `--prefix`
- ▶ `DESTDIR`

Remark: `tools/rsync.sh`

Ein ganzes GNU/Linux

Hans Buchmann FHNW/ISE

4. Dezember 2018

Teil I

Build again

- ▶ Toolchain:
 - ▶ bare
- ▶ Kernel:
 - ▶ ohne module
 - ▶ ohne unötigen *drivers*
 - ▶ USB Gadgets
- ▶ libc:
 - ▶ glibc
- ▶ Toolchain:
 - ▶ voll inkl C++
- ▶ busybox:

Supplement Internet

- ▶ ssh
 - ▶ zlib
 - ▶ openssl
 - ▶ openssh
- ▶ init
 - ▶ usb - ethernet
 - ▶ ssh
 - ▶ **neu** _{ntp}
 - ▶ network time protocol

Supplement Wi-Fi

- ▶ wpa_supplicant
 - ▶ Configuration

- ▶ die Verbindung **RootFS**
 - ▶ *Host* ↔ **BBB**
- ▶ Versionen
 - ▶ wie ist ein RootFS erzeugt worden